Nichtnumerische Algorithmen

Andreas Hasenkopf
Michael Kugler

Inhaltsverzeichnis

Teil I
Funktionen, Algorithmen, Probleme

1 Grundlagen

1.1 Funktionen und Rekursion

Bei der rekursiven Programmierung ruft sich eine Prozedur, Funktion oder Methode in einem Computerprogramm selbst wieder auf. Auch der gegenseitige Aufruf stellt eine Rekursion dar. Wichtig dabei ist eine Abbruchbedingung in dieser Funktion, weil sich das rekursive Programm sonst (theoretisch) bis in alle Ewigkeit selbst aufrufen würde. Die rekursive Programmierung findet oft Anwendung in prozeduralen und objektorientierten Programmiersprachen. Obwohl diese Sprachen in ihrem Sprachstandard die Rekursion ausdrücklich zulassen, stellen Selbstaufrufe und gegenseitige Aufrufe bei der Programmierung die Ausnahme dar. Die meisten Programme in diesen Sprachen sind rein iterativ.

In einigen Sprachen, wie z. B. in manchen funktionalen Programmiersprachen oder Makroprozessoren, muss die rekursive Programmiermethode zwingend verwendet werden, da iterative Sprachkonstrukte fehlen.

Ein Beispiel für die Verwendung einer rekursiven Programmierung ist die Berechnung der Fakultät einer Zahl. Die Fakultät ist das Produkt aller ganzen Zahlen von 1 bis zu dieser Zahl. Die Fakultät von 4 ist also $1 \cdot 2 \cdot 3 \cdot 4 = 24$.

Mathematiker definieren die Fakultät so (eine rekursive Definition):

$$n! = n \cdot (n-1)! \quad (1)$$

$$1! = 1 \quad (2)$$

Man definiert eine Funktion fac, die eine Zahl x als Eingabewert bekommt. Diese Funktion multipliziert x mit dem Rückgabewert von fac(x-1) außer x=1, dann liefert die Funktion das Ergebnis 1. Dies ist die Abbruchbedingung:

```
int fac(int n){
        if (n==1) return 1;
        return fac(n-1)*n;
}
```

Rekursive Programme haben in der Regel keine gute Performance. Durch die wiederholten Funktionsaufrufe wird immer wieder derselbe Methodeneintrittscode bearbeitet und jedesmal der Kontext gesichert, was zu hohem Arbeitsspeicherverbrauch führt. Alle rekursiven Algorithmen lassen sich jedoch auch durch iterative Programmierung implementieren (und umgekehrt). Man hätte die Fakultät auch so implementieren können:

```
int i,p;
for (i=1; i<=n; i++){
        p = p*i;
}
```

Hierbei gilt die Regel, dass für einfache Probleme eine iterative Implementierung häufig effizienter ist. So sollte z. B. auch die Fakultätsfunktion der Effizienz

wegen in der Praxis iterativ implementiert werden. Bei komplizierten Problemstellungen (z. B. Aufgaben mit Bäumen) hingegen lohnt sich oftmals der Einsatz einer rekursiven Lösung, da für solche Probleme eine iterative Formulierung schnell sehr unübersichtlich werden kann.

Nicht alle höheren Programmiersprachen lassen rekursive Aufrufe zu; ein Beispiel dazu ist Fortran. Andere Programmiersprachen sind dagegen grundsätzlich rekursiv (wie z. B. Prolog). Solche rekursiven Programmiersprachen (und auch andere Sprachen wie z. B. Scheme) setzen die Rekursion meistens effizient um.

1.2 Landau - Symbole

Landau-Symbole werden in der Mathematik und in der Informatik verwendet, um das asymptotische Verhalten von Funktionen und Folgen zu beschreiben. In der Informatik werden sie insbesondere in der Komplexitätstheorie verwendet, um verschiedene Probleme und Algorithmen danach zu vergleichen, wie "schwierig" oder aufwendig sie zu berechnen sind.

Der Großbuchstabe O (damals eigentlich ein großes Omikron) als Symbol für Ordnung von wurde erstmals vom deutschen Zahlentheoretiker Paul Bachmann in der 1894 erschienenen zweiten Auflage seines Buchs Analytische Zahlentheorie verwendet. Bekannt gemacht wurde diese Notation durch den ebenfalls deutschen Zahlentheoretiker Edmund Landau, mit dessen Namen sie insbesondere im deutschen Sprachraum heute in Verbindung gebracht wird.

Groß O und klein o sind die am häufigsten verwendeten Landau-Symbole; darüber hinaus gibt es noch Ω, ω und θ.

Sie vergleichen das Wachstum von zwei Funktionen, meist im Unendlichen. Dabei ist typischerweise f die Folge oder Funktion, über die eine Aussage getroffen werden soll, und g der einfachste Repräsentant einer Klasse gleich schnell wachsender Funktionen, die als Vergleich dienen.

Notation	Anschauliche Bedeutung
$f \in \mathrm{O}(g)$	f wächst höchstens so schnell wie g
$f \in \mathrm{o}(g)$	f wächst langsamer als g
$f \in \Omega(g)$	f wächst mindestens so schnell wie g
$f \in \omega(g)$	f wächst schneller als g
$f \in \Theta(g)$	f wächst genauso schnell wie g

In der folgenden Tabelle bezeichnen f und g entweder

- Folgen reeller Zahlen, dann ist $x \in \mathbb{N}$ und der Grenzwert $a = \infty$, oder
- reellwertige Funktionen der reellen Zahlen, dann ist $x \in \mathbb{R}$ und der Grenzwert aus den erweiterten reellen Zahlen: $a \in \mathbb{R} \cup \{-\infty, +\infty\}$, oder
- reellwertige Funktionen beliebiger topologischer Räume $(X, \mathfrak{T})$, dann ist $x \in X$ und auch der Grenzwert $a \in X$. Wichtigster Spezialfall ist dabei $X = \mathbb{R}^n$.

Formal lassen sich die Landau-Symbole dann mittels Limes superior und Limes inferior folgendermaßen definieren:

$$f \in \mathrm{O}(g) \quad \Leftrightarrow \quad 0 \leq \limsup_{x \to a} \left| \frac{f(x)}{g(x)} \right| < \infty \tag{3}$$

$$f \in \mathrm{o}(g) \quad \Leftrightarrow \quad 0 = \lim_{x \to a} \left| \frac{f(x)}{g(x)} \right| \tag{4}$$

Beispiel:
Die Landau-Notation wird verwendet, um das asymptotische Verhalten bei Annäherung an einen endlichen oder unendlichen Grenzwert zu beschreiben. Das große O wird verwendet, um eine maximale Größenordnung anzugeben. So gilt beispielsweise nach der Stirling-Formel für das asymptotische Verhalten der Fakultät

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \text{ für } n \to \infty \tag{5}$$

und

$$n! = \mathrm{O}\left(\sqrt{n} \cdot n^n\right) \text{ für } n \to \infty. \tag{6}$$

Der Faktor $\sqrt{2\pi}$ ist dabei nur eine Konstante und kann für die Abschätzung der Größenordnung vernachlässigt werden. Die Landau-Notation kann auch benutzt werden, um den Fehlerterm einer Approximation zu beschreiben. Beispielsweise besagt

$$e^x = 1 + x + x^2/2 + \mathrm{O}(x^3) \qquad \text{für } x \to 0 \tag{7}$$

dass der Absolutbetrag des Approximationsfehlers kleiner als eine Konstante mal x^3 für x hinreichend nahe bei Null. Das kleine o wird verwendet, um zu sagen, dass ein Ausdruck vernachlässigbar klein gegenüber dem angegebenen Ausdruck ist. Für differenzierbare Funktionen gilt beispielsweise

$$f(x+h) = f(x) + hf'(x) + \mathrm{o}(h) \qquad \text{für } h \to 0, \tag{8}$$

der Fehler bei Approximation durch die Tangente geht also schneller als linear gegen 0.

In der Komplexitätstheorie werden die Landau-Symbole vor allem angewendet, um den (minimalen oder maximalen) Bedarf an Speicher (Platzkomplexität) und Zeit (Zeitkomplexität) bezüglich eines bestimmten Maschinenmodells zu beschreiben.

Normalerweise ist es sehr aufwändig oder ganz unmöglich, für ein Problem L eine Funktion $f_L : w \to f_L(w)$ anzugeben, die allgemein jeder beliebigen Eingabe für ein Problem die zugehörige Anzahl der Rechenschritte (bzw. der Speicherzellen) zuordnet. Daher begnügt man sich in der Regel damit, statt jede Eingabe einzeln zu erfassen, sich lediglich auf die Eingabelänge $n = |w|$ zu beschränken. Es ist aber meist ebenfalls zu aufwändig, eine Funktion $f_L : n \to f_L(n), n = |w|$ anzugeben.

Daher hat man die Landau-Notation entwickelt, die sich auf das asymptotische Verhalten der Funktion f_L beschränkt. Man betrachtet also, in welchen Schranken sich der Rechenaufwand (der Bedarf an Speicher und Rechenzeit)

hält, wenn man die Eingabe vergrößert. Das wichtigste Landau-Symbol ist O, mit dem man obere Schranken angeben kann; untere Schranken sind im allgemeinen viel schwieriger zu finden. Dabei meint $f = \mathrm{O}(g)$, dass eine Konstante c und ein $n_0 \in \mathbb{N}$ existieren, so dass für alle $n > n_0$ gilt: $f(n) \leq c \cdot g(n)$. In anderen Worten: Für alle Eingabelängen ist der Rechenaufwand $f(n)$ nicht wesentlich größer, d. h. höchstens um einen konstanten Faktor c, als $g(n)$.

Dabei ist die Funktion f nicht immer bekannt: Die Landau-Notation ist gerade dazu da, den Rechenaufwand (Platzbedarf) abzuschätzen, wenn es zu aufwändig ist, die genaue Funktion anzugeben, bzw. wenn diese zu kompliziert ist.

Die Landau-Symbole erlauben es dadurch, Probleme und Algorithmen nach ihrer Komplexität in Komplexitätsklassen zusammenzufassen.

In der Komplexitätstheorie lassen sich die verschiedenen Probleme und Algorithmen dann folgendermaßen vergleichen: Man kann für Problemstellungen mit eine untere Schranke für beispielsweise die asymptotische Laufzeit angeben, mit O entsprechend eine obere Schranke. Bei $O(f)$ wird die Form von f (z.B. $f(n) = n^2$) auch als die Komplexitätsklasse oder Aufwandsmaß bezeichnet (also z.B. quadratisch). Bei dieser Notation werden, wie die Definitionen der Landau-Symbole zeigen, konstante Faktoren vernachlässigt. Dies ist gerechtfertigt, da die Konstanten zu großen Teilen vom verwendeten Maschinenmodell bzw. bei implementierten Algorithmen von der Qualität des Compilers und diversen Eigenschaften der Hardware des ausführenden Computer abhängig sind. Damit können sie nicht direkt mit der Laufzeit des Algorithmus in Verbindung gebracht werden.

2 Algorithmus - Was ist das?

Unter einem Algorithmus versteht man allgemein eine genau definierte Handlungsvorschrift zur Lösung eines Problems oder einer bestimmten Art von Problemen.

Im täglichen Leben lassen sich leicht Beispiele für Algorithmen finden: Zum Beispiel ist ein Kochrezept ein Algorithmus zumindest dann, wenn alle Angaben genau genug sind und es für alle Teilaufgaben, wie Braten, Rühren, etc., ebenfalls Algorithmen gibt. Auch Reparatur- und Bedienungsanleitungen oder Hilfen zum Ausfüllen von Formularen sind in der Regel Algorithmen. Ein weiteres, etwas präziseres Beispiel sind Waschmaschinenprogramme.

Das Wort Algorithmus ist eine Abwandlung oder Verballhornung des Namens von Muhammed Al Chwarizmi (* ca. 783; † ca. 850), einem Perser, dessen arabisches Lehrbuch Über das Rechnen mit indischen Ziffern (um 825) in der mittelalterlichen lateinischen Übersetzung mit den Worten "Dixit Algorismi" begann. Im Mittelalter wurde daraus lat. algorismus (mit lat. Varianten wie alchorismus, algoarismus, altfranzösisch algorisme, argorisme, mittel-englisch augrim, augrym) als Bezeichnung für die Kunst des Rechnens mit den arabischen Ziffern und als Titel für Schriften über diese Kunst.

Die lateinischen Autoren pflegten zu erklären, dass das Wort 'algorismus' aus dem Namen des Erfinders dieser Kunst, einem Philosophen namens Algus, und dem griechischen Wort rismus (rhythmós) für 'Zahl' zusammengesetzt sei. Dabei wurde Algus von einigen als Araber, von anderen als Grieche oder zumindest griechisch schreibender Autor, oder gelegentlich auch als 'König von Kastilien'

(Johannes von Norfolk) betrachtet. In der volkssprachlichen Tradition erscheint dieser 'Meister Algus' dann zuweilen in einer Reihe mit großen antiken Schriftstellern wie Platon, Aristoteles und Euklid, so im altfranzösischen Roman de la Rose, während das altitalienische Gedicht Il Fiore ihn sogar als Erbauer des Schiffes Argo ausgibt, mit dem Jason sich auf die Suche nach dem Goldenen Vlies begab.

Durch Gräzisierung der Schreibweise des angenommen griechischen Wortbestandteiles rismus hat sich dann in der lateinischen Wissenschaftssprache die Schreibung 'algorithmus' ergeben, und in dieser Form hat sich das Wort später als Fachterminus für geregelte Prozeduren zur Lösung definierter Probleme eingebürgert.

2.1 Definition

2.1.1 Turing - Maschinen und Algorithmusbegriff

In der ersten Hälfte des 20. Jahrhunderts wurden eine ganze Reihe von Ansätzen entwickelt, um zu einer genauen Definition zu kommen. Formalisierungen des Berechenbarkeitsbegriffs sind die Turing-Maschine (Alan Turing), der Lambda-Kalkül (Alonzo Church), rekursive Funktionen, Chomsky-Grammatiken und Markow-Algorithmen.

Es wurde unter maßgeblicher Beteiligung von Alan Turing selbst gezeigt, dass all diese Methoden ebenso leistungsfähig (gleich mächtig) sind. Sie können durch eine Turing-Maschine emuliert werden, und sie können umgekehrt eine Turing-Maschine emulieren.

Mit Hilfe des Begriffs der Turing-Maschine kann folgende formale Definition des Begriffs formuliert werden:

> Eine Berechnungsvorschrift zur Lösung eines Problems heißt genau dann Algorithmus, wenn eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert, die für jede Eingabe, die eine Lösung besitzt, stoppt.

Aus dieser Definition sind folgende Eigenschaften eines Algorithmus ableitbar:

1. Das Verfahren muss in einem endlichen Text eindeutig beschreibbar sein (Finitheit)
2. Jeder Schritt des Verfahrens muss auch tatsächlich ausführbar sein (Ausführbarkeit)
3. Das Verfahren darf zu jedem Zeitpunkt nur endlich viel Speicherplatz benötigen (Dynamische Finitheit, siehe Platzkomplexität)
4. Das Verfahren darf nur endlich viele Schritte benötigen (Terminierung, siehe auch Zeitkomplexität)

Darüber hinaus wird der Begriff Algorithmus in praktischen Bereichen oft auf die folgenden Eigenschaften eingeschränkt:

1. Der Algorithmus muss bei denselben Voraussetzungen das gleiche Ergebnis liefern (Determiniertheit)

2. Die nächste anzuwendende Regel im Verfahren ist zu jedem Zeitpunkt eindeutig definiert (Determinismus)

2.1.2 Church - Turing - These

Die Church-Turing-These lautet:

> Jedes intuitiv berechenbare Problem kann durch eine Turingmaschine gelöst werden.

Als formales Kriterium für einen Algorithmus zieht man die Implementierbarkeit in einem beliebigen zu einer Turing-Maschine äquivalenten Formalismus heran, insbesondere die Implementierbarkeit in einer Programmiersprache die von Church verlangte Terminiertheit ist dadurch allerdings noch nicht gegeben.

Der Begriff der Berechenbarkeit ist dadurch dann so definiert, dass ein Problem genau dann berechenbar ist, wenn es einen (terminierenden) Algorithmus zu dem Problem gibt, d. h. wenn eine entsprechend programmierte Turing-Maschine das Problem in endlicher Zeit lösen könnte.

2.2 Eigenschaften

Nichtdeterministische Algorithmen finden vor allem in der Theoretischen Informatik Anwendung, so, dass in anderen Bereichen oft vorausgesetzt wird, dass es sich um einen deterministischen Algorithmus handelt. Eine Ausnahme bilden sogenannte stochastische randomisierte oder probabilistische Algorithmen, in die absichtlich ein Zufallsfaktor eingebaut wurde. Solche Algorithmen sind demnach nicht deterministisch und auch nicht determiniert. Stochastische Algorithmen dagegen sind im Allgemeinen deterministisch, orientieren sich aber an Erfahrungswerten.

Die verschiedenen formalen Eigenschaften in Kürze:

- **Determiniertheit**
  Bei jeder Ausführung mit gleichen Startwerten muss das gleiche Ergebnis berechnet werden.

  Algorithmen sind determiniert, wenn sie bei gleichen Parametern und Startwert stets das gleiche Resultat liefern. Das trifft zum Beispiel nicht für randomisierte Algorithmen zu, bei denen das Ergebnis zu einem gewissen Grad auf Zufall beruht.

- **Determinismus**
  Es darf immer nur eine Möglichkeit vorhanden sein.

  Deterministisch heißen alle Algorithmen, bei denen zu jedem Zeitpunkt der Ausführung maximal eine Möglichkeit der Programmfortsetzung besteht. Gibt es mehrere Möglichkeiten der Programmfortsetzung und lassen sich diesen Wahrscheinlichkeiten zuweisen, so spricht man von stochastischen, randomisierten oder probabilistischen Algorithmen. In der theoretischen Informatik gibt es neben dem Determinismus auch den Nichtdeterminismus, der aber in der Praxis kaum Verwendung findet. Zusätzliche Bedeutung bekommen solche nichtdeterministische Algorithmen allerdings durch den Einsatz von Quantencomputern, welche auch solche Algorithmen erfolgreich ausführen.

Es gilt übrigens: Jeder deterministische Algorithmus ist auch determiniert. Nicht jeder determinierte Algorithmus ist jedoch deterministisch.

- **Statische Finitheit**
  Die Beschreibung ist endlich.

  Die Beschreibung eines Algorithmus darf nicht unendlich groß sein. Als statische Finitheit wird die Endlichkeit des Quelltextes bezeichnet. Der Quelltext darf nur eine begrenzte Anzahl, wenn auch bei Bedarf sehr viele Regeln enthalten.

- **Dynamische Finitheit**
  Ein Algorithmus benötigt zu jedem Zeitpunkt seiner Ausführung nur endlich viel Speicherplatz. Diese Bedingung ist notwendig, da keinem Algorithmus unendlich viel Speicherplatz zur Verfügung steht.

- **Terminiertheit**
  Der Algorithmus bricht nach endlicher Zeit kontrolliert ab.

  Algorithmen sind terminierend, wenn sie für jede mögliche Eingabe nach einer endlichen Zahl von Schritten zu einem Ergebnis kommen. Die tatsächliche Zahl der Schritte kann dabei beliebig groß sein.

  Steuerungssysteme und Betriebssysteme und auch viele Programme, die auf Interaktion mit dem Benutzer aufbauen, erfüllen diese Eigenschaft nicht: Wenn der Benutzer keinen Befehl zum Beenden gibt, läuft das Programm endlos weiter. Donald Knuth schlägt in diesem Zusammenhang vor, nicht terminierende Algorithmen als rechnergestützte Methoden ("Computational Methods") zu bezeichnen.

  Es ist übrigens im Allgemeinen nicht für jeden beliebigen Algorithmus möglich zu bestimmen, ob er terminiert oder nicht - siehe Halteproblem.

3 Komplexitätstheorie

3.1 Komplexitätsklasse P

In der Komplexitätstheorie ist P diejenige Komplexitätsklasse, welche die Entscheidungsprobleme enthält, die in Polynomialzeit für deterministische Turingmaschinen lösbar sind. Diese Problemklasse wird allgemein als die Klasse der "praktisch lösbaren" Probleme betrachtet.

Eine Verallgemeinerung von P ist die Klasse NP. Die Probleme aus NP sind zwar ebenfalls in Polynomialzeit entscheidbar, jedoch wird hierfür ein nicht realisierbares, nämlich nichtdeterministisches Maschinenmodell eingesetzt. P ist sicher eine Teilmenge von NP. Es gehört jedoch zu den wichtigsten ungelösten Fragen der theoretischen Informatik, ob auch NP eine Teilmenge von P ist und somit, ob P=NP gilt (siehe auch P/NP-Problem).

3.2 Komplexitätsklasse NP

Die Klasse NP (von englisch Non-deterministic Polynomial time) stammt aus der Komplexitätstheorie. Sie bezeichnet die Klasse aller Entscheidungsprobleme,

die von einer nichtdeterministischen Turingmaschine bezüglich der Eingabe in Polynomialzeit entschieden werden können.

Äquivalent ist ein Entscheidungsproblem genau dann in NP, wenn es von einer deterministischen Turingmaschine in Polynomialzeit gelöst wird, die Zugriff hat auf eine wiederum polynomiell beschränkte, eingabespezifische Zusatzinformation. Die Äquivalenz ergibt sich kurzgefasst dadurch, dass einerseits der Nichtdeterminismus der ersten Turingmaschine ermöglicht, die zusätzliche Information zu erraten, und andererseits die zu einer positiven Antwort führenden Entscheidungen der nichtdeterministischen Turingmaschine als Zusatzinformation kodiert werden können.

Beispielsweise liegt das Problem des Handlungsreisenden in der Klasse NP. Gegeben ist eine Menge von Städten, deren paarweise Entfernungen und eine feste Länge λ. Entschieden werden soll, ob eine Rundreise existiert, bei der jede Stadt nur einmal besucht wird und deren Gesamtlänge λ nicht übersteigt. Die Zusatzinformation ist in diesem Fall einfach eine Rundreise mit genügend kleiner Länge. Dann müssen nur noch die einzelnen Entfernungen addiert und die Summe mit λ verglichen werden.

Gelegentlich wird NP irrtümlich als die Klasse der nicht in Polynomialzeit lösbaren Probleme bezeichnet. Dies ist aber falsch, da insbesondere die Menge der in Polynomialzeit deterministisch lösbaren Probleme eine (von vielen Wissenschaftlern als echt vermutete) Teilmenge von NP ist. Des Weiteren gibt es auch Probleme, die nicht in polynomialer Zeit gelöst werden können und deren Lösungen sich nicht in polynomialer Zeit verifizieren lassen. Diese liegen demnach auch nicht in NP.

Viele Probleme, die in der Komplexitätsklasse NP liegen, insbesondere die NP - vollständigen, lassen sich vermutlich nicht effizient lösen. Alle bekannten Algorithmen erfordern exponentiellen Rechenaufwand und es wird vermutet, dass es keine besseren Algorithmen gibt. Die Bestätigung oder Widerlegung dieser Vermutung wird als P/NP-Problem bezeichnet und gilt als eines der wichtigsten offenen Probleme der Informatik.

Die Klasse der Sprachen, deren Komplemente in NP liegen, heißt CoNP.

3.3 NP - Vollständigkeit

Der Begriff NP-Vollständigkeit ist ein Begriff aus der Komplexitätstheorie der Theoretischen Informatik, der im Jahre 1971 von Stephen A. Cook durch den *Satz von Cook* eingeführt wurde. Er zeichnet spezielle formale Sprachen der Komplexitätsklasse NP aus. So werden solche Sprachen NP - vollständig genannt, die intuitiv gesprochen die vollständige Schwierigkeit aller Sprachen aus der Komplexitätsklasse NP in sich tragen.

Der Begriff benötigt eine Rechtfertigung in dem Sinn, dass überhaupt wenigstens eine derartige Sprache existiert. Die wesentliche Leistung von Cook bestand zunächst darin, diesen Nachweis erbracht zu haben. Heute existieren wesentlich einfachere Nachweise für die Existenz einer solchen Sprache, allerdings sind die dafür verwendeten Sprachen sehr künstlich. Cooks Verdienst besteht also auch darin, für eine besonders interessante Sprache diesen Nachweis erbracht zu haben.

Aufbauend auf der Arbeit von Cook konnte Richard Karp im Jahre 1972 eine weitere bahnbrechende Arbeit vorlegen, die der Theorie der NP-Vollständigkeit zu noch größerer Popularität verhalf. Karps Verdienst besteht darin, die Technik

der Polynomialzeitreduktion konsequent genutzt zu haben, um für weitere 21 populäre Probleme die NP-Vollständigkeit nachzuweisen.

Die Bedeutung der gesamten Theorie begründet sich vor allem auf folgenden Eigenschaften NP - vollständiger Sprachen:

1. Es konnte bisher für keine dieser Sprachen nachgewiesen werden, dass ein Algorithmus existiert, der für ihr Wortproblem in polynomieller Zeit eine korrekte Antwort liefert.

2. Wenn für eine NP - vollständige Sprache ein Algorithmus gefunden wird, der in polynomieller Zeit eine korrekte Antwort liefert, dann kann das Wortproblem für alle Sprachen in der Komplexitätsklasse NP in polynomieller Zeit entschieden werden.

3.3.1 Definition

Ein Problem (genauer: ein Entscheidungsproblem) L heißt NP - vollständig genau dann, wenn:

- L in der Klasse NP liegt, das heißt $L \in NP$ und
- L ist NP-schwer, das heißt $\forall L^* \in NP : L^* \leq_p L$.

Letztere Bedingung bedeutet, dass jedes Problem in NP durch eine Polynomialzeitreduktion auf L reduziert werden kann.

3.3.2 Nachweis

Der Nachweis der ersten Eigenschaft für ein Problem ist in aller Regel leicht. Man "rät" eine Lösung und zeigt, dass man in Polynomialzeit verifizieren kann, ob die Lösung wirklich zutrifft. Im Raten der (korrekten) Lösung findet sich der Nichtdeterminismus wieder.

Der Nachweis der zweiten Eigenschaft, die man für sich allein mit NP-Schwer (oder oft eigentlich falsch aus dem englischen zurückübersetzt mit NP-Hart) bezeichnet, ist schwieriger. Insbesondere bereitet es Probleme, eine Aussage für beliebige Probleme in NP zu zeigen. Daher nimmt man gewöhnlich ein ähnliches Problem, für das die NP-Vollständigkeit schon bekannt ist und reduziert es auf dasjenige Problem, für das man die Eigenschaft der NP-Schwere zeigen will. Aus der Transitivität von Polynomialzeitreduktionen folgt dann, dass alle Probleme aus NP auch auf das betrachtete Problem reduzierbar sind.

Die obige Definition erfordert streng genommen einen Existenzbeweis. Es ist nicht sofort ersichtlich, dass derartige Probleme überhaupt existieren. Es lässt sich aber leicht ein solches Problem konstruieren. Allerdings ist ein derart konstruiertes Problem kaum praxisrelevant. Cook konnte jedoch zeigen, dass das Erfüllbarkeitsproblem der Aussagenlogik NP - vollständig ist und hat damit für ein praxisrelevantes Problem den Nachweis geführt. Dieser Beweis konnte im Gegensatz zu anderen Problemen natürlich noch nicht wie oben dargestellt über die Transitivität von Polynomialzeitreduktionen geführt werden und musste direkt erfolgen.

3.3.3 Karps 21 NP - vollständige Probleme

Eines der bedeutendsten Resultate der Komplexitätstheorie ist der von Stephen Cook im Jahr 1971 erbrachte Nachweis, dass das Erfüllbarkeitsproblem der Aussagenlogik (meist nur kurz SAT genannt) NP - vollständig ist.

1972 griff Richard Karp diese Idee auf und zeigte für 21 verschiedene kombinatorische und graphentheoretische Probleme, die dafür bekannt waren, dass sie sich hartnäckig einer effizienten algorithmischen Lösbarkeit entzogen, dass diese ebenfalls NP - vollständig sind.

Indem er aufzeigte, dass eine große Anzahl von bedeutenden Problemen NP - vollständig sind, motivierte Karp die weitere Erforschung der Klasse NP, der Theorie der NP-Vollständigkeit sowie der Fragestellung, ob die Klassen P und NP identisch sind oder sich unterscheiden (P/NP-Problem). Letzteres zählt heute zu den wichtigsten offenen mathematischen Fragestellungen. Unter anderem zählt es zu den sieben Millennium-Problemen des Clay Mathematics Institute, für deren Lösung Preisgelder von jeweils einer Million US-Dollar ausgelobt wurden.

Der folgende Baum zeigt Karps 21 Probleme, inklusive der zugehörigen Reduktion, die er nutzte, um ihre NP-Vollständigkeit nachzuweisen. Zum Beispiel wurde die NP-Vollständigkeit des Rucksackproblems gezeigt, indem das Problem der exakten Überdeckung darauf reduziert wurde.

1. SATISFIABILITY: das Erfüllbarkeitsproblem der Aussagenlogik für Formeln in Konjunktiver Normalform
   (a) CLIQUE: Cliquenproblem
      - SET PACKING: Mengenpackungsproblem
      - VERTEX COVER: Knotenüberdeckungsproblem
        - SET COVERING: Mengenüberdeckungsproblem
        - FEEDBACK ARC SET: feedback arc set
        - FEEDBACK NODE SET: feedback vertex set
        - (UN-)DIRECTED HAMILTONIAN CIRCUIT: siehe Hamiltonkreisproblem
   (b) 0-1 INTEGER PROGRAMMING: siehe Integer Linear Programming
   (c) 3-SAT: siehe 3-SAT
      - CHROMATIC NUMBER: graph coloring problem
        - CLIQUE COVER: Covering by cliques
        - EXACT COVER: Problem der exakten Überdeckung
          * 3-dimensional MATCHING: 3-dimensional matching
          * STEINER TREE: Steinerbaumproblem
          * HITTING SET: Hitting set
          * KNAPSACK: Rucksackproblem
            · JOB SEQUENCING: Job sequencing
            · PARTITION: Partitionsproblem
              MAX-CUT: Max cut

4 NP-Probleme

4.1 Dijkstra - Algorithmus

Der Algorithmus von Dijkstra (nach seinem Erfinder Edsger W. Dijkstra) dient der Berechnung eines kürzesten Pfades zwischen einem Startknoten und einem beliebigen Knoten in einem kantengewichteten Graphen. Die Gewichte dürfen dabei nicht negativ sein. Für Graphen mit negativen Gewichten aber ohne negative Zyklen ist der Bellman-Ford-Algorithmus geeignet.

Für nicht zusammenhängende, ungerichtete Graphen kann der Abstand zu bestimmten Knoten auch unendlich sein, wenn ein Pfad zwischen Startknoten und diesen Knoten nicht existiert. Dasselbe gilt auch für gerichtete, nicht stark zusammenhängende Graphen.

4.1.1 Algorithmus

G bezeichnet den gewichteten Graphen mit V (engl. vertex) als Knotenmenge, E (engl. edge) als Kantenmenge und w als Gewichtsfunktion, welche Kanten auf positive reelle Zahlen abbildet. Der Knoten s ist der Startknoten, Q ist die Prioritätswarteschlange der noch zu bearbeitenden Knoten und g ist ggf. ein spezieller Zielknoten, bei dem abgebrochen werden kann, wenn seine Distanz zum Startknoten bekannt ist.

Nach Ende des Algorithmus enthält d[v] die Abstände aller Knoten v zu s. In P[v] ist ein spannender Baum der von s aus ausgehenden minimalen Wege in Form eines In-Tree gespeichert.

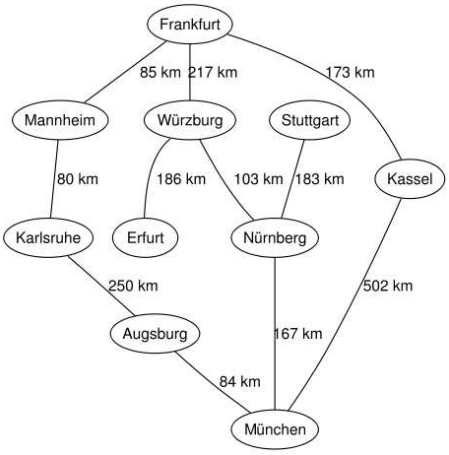
Wird bei Erreichen von g abgebrochen, so enthalten d[v] und P[v] diese Werte nur für alle zuvor betrachteten Knoten v. Das sind mindestens die, die kleineren Abstand als g zu s besitzen.

```
/*
Algorithmus zur Rekonstruktion des kürzesten Pfades
---------------------------------------------------
p: berechneter kürzester Pfad
n: Zielknoten
*/
reconstructShortestPath (n, p){
        // Prüfe ob schon wieder beim
        // Startknoten angekommen
    while (P(n) != NIL)){
                push(n, p);
                // Füge Knoten dem Pfad hinzu
                n := P(n);
                // Mache dasselbe für den Vorgängerknoten
        }
        return p; // Liefere den gefundenen Pfad zurück
}

/*
Algorithmus zum Initialisieren des Graphen
------------------------------------------
G: Zu untersuchender Graph
s: Startknoten
*/
initialize(G, s){
        // Setze die Entfernung zu allen Knoten
        // auf unendlich
        forall v do
                d[v] := "infinity";
                d[s] := 0;
}

/*
```

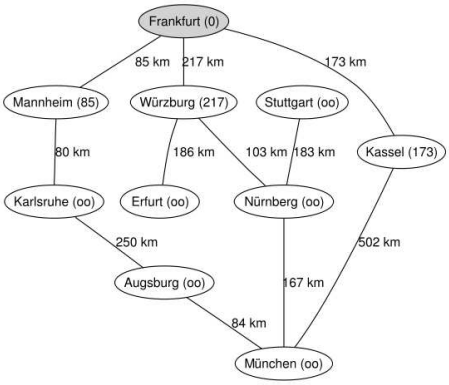
```
    Algorithmus zum Relaxieren einer Kante
    --------------------------------------
    u: Knoten, der gerade expandiert wurde
    v: Aktuell betrachteter Nachfolger von u
    w: Gewichtsfunktion zum Berechnen der Kosten für eine Kante
    d: Funktion zum Berechnen des Abstandes eines Knotens
       zum Startknoten
    */
    relax(u,v,w){
            // Prüfe, ob der Weg über u zu v kürzer ist
            // als der aktuelle
            if d[v] > d[u] + w(u,v) {
                    // Neuer Weg kürzer als bisher gefundener Weg
                    // Aktualisiere geschätzte restliche Weglänge
                    d[v] := d[u] + w(u,v);
                    // Aktualisiere Vorgänger in kürzestem Pfad
                    P[v] := u;
            }
            else
    ; // Ändere nichts, da bereits besserer Pfad bekannt
    }

    /*
    Der eigentliche Algorithmus von Dijkstra
    ----------------------------------------
    G: Zu untersuchender Graph
    s: Startknoten
    g: Der Zielknoten
    w: Gewichtsfunktion zum Berechnen der Kosten für eine Kante
    Q: Prioritätswarteschlange der Knoten, aufsteigend nach
       d-Werten sortiert
    */
    Dijkstra (s, g, w, G) {
(01)        initialize(G, s);
(02)        Q := V[G];
            // Füge alle Knoten aus dem Graph in die Warteschlage ein
            while not isEmpty(Q) {
(05)                // Betrachte Knoten mit geringsten
                    // Abstand zum Startknoten
                    u := pop(Q);
                    if (u == g) then
                    return reconstructShortestPath(g);
(10)                else {
                            // Betrachte alle vom aktuellen Knoten u aus
                           // erreichbaren Knoten (Nachfolger)
                            forall v := successors(u) do {
                                  // relaxiere die Kante zwischen u
(15)                              // und seinem Nachfolger
                                  relax(u, v, w);
                            }
                    }
            }
            // Es konnte kein Pfad gefunden werden
            return fail;
    }
```

4.1.2 Beispiel

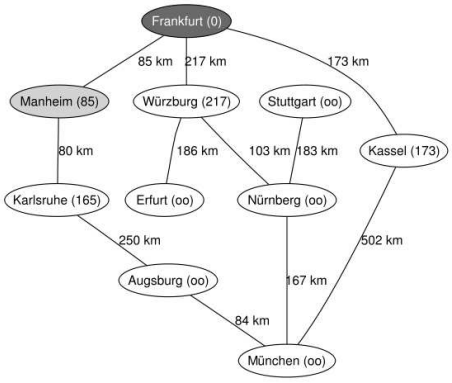
Ein Beispiel für die Anwendung des Algorithmus von Dijkstra ist die Suche nach einem kürzesten Pfad auf einer Landkarte. Im hier verwendeten Beispiel will man in der rechts gezeigten Landkarte von Deutschland einen kürzesten Pfad von Frankfurt nach München finden. Die Zahlen auf den Verbindungen zwischen zwei Städten geben jeweils die Entfernung zwischen den beiden durch die Kante verbundenen Städten an.



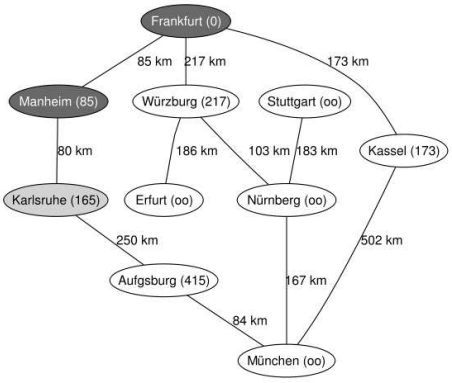
Anfang



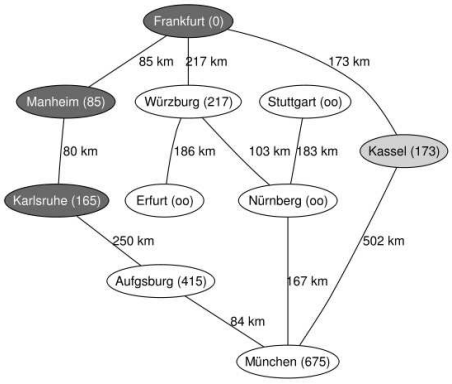
Schritt 1



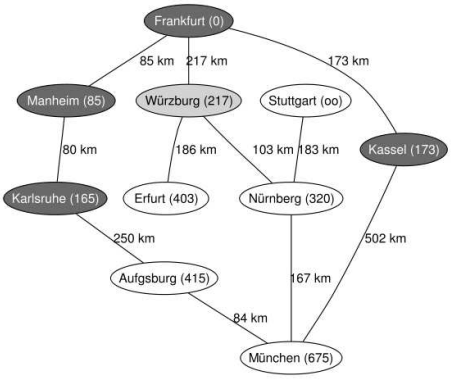
Schritt 2



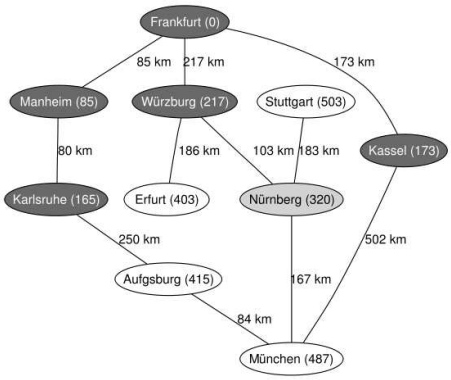
Schritt 3



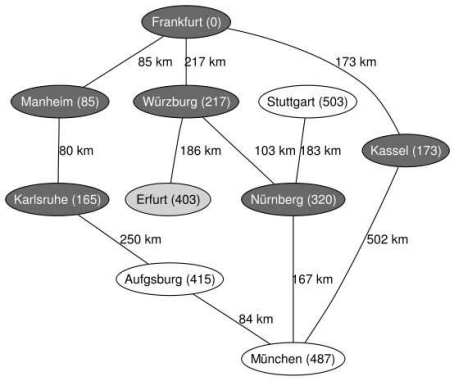
Schritt 4



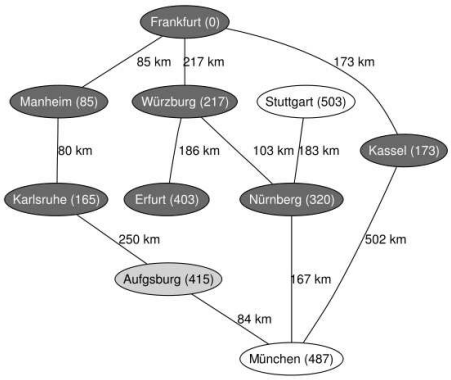
Schritt 5



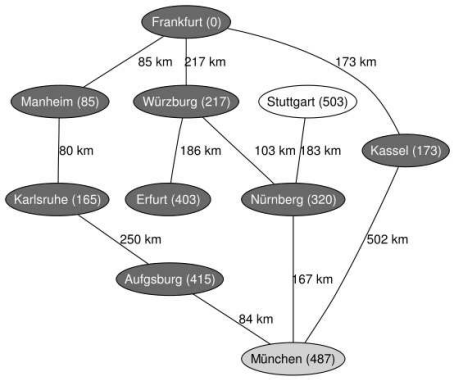
Schritt 6



Schritt 7



Schritt 8



Schritt 9

Der Algorithmus gehört zur Klasse der Greedy-Algorithmen. Sukzessive wird der nächstbeste Knoten, der einen kürzesten Pfad besitzt (Zeile 05), aus der Menge der noch zu bearbeitenden Knoten entfernt. Damit findet sich eine Verwandtschaft zur Breitensuche, die ebenfalls solch ein gieriges Verhalten aufweist.

Ein alternativer Algorithmus zur Suche kürzester Pfade, der sich dagegen auf das Optimalitätsprinzip von Bellman stützt, ist der Floyd-Warshall-Algorithmus. Das Optimalitätsprinzip besagt, dass, wenn der kürzeste Pfad von A nach C über B führt, der Teilpfad A B auch der kürzeste Pfad von A nach B sein muss.

Ein weiterer alternativer Algorithmus ist der A*-Algorithmus, der den Algorithmus von Dijkstra um eine Abschätzfunktion erweitert. Falls diese gewisse Eigenschaften erfüllt, kann damit der kürzeste Pfad unter Umständen schneller gefunden werden.

4.1.3 Zeitkomplexität

Im Folgenden sei m die Anzahl der Kanten und n die Anzahl der Knoten.

Die Zeitkomplexität des Algorithmus hängt in hohem Maße von der Datenstruktur ab, welche zur Speicherung der noch nicht besuchten Knoten (Q) benutzt wird. Im Normalfall wird man hier auf eine Vorrangwarteschlange zurückgreifen, indem man dort die Knoten als Elemente mit ihrer jeweiligen bisherigen Distanz als Schlüssel/Priorität verwendet.

Betrachtet man den Algorithmus unter diesem Aspekt ergibt sich folgender Aufwand:

Das "Füllen" von Q in Zeile 02 wird dadurch erreicht, dass man jeden Knoten sowie seine Priorität (= Distanz) mittels insert in die Warteschlange einfügt; das wird, da es insg. n Knoten gibt, also n Mal durchgeführt.

Da in Zeile 05 jeweils ein Knoten aus Q (bzw. der Warteschlange) entfernt wird, folgt, dass die Schleife in Zeile 03 ebenfalls n Mal durchlaufen wird (ein evtl. früherer Abbruch wegen Erreichen von g sei hier ausgenommen). In jedem Schleifendurchlauf muss geprüft werden, ob die Warteschlange leer ist (empty, Zeile 04) und es muss das nächste Element mit der niedrigsten Priorität entfernt werden (extractMin, Zeile 05); die beiden Operationen werden also jeweils n Mal durchgeführt.

Die Anzahl Aufrufe der relax Funktion kann zwar für jeden Durchlauf der Schleife in Zeile 09 variieren (da die Anzahl der von jedem Knoten u ausgehenden Kanten natürlich unterschiedlich sein kann), insg. über die Laufzeit des gesamten Algorithmus gesehen wird die Schleife aber m Mal ausgeführt, da jeder Knoten nur einmal betrachtet wird, und somit jede von ihm ausgehende Kante ebenfalls nur einmal betrachtet werden kann. Es werden somit alle ausgehenden Kanten aller Knoten im Graphen überprüft und damit erhält man natürlich alle Kanten des Graphen, also m Stück).

Sollte sich hier die bisher berechnete Distanz des Knotens v verringern, muss natürlich auch die Priorität in der Warteschlange mittels decreaseKey entsprechend verringert werden. Das passiert im Worst Case (im - unwahrscheinlichen - Fall dass die Überprüfung jeder Kante einen kürzeren Weg liefert) höchstens m Mal.

Der Vollständigkeit halber sollte man außerdem auch noch das Setzen von $d[v]$ und $P[v]$ in der relax Funktion betrachten, jedoch lässt sich dies jeweils in Konstanter Zeit mit der Komplexität realisieren.

Insgesamt ergibt sich also eine Komplexität von $O(n \cdot (1 + T_{insert} + T_{empty} + T_{extractMin}) + m \cdot (1 + T_{decreaseKey}))$.

Würde man zur Verwaltung der Knoten nun z.B. einfach eine Liste verwenden, ergäbe das eine Laufzeit von $O(n^2 + m)$; insert, empty und decreaseKey lassen sich alle in $O(1)$ realisieren, aber das Suchen des Elements mit der kleinsten Priorität erfordert eine lineare Suche mit $O(n)$.

Besser fährt man hier mit der Verwendung der Datenstruktur Fibonacci-Heap. Diese ermöglicht es ebenfalls, die Operationen insert, empty und decreaseKey (amortisiert betrachtet) in $O(1)$ zu realisieren, die Komplexität von extractMin ist hier aber nur $O(\log n)$. Die gesamte Laufzeit beträgt dann lediglich $O(n \cdot \log n + m)$.

4.2 Hamilton Kreis

Das Hamiltonkreisproblem ist eines der fundamentalen Problemstellungen der Graphentheorie. Es fragt, ob in einem gegebenen Graph ein sogenannter Hamiltonkreis existiert. Ein Hamiltonkreis ist dabei ein Kreis, der alle Knoten des Graphen enthält.

Man unterscheidet zwei grundlegende Varianten des Problems. Beim Gerichteten Hamiltonkreisproblem fragt man nach der Existenz eines gerichteten Hamiltonkreises in einem gerichteten Graphen. Entsprechend fragt man beim Ungerichteten Hamiltonkreisproblem nach der Existenz eines ungerichteten Hamiltonkreises in einem ungerichteten Graphen.

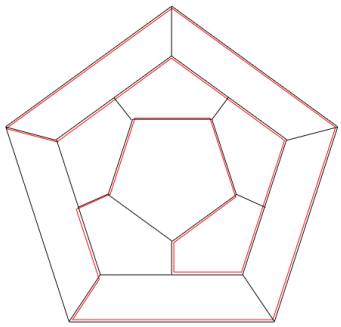


Abbildung 1: Das Dodekaeder, wie alle platonischen Körper, ist hamiltonisch.

Namensgeber des Problems ist der irische Astronom und Mathematiker Sir William Rowan Hamilton, der 1857 das Spiel "The Icosian Game" erfand (und später verbesserte zum "Traveller's Dodecahedron or A Voyage Round The World").

Der "Traveller's Dodecahedron" besteht aus einem hölzernen, regulären Dodekaeder, wobei die 20 Knoten mit Namen bekannter Städte assoziiert sind. Ziel ist es, eine Reiseroute entlang der Kanten des Dodekaeders zu finden, die jede Stadt genau einmal besucht und dort aufhört, wo sie beginnt.

Zunächst erscheint die Aufgabenstellung ähnlich dem 1736 von L. Euler (verneinend) gelösten Königsberger Brückenproblem, einem Spezialfall des Eulerkreisproblems und Grundsteinlegung der Graphentheorie. Während für das Eulerkreisproblem aber besonders effiziente Lösungs-Algorithmen existieren ist bekannt, dass beide Varianten des Hamiltonkreisproblems besonders schwer algorithmisch lösbare Probleme sind. Sowohl die gerichtete als auch die ungerichtete Variante des Hamiltonkreisproblems gehört zur Liste der 21 klassischen NP - vollständigen Probleme, für die Richard Karp 1972 in seinem berühmten Artikel die Zugehörigkeit zu dieser Klasse von Problemen nachgewiesen hat.

4.2.1 Definitionen

Sei $G = (V, E)$ ein Graph mit $|V| = n$ Knoten (oder Ecken) und $|E| = m$ Kanten.

G ist hamiltonisch, wenn er einen Hamiltonkreis zulässt, d.h., wenn es einen Kreis in G gibt, der alle Knoten aus V enthält. Ein Hamiltonpfad ist ein Pfad in G, der alle Knoten aus V enthält. Hat G Hamiltonpfade, jedoch keinen Hamiltonkreis, so ist G semihamiltonisch.

Zur **Potenz eines Graphen**: Für einen Graphen G und $d \in \mathbb{N}$ bezeichnet G^d den Graphen auf V, bei dem zwei Knoten genau dann benachbart sind, wenn sie in G einen Abstand (oder Distanz) $\leq d$ haben. Offenbar gilt $G = G^1 \subseteq G^2 \subseteq G^3 \subseteq ....$

Ist G ein Graph mit n Knoten und Knotengraden $d_1 \leq ... \leq d_n$, so nennt man das Tupel $(d_1, ..., d_n)$ die **Gradsequenz** (oder Valenzfolge) von G, welche eindeutig bestimmt ist. Ein beliebiges Tupel $(a_1, ..., a_n)$ natürlicher Zahlen heißt

hamiltonisch, wenn jeder Graph mit n Knoten und punktweise größerer Gradsequenz hamiltonisch ist. (Eine Gradsequenz $(d_1, ..., d_n)$ ist **punktweise größer** als $(a_1, ..., a_n)$, wenn $d_i \geq a_i$ gilt für alle i.)

4.3 Erfüllbarkeitsproblem

4.3.1 Satisfiability (SAT)

Das Erfüllbarkeitsproblem der Aussagenlogik (oft mit SAT vom Englischen satisfiability notiert) ist ein Entscheidungsproblem der Aussagenlogik. Es fragt, ob eine aussagenlogische Formel erfüllbar ist.

Anwendungen davon finden sich unter anderem in der Komplexitätstheorie, Verifikation und im Entwurf von logischen Schaltungen.

Häufig, insbesondere in der Komplexitätstheorie, wird mit SAT auch nur der Spezialfall von Formeln bezeichnet, die in konjunktiver Normalform vorliegen.

Das Erfüllbarkeitsproblem der Aussagenlogik ist NP - vollständig. Stephen A. Cook und Lenonid Levin zeigten (unabhängig voneinander) diese Eigenschaft Anfang der 1970er Jahre. Ein Problem einer Komplexitätsklasse ist vollständig, wenn man jedes andere Problem dieser Klasse durch eine Polynomialzeitreduktion auf das vollständige Ursprungsproblem (oftmals SAT) übersetzen kann. Cook zeigte hierfür einen Algorithmus auf, mit dem die Berechnungsschritte einer nicht-deterministischen Turingmaschine simuliert werden können.

Richard Karp zeigte 1972 die NP-Vollständigkeit weiterer Probleme. Er prägte damit das heutige Verständnis von NP-Vollständigkeit.

SAT kann auf viele Weisen durch Einschränkungen und Verallgemeinerungen variiert werden. Das Problem 3-SAT ist eine Variante, die ebenfalls NP - vollständig ist, da sich SAT in polynomieller Zeit auf 3-SAT reduzieren lässt. Somit ist auch 3-SAT NP - vollständig.

In der Informatik wird SAT intensiv untersucht. Es gibt viele Arbeitsgruppen, die sich mit der Erforschung der Struktur dieses Problems beschäftigen und neue Verfahren für sogenannte SAT-Solver entwickeln. Dies sind Algorithmen bzw. Programme, die möglichst schnell entscheiden sollen, ob eine aussagenlogische Formel erfüllbar ist oder nicht.

4.3.2 3-SAT

3-SAT ist eine Variante des Erfüllbarkeitsproblems der Aussagenlogik.

Es beschäftigt sich mit der Frage, ob eine in konjunktiver Normalform vorliegende aussagenlogische Formel F, die höchstens 3 Literale pro Klausel enthält, erfüllbar ist. Ein Beispiel für eine solche Formel:

$$F = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4) \wedge (x_1 \vee \overline{x_2}) \tag{9}$$

Gesucht ist nun eine Belegung der Variablen x_1 bis x_4 mit 0 oder 1, für die F den Wert 1 (wahr) annimmt. Falls es eine solche Belegung gibt, ist F erfüllbar, sonst nicht. Wie bei allen NP - vollständigen Problemen ist es "einfach", einen Lösungskandidaten auf seine Gültigkeit zu überprüfen, hier also festzustellen, ob eine vorgegebene Belegung der Variablen die Formel erfüllt. Das Auffinden eines gültigen Lösungskandidaten ist jedoch im allgemeinen "schwierig". Anders formuliert: Eine Ja-Antwort auf das gestellte Entscheidungsproblem ist durch

die Angabe einer Lösung "einfach" zu belegen, eine Nein-Antwort ist dagegen nur "schwierig" zu beweisen.

Das allgemeine Erfüllbarkeitsproblem der Aussagenlogik (SAT) lässt sich auf 3-SAT polynomiell reduzieren, und somit ist 3-SAT nach dem Satz von Cook NP - vollständig.

3-SAT lässt sich wiederum u.a. auf das Cliquenproblem, das Rucksackproblem und auf den gerichteten Hamiltonkreis (DHC) polynomial reduzieren, wodurch auch diese Probleme als NP-schwer nachgewiesen sind.

4.4 Partitionsproblem

Das Partitionsproblem (auch Zahlenaufteilungsproblem, oft mit PARTITION notiert) ist ein Optimierungs- bzw. Entscheidungsproblem der Kombinatorik.

Die Aufgabenstellung beim Partitionsproblem lautet: Gegeben sei eine Menge von (positiven) Zahlen. Gesucht wird eine Aufteilung dieser Zahlen auf zwei Haufen, so dass die Differenz der Summen der Zahlen in den beiden Haufen möglichst klein ist.

Eine äquivalente Formulierung lautet präziser: Gegeben sei eine Menge A von N positiven Zahlen ai. Gesucht wird eine Untermenge $A_1 \subset A$, so dass

$$E := \left| \sum_{a_i \in A_1} a_i - \sum_{a_i \in A, a_i \notin A_1} a_i \right| \tag{10}$$

minimal wird.

Ist die Summe aller N Zahlen ungerade, so ist die minimale Differenz E_{min} eins, ansonsten null. Eine Aufteilung, für die $E = E_{min}$ ist, heißt perfekte Aufteilung.

Als zusätzliche Bedingung kann man die Lösungsmenge des Partitionsproblems von vornherein einschränken, indem man nur ausgewogene Aufteilungen zulässt, in denen beide Haufen gleich groß sind, das heißt die Anzahl der Zahlen in den Untermengen muss für gerades N gleich sein und muss sich für ungerades N um 1 unterscheiden.

Wandelt man die Fragestellung ab und fragt, "Gibt es eine perfekte Aufteilung?" oder "Existiert eine Aufteilung, für die E maximal ... ist?", so wird das oben beschriebene Optimierungsproblem zu einem Entscheidungsproblem, das heißt, man sucht nicht mehr nach der besten Aufteilung, sondern fragt nur noch nach deren Existenz.

4.4.1 Phasenübergang im Partitionsproblem

Man beobachtet beim Partitionsproblem zwei verschiedene sogenannte Phasen: Besteht die Menge A aus vielen kleinen Zahlen, so ist anschaulich klar, dass es viele perfekte Aufteilungen gibt, und es einfach ist, eine dieser Aufteilungen zu finden (einfache Phase). Besteht A hingegen aus wenigen großen Zahlen, so ist es unwahrscheinlich, dass überhaupt eine perfekte Aufteilung existiert, und man muss alle Möglichkeiten durchprobieren, um die beste Aufteilung zu finden (harte Phase).

Zwischen der einfachen und der harten Phase beobachtet man einen Übergang, den man in Analogie zur statistischen Physik Phasenübergang nennt. An diesem Phasenübergang fällt die Wahrscheinlichkeit, eine perfekte Aufteilung

zu finden, sprunghaft von 1 in der einfachen Phase auf 0 in der harten Phase. Mit wachsender Anzahl N der Zahlen wird der Übergang immer schärfer.

Die Lage des Phasenübergangs in Abhängigkeit von der Anzahl und der Größe der einzelnen Zahlen lässt sich mit Methoden der statistischen Physik berechnen.

4.4.2 Komplexität

Das Partitionsproblem gehört zu den 21 klassischen NP - vollständigen Problemen, von denen Richard Karp 1972 die Zugehörigkeit zur Klasse der NP - vollständigen Probleme zeigen konnte.

Es hat eine "worst-case-Laufzeit", die exponentiell mit der Anzahl der Zahlen N wächst, das heißt im schlimmsten Fall benötigt ein Algorithmus zur Lösung des Entscheidungsproblem eine Rechenzeit, die exponentiell mit N ansteigt. In vielen Fällen liegt die tatsächlich benötigte Rechenzeit jedoch deutlich darunter: In der einfachen Phase stößt der Algorithmus schnell auf eine der vielen perfekten Lösungen und kann das Entscheidungsproblem somit mit "ja, es gibt eine perfekte Lösung" beantworten und die Suche abbrechen. Auch in der harten Phase können geeignete Algorithmen (z.B. der cBLDM-Algorithmus[1]) die Suche schnell mit einer negativen Entscheidung abschließen, falls keine Lösung existiert. Die "schwierigsten Probleme" liegen somit direkt am Phasenübergang, wo erst alle 2^{N-1} Aufteilungen durchprobiert werden müssen, bevor das Problem entschieden werden kann.

4.5 Ganzzahlige lineare Optimierung

Die Ganzzahlige lineare Optimierung (auch ganzzahlige Optimierung) ist ein Teilgebiet der angewandten Mathematik. Wie die Lineare Optimierung beschäftigt sie sich mit der Optimierung linearer Zielfunktionen über einer Menge, die durch lineare Gleichungen und Ungleichungen eingeschränkt ist. Der Unterschied liegt darin, dass in der ganzzahligen Optimierung einige oder alle Variablen nur ganzzahlige Werte annehmen dürfen und nicht beliebige reelle Werte wie in der linearen Optimierung. Die ganzzahlige Optimierung lässt sich geometrisch als Optimierung über einem konvexen Polyeder (einem höherdimensionalen Vieleck) auffassen und ist damit ein Spezialfall der konvexen Optimierung. Im Unterschied zur linearen Programmierung ist allerdings das zugrundeliegende Polyeder meist nicht genau bekannt, was das Problem aus komplexitätstheoretischer Sicht NP-schwer macht.

Da die Menge der zulässigen Lösungen diskret, also nicht kontinuierlich ist, ist auch der Begriff diskrete Optimierung gebräuchlich. Eine weitere häufige Bezeichnung ist ganzzahlige (lineare) Programmierung (von engl. integer (linear) programming), wobei der Begriff Programm im Sinne von Planung zu verstehen ist und nicht im Sinne eines Computerprogramms. Er wurde schon in den 1940er Jahren von George Dantzig geprägt, bevor Computer zur Lösung von Optimierungsproblemen eingesetzt wurden.

Noch stärker als die lineare hat sich die ganzzahlige Optimierung seit ihren Anfängen in den 1950er Jahren zu einem Modellierungs- und Optimierungswerkzeug für viele praktische Probleme entwickelt, für die keine speziellen Algorithmen bekannt sind. Durch bedeutende Fortschritte in der Entwicklung der

Lösungsverfahren in den 1980er und 1990er Jahren hat die ganzzahlige Optimierung heute viele Anwendungen, beispielsweise in der Produktion, in der Planung von Telekommunikations- und Nahverkehrsnetzen und in der Tourenplanung.

Zur Lösung ganzzahliger Optimierungsprobleme gibt es einerseits exakte Lösungsverfahren wie beispielsweise Branch-and-Bound und Schnittebenenverfahren, die auf der Lösung vieler ähnlicher linearer Programme basieren, und andererseits eine Vielzahl von Heuristiken. Trotzdem ist die Lösung ganzzahliger linearer Programme in der Praxis immer noch eine schwere Aufgabe, die je nach Größe und Struktur des zu lösenden Problems eine geschickte Modellierung und mehr oder weniger speziell entwickelte oder angepasste Algorithmen erfordert. Oft werden daher mehrere Lösungsverfahren kombiniert.

4.5.1 Problemdefinition

Ein ganzzahliges Programm (engl. integer program, IP) hat die gleiche Form wie ein lineares Programm (LP), mit dem Unterschied, dass die Variablen ganzzahlig sein müssen:

$$\max\{c^T x \mid Ax \leq b, x \geq 0, x \in \mathbb{Z}^n\} \tag{11}$$

Dabei ist A eine reelle Matrix und b und c sind Vektoren passender Dimension. Die Bedingung $Ax \leq b$ ist komponentenweise zu verstehen, also als

$$a_i \cdot x = \sum_{j=1}^{n} a_{ij} x_j \leq b_i \tag{12}$$

für alle Zeilen i der Matrix A. Genauso bedeutet die Bedingung $x \geq 0$, dass alle Einträge des Vektors x nichtnegativ sein müssen. Gelten die Ganzzahligkeitsbedingungen nur für einen Teil der Variablen, spricht man auch von einem gemischt-ganzzahligen Programm (engl. mixed-integer program, MIP). Auch die Präzisierung ganzzahliges lineares Programm (engl. integer linear program, ILP) ist gebräuchlich. Wie auch in der linearen Optimierung gibt es mehrere äquivalente Formulierungen, die sich ineinander transformieren lassen.

Die ganzzahlige Optimierung lässt sich, wie die lineare Variante, zu einem großen Teil geometrisch interpretieren. Die Menge

$$P := \{x \mid Ax \leq b, x \geq 0\}, \tag{13}$$

die durch Weglassen der Ganzzahligkeitsbedingungen entsteht, bildet ein konvexes Polyeder im n-dimensionalen Raum, dessen beschränkende Hyperebenen den Zeilen des Ungleichungssystems entsprechen. P enthält u. a. alle zulässigen Punkte des Ausgangssystems, also alle ganzzahligen Punkte, die die Bedingungen $Ax \leq b$ erfüllen, aber im Unterschied zur linearen Optimierung sind nicht alle Punkte in P zulässig. Das lineare Programm

$$\max\{c^T x \mid x \in P\} \tag{14}$$

wird als LP-Relaxierung des ganzzahligen Problems bezeichnet und spielt eine bedeutende Rolle für einige Lösungsverfahren (siehe unten).

Wie lineare Programme können auch ganzzahlige Programme unlösbar oder unbeschränkt sein. In allen anderen Fällen gibt es mindestens eine Optimallösung. Im Unterschied zu LPs ist allerdings die Menge der Optimallösungen

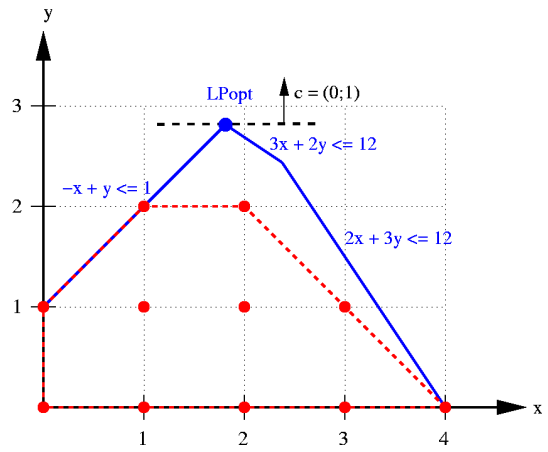


Abbildung 2: Polytop der zulässigen ganzzahligen Punkte (rot) mit LP-Relaxierung (blau)

eines IPs keine Seitenfläche des Polyeders P, so dass es neben genau einer oder unendlich vielen optimalen Lösungen auch andere endliche Anzahl (größer als 1) davon geben kann.

4.5.2 Beispiel

Im Bild 2 ist das ganzzahlige lineare Programm

$$\begin{aligned} \max \quad & y && && (15)\\ -x \quad & +y && \leq 1 && (16)\\ 3x \quad & +2y && \leq 12 && (17)\\ 2x \quad & +3y && \leq 12 && (18)\\ x, y \in Z_+ & && && (19) \end{aligned}$$

dargestellt. Die zulässigen ganzzahligen Punkte sind rot eingezeichnet, und die rot gestrichelten Linien kennzeichnen ihre konvexe Hülle, also das kleinste Polyeder, das alle diese Punkte enthält. Über diesem Polyeder soll eigentlich optimiert werden, aber es ist meist nicht genau bekannt. Die blauen Linien zusammen mit den Koordinatenachsen begrenzen das Polyeder P der LP-Relaxierung, das durch das Ungleichungssystem ohne Ganzzahligkeitsbedigungen gegeben ist. Ziel der Optimierung ist es, die schwarz gestrichelte Linie so weit parallel nach oben (in Richtung des Vektors c = (0;1)) zu verschieben, dass sie das jeweilige Polyeder gerade noch berührt. Die Optimallösungen des ganzzahligen Problems sind also die Punkte (1;2) und (2;2) mit dem Zielfunktionswert $c^T x = (0;1)^T(1;2) = 2$. Die - in diesem Fall eindeutige - optimale Lösung der LP-Relaxierung mit dem Zielfunktionswert 2,8 ist der blau markierte Punkt $LP_{opt} = (1,8;2,8)$, der nicht ganzzahlig und damit auch nicht zulässig für das IP ist.

4.5.3 Komplexität

Im Unterschied zu linearen Programmen, die beispielsweise mit Innere-Punkte-Verfahren in Polynomialzeit optimal gelöst werden können, ist das Finden einer beweisbaren Optimallösung für ganzzahlige Programme ein NP-schweres Problem. Dies macht sich auch in der Praxis bemerkbar. Während selbst große lineare Programme heute weitgehend mit Standardmethoden gelöst werden können, hängt die Lösbarkeit ganzzahliger Programme sehr viel stärker von den spezifischen Eigenheiten des jeweiligen Planungsproblems und von der gewählten Modellierung ab. Ein Optimierungsproblem mit hundert ganzzahligen Variablen kann aus praktischer Sicht unlösbar sein, während andere Probleme mit tausenden ganzzahliger Variablen innerhalb weniger Sekunden gelöst werden. Es gibt zwar auch in der ganzzahligen Optimierung Standardmethoden, mit denen durch große algorithmische Fortschritte innerhalb der letzten zehn Jahre mittlerweile viele praktische Planungsprobleme als IP gelöst werden können, aber gerade die Lösung großer ganzzahliger Programme in annehmbarer Zeit erfordert oft eine geschickte Modellierung und eine Kombination mehrerer Lösungsverfahren mit problemspezifischen Anpassungen.

4.6 Faktorisierungsverfahren

Das Faktorisierungsproblem für ganze Zahlen ist eine Aufgabenstellung aus dem mathematischen Teilgebiet der Zahlentheorie. Dabei soll zu einer zusammengesetzten Zahl ein nichttrivialer Teiler ermittelt werden. Ist beispielsweise die Zahl 91 gegeben, so sucht man eine Zahl wie 7, die 91 teilt. Entsprechende Algorithmen, die dies bewerkstelligen, bezeichnet man als Faktorisierungsverfahren. Durch rekursive Anwendung von Faktorisierungsverfahren in Kombination mit Primzahltests, kann die Primfaktorzerlegung einer ganzen Zahl berechnet werden.

Bis heute ist kein Faktorisierungsverfahren bekannt, das nichttriviale Teiler und damit die Primfaktorzerlegung einer Zahl effizient berechnet. Das bedeutet, dass ein enormer Rechenaufwand notwendig ist, um eine Zahl mit mehreren hundert Stellen zu faktorisieren. Dies wird in der Kryptografie ausgenutzt, wo es zum Entschlüsseln von Nachrichten notwendig ist solche Zahlen zu faktorisieren. Dabei kommt zusätzlich zum Tragen, dass man mittels eines Primzahltests ziemlich einfach feststellen kann, ob eine Zahl eine Primzahl ist.

In der theoretischen Informatik werden Probleme in Komplexitätsklassen eingeteilt, die darüber Aufschluss geben, welchen Aufwand die Lösung eines Problems erfordert. Beim Faktorisierungsproblem für ganze Zahlen ist nicht bekannt welcher Komplexitätsklasse es angehört. Das heißt, dass es zumindest theoretisch möglich ist, dass irgendwann ein Algorithmus entdeckt wird, der ganze Zahlen mit überschaubaren Aufwand faktorisieren kann.

Die besten bekannten Algorithmen sind das 1981 von Carl Pomerance erfundene quadratische Sieb, das um 1990 von mehreren Mathematikern (u.a. Pollard, A. Lenstra, H.W. Lenstra Jr., Manasse, Pomerance) gemeinsam entwickelte Zahlkörpersieb und die Methode der Elliptischen Kurven, die 1987 von Hendrik W. Lenstra, Jr. vorgestellt wurde.

Den aktuellen Forschungsstand auf dem Gebiet der Faktorisierungsverfahren zu verfolgen, ist Ziel der RSA Factoring Challenge. Daraus ergeben sich Anhaltspunkte für die notwendige Größe der im RSA-Kryptosystem verwand-

ten Semiprimzahlen.

In der Praxis wird man, um eine Zahl zu faktorisieren, wie folgt vorgehen:

1. Durch Probedivision kleine Faktoren finden/entfernen
2. Mit Hilfe eines Primzahltests herausfinden, ob die Zahl eine Primzahl oder eine Primpotenz ist
3. Mit der Methode der Elliptischen Kurven nach vergleichsweise kleinen Primfaktoren ($< 10^{30}$) suchen
4. Mit dem Quadratischen Sieb (für Zahlen mit weniger als 120 Dezimalstellen) oder dem Zahlkörpersieb faktorisieren

4.6.1 Überblick der Faktorisierungsverfahren

Im Folgenden bezeichnet n immer eine zusammengesetzte Zahl, für die ein Teiler ermittelt werden soll.

Probedivision: Das einfachste Verfahren zur Ermittlung eines Teilers von n ist die Probedivision. Dabei wird n durch alle Primzahlen beginnend mit der Zwei dividiert bis sich eine Primzahl als deren Teiler erweist oder bis der Probedivisor größer als $\sqrt{n}$ ist. Das Verfahren ist zwar sehr aufwändig, eignet sich allerdings sehr gut zur Bestimmung kleiner Primfaktoren.

Berechnung des größten gemeinsamen Teilers: Die Probedivision kann durch den Euklidischen Algorithmus oder andere Verfahren zur Bestimmung des größten gemeinsamen Teilers so erweitert werden, dass man alle Primfaktoren von n aus einem bestimmten Intervall findet. Dazu verwendet man das Produkt m aller Primzahlen des Intervalls und berechnet den größten gemeinsamen Teiler der beiden Zahlen m und n. Dieser ist das Produkt aller Primfaktoren, die aus dem gewählten Intervall stammen und man kann aus ihm die einzelnen Primfaktoren zurückgewinnen. Der Vorteil dieses Verfahrens liegt darin, dass man die Probedivision dann nur noch auf den Quotienten n:m anwenden muss, der viel kleiner als n ist.

Faktorisierungsmethode von Fermat: Ein Verfahren, das sich besonders gut eignet um Teiler in der Nähe von $\sqrt{n}$ zu finden, ist die Faktorisierungsmethode von Fermat. Dieser Algorithmus funktioniert nur für ungerade n und nutzt aus, dass sich diese als Differenz zweier Quadratzahlen darstellen lassen. Er berechnet zuerst die kleinste ganze Zahl x, die größer oder gleich $\sqrt{n}$ ist. Anschließend berechnet der Algorithmus die Differenzen $x^2-n, (x+1)^2-n, (x+2)^2-n, \ldots$ bis eine dieser Differenzen eine Quadratzahl ist. Aus dieser werden Teiler von n berechnet.

Weitere Verfahren:

- Faktorisierungsmethode von Lehman
- Pollard-Rho-Methode
- Pollard-p-1-Methode
- Pollard-p+1-Methode
- Methode der elliptischen Kurven

- Kettenbruchmethode
- Methode der Klassengruppen
- Quadratisches Sieb
- Zahlkörpersieb

Shor-Algorithmus: Eine besondere Stellung unter den Faktorisierungsverfahren nimmt der Shor-Algorithmus ein. Er kann nicht auf klassischen Rechnern ausgeführt werden, sondern benötigt einen Quantencomputer. Auf diesem kann er jedoch in Polynomialzeit einen Faktor von n berechnen. Allerdings können noch keine Quantencomputer gebaut werden, die über eine für die Faktorisierung großer Zahlen ausreichende Registergröße verfügen. Die Funktion des Shor-Algorithmus beruht darauf, die Ordnung eines Elements der primen Restklassengruppe $(\mathbb{Z}/n\mathbb{Z})^\times$ mit Hilfe der Quantenfouriertransformation zu bestimmen.

4.6.2 Zahlkörpersieb

Das Zahlkörpersieb ist ein Begriff aus dem mathematischen Teilgebiet der Zahlentheorie. Es ist einer der schnellsten bekannten Algorithmen zur Faktorisierung großer Zahlen.

Das Zahlkörpersieb wird vor allem für Zahlen mit über 100 Stellen benutzt, die durch andere Verfahren nicht zerlegt werden konnten. Typischerweise werden dabei mehrere 100 Rechner parallel betrieben.

Geschichte Am 31. August 1988 schrieb John M. Pollard einen Brief an A. M. Odlyzko mit Kopien an R. P. Brent, J. Brillhart, H. W. Lenstra, C. P. Schnorr und H. Suyama, worin er ein neues Faktorisierungsverfahren für ganz spezielle Zahlen beschrieb. In diesem Brief illustrierte er dieses Verfahren an der Fermat-Zahl F_7 und vermutete, dass damit die bis dato noch nicht faktorisierte Zahl F_9 möglicherweise ein Kandidat für dieses Verfahren ist. Pollard benutzte aber noch kein Siebverfahren im algebraischen Zahlkörper.

In den Folgejahren wurde diese Idee u.a. von A. K. Lenstra, H. W. Lenstra, M. S. Manasse und J. M. Pollard ausgebaut. Daraus entstand das spezielle Zahlkörpersieb (wie das Verfahren heutzutage bezeichnet wird, um es vom allgemeinen Zahlkörpersieb unterscheiden zu können). Das spezielle Zahlkörpersieb lässt sich nur für Zahlen der Form $b^m - r$ mit b, r klein und m groß anwenden.

Das allgemeine Zahlkörpersieb wurde annähernd zeitgleich zum speziellen Zahlkörpersieb von J. P. Buhler, H. W. Lenstra und Carl Pomerance gefunden. Dieses ist für beliebige Zahlen anwendbar, dafür muss man aber Einbußen bei der Geschwindigkeit hinnehmen.

1992 gelang mit Hilfe des speziellen Zahlkörpersiebs die Faktorisierung von F_9.

Bereits 1991 publizierte J. M. Pollard eine Variante des Zahlkörpersiebs, bei der ein zweidimensionales Sieb benutzt wird, welches er als Gittersieb bezeichnet.

Mit dieser Gittersiebvariante, kombiniert mit anderen Methoden, wurde von 2003 bis 2005 die bislang größte aus zwei großen Primfaktoren zusammengesetzte Zahl ohne spezielle Struktur faktorisiert. Dabei handelt es sich um RSA-200, eine 200-stellige Dezimalzahl.

Asymptotische Laufzeit Die asymptotische Laufzeit des Zahlkörpersiebs konnte bislang nicht exakt bewiesen werden. Unter einigen, als wahrscheinlich geltenden Annahmen, kann man diese jedoch zu

$$e^{(C+o(1))(n)^{\frac{1}{3}}(\log n)^{\frac{2}{3}}} \tag{20}$$

berechnen. N bezeichnet hierbei die Länge der Zahl. Dabei ist die Konstante C davon abhängig, ob das spezielle oder das allgemeine Zahlkörpersieb benutzt wird:

- Spezielles Zahlkörpersieb: $C = (32/9)^{1/3} = 1,526285...$
- Allgemeines Zahlkörpersieb: $C = (64/9)^{1/3} = 1,922999...$

Teil II
Sortieren

5 Grundlagen

Ein Sortierverfahren ist ein Algorithmus, der dazu die dient, eine Liste von Elementen zu sortieren. Voraussetzung ist, dass auf der Menge der Elemente eine strenge schwache Ordnung definiert ist, z.B. die lexikographische Ordnung von Zeichenketten oder die numerische Ordnung von Zahlen.

Es gibt verschiedene Sortierverfahren, die unterschiedlich effizient arbeiten. Die Komplexität eines Algorithmus, also die Anzahl der nötigen Operationen, wird üblicherweise in der Landau-Notation dargestellt. Einige Sortierverfahren benötigen außerdem neben dem zur Speicherung des Arrays nötigen noch weiteren Speicherplatz. Komplexität und Speicherbedarf hängen bei einigen Sortierverfahren von der anfänglichen Anordnung der Werte im Array ab, man unterscheidet dann zwischen Best Case (bester Fall), Average Case (Durchschnittsverhalten) und Worst Case (schlechtester Fall).

Man unterscheidet zudem zwischen stabilen und instabilen Sortierverfahren. Stabile Sortierverfahren sind solche, die die relative Reihenfolge von Elementen, die bezüglich der Ordnung äquivalent sind, nicht verändern, während instabile Sortierverfahren dies nicht garantieren.

Zudem unterscheidet man zwischen Sortierverfahren, die in-place (auch in situ) arbeiten, d.h. die ohne zusätzlichen Speicherplatz funktionieren, und solchen, die dies nicht tun.

Allgemeine Vorraussetzungen:

- Feld mit Elementen x_0 bis x_{n-1} (n Elemente)
- Gefüllt mit beliebigen Werten (z.B. Zufallszahlen)
- Nicht zwingend die Zahlen 1 bis n
- Elemente dürfen doppelt vorkommen

6 Sortierverfahren

6.1 Bubblesort

Der Algorithmus vergleicht der Reihe nach zwei benachbarte Elemente und vertauscht sie, falls sie in der falschen Reihenfolge vorliegen. Dieser Vorgang wird solange wiederholt, bis keine Vertauschungen mehr nötig sind. Hierzu sind in der Regel mehrere Durchläufe erforderlich.

Je nachdem, ob auf- oder absteigend sortiert wird, steigen die größeren oder kleineren Elemente wie Blasen im Wasser (daher der Name) immer weiter nach oben, d.h. an das Ende der Reihe. Auch werden immer zwei Zahlen miteinander in “Bubbles” vertauscht.

C-Beispielcode:

```
int h;
int v[n];
    for (i=0, i<n, i++)
        v[i]=rand();
for (i=0, i<n-1, i++)
    {
    for j=i+1, j<n, j++)        \\Vergleiche v_i und v_j
        {
        if v[j]<v[i]
            {
            h=v{j};             \\Vertausche v_i und v_j
            v[j]=v[i]
            v[i]=h;
            }
    };
```

Rechenaufwand:

$$r(n) = \frac{(n-1)n}{2} = O(n^2) \tag{21}$$

(für das Vergleichen, nochmal die Hälfte werden statistisch gesehen vertauscht)

Beachte: Unsortierte Elemente sind in Unordnung (im Gegensatz zu z.B. Quicksort)

6.2 Quicksort

Quicksort ist ein schneller, rekursiver, nicht-stabiler Sortieralgorithmus, der nach dem Prinzip Teile und herrsche (engl. Divide and conquer) arbeitet. Er wurde 1962 von *C. Antony R. Hoare* in seiner Grundform entwickelt und seitdem von vielen Forschern verbessert.

Quicksort wählt ein Element aus der zu sortierenden Liste aus (Pivotelement) und zerlegt die Liste in zwei Teillisten, eine untere, die alle Elemente kleiner und eine obere, die alle Elemente gleich oder größer dem Pivotelement enthält. Diese Teillisten werden rekursiv sortiert. Anschließend wird das Ergebnis zusammengesetzt. Es besteht (in der Reihenfolge) aus der unteren Liste, dem Pivotelement und der oberen Liste.

C-Beispielcode:

```
void tausche(int daten[], int a, int b) {
        int temp = daten[a];
        daten[a] = daten[b];
        daten[b] = temp;
}

int teile(int daten[], int links, int rechts) {
        int index = links;
        for (int zeiger = links; zeiger < rechts; zeiger++) {
                if (daten[zeiger] <= daten[rechts]) {
                        tausche(daten, index, zeiger);
                        index++;
                }
        }
        tausche(daten, index, rechts);
        return index;
}

void quicksort(int daten[], int links, int rechts) {
        if (rechts > links) {
                int teiler = teile(daten, links, rechts);
                quicksort(daten, links, teiler - 1);
```

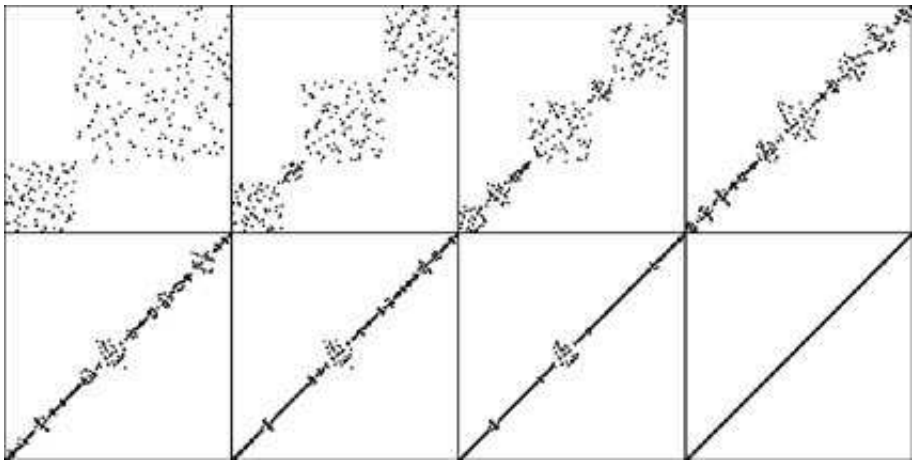


Abbildung 3: Grafische Darstellung des Quicksort-Algorithmus. Gezeigt sind Momentaufnahmen des Zahlenfelds für unterschiedlichen Fortschritt des Sortierverfahrens.

```
                quicksort(daten, teiler + 1, rechts);
        }
}
```

Die **Laufzeit** des Algorithmus hängt im wesentlichen von der Wahl des Pivotelementes ab.

$$r(n) = n - 1 + 2(\frac{n}{2} - 1) + 4(\frac{n}{4} - 1) + ...(bis \frac{n}{2^1} < 1) = \log_2 n * n \qquad (22)$$

Im Worst Case (schlechtesten Fall) wird das Pivotelement stets so gewählt, dass es das größte oder das kleinste Element der Liste ist. Dies ist etwa der Fall, wenn als Pivotelement stets das Element am Ende der Liste gewählt wird und die zu sortierende Liste bereits sortiert vorliegt. Die zu untersuchende Liste wird dann in jeden Rekursionsschritt nur um eins kleiner und die benötigte Rechenzeit liegt in der Komplexitätsklasse $O(n^2)$.

Im Best Case (bester Fall) wird das Pivotelement stets so gewählt, dass die entstehenden Teillisten beide gleich groß sind. In diesem Fall liegt der Algorithmus in $O(n*log(n))$. Auch für den durchschnittlichen Fall lässt sich zeigen, dass Quicksort in $O(n*log(n))$ ist.

Für die Wahl des Pivotelementes sind in der Literatur verschiedene Ansätze beschrieben worden. Die Wahrscheinlichkeit des Eintreffens des Worst Case ist bei diesen unterschiedlich groß, aber immer ungleich Null.

Ein möglicher Ansatz ist es, immer das erste, das letzte oder das mittlere Element der Liste zu wählen. Dieser naive Ansatz ist aber relativ ineffizient. Eine andere Möglichkeit ist es den Median dieser drei Elemente zu bestimmen und als Pivotelement zu verwenden. Ein anderer Ansatz ist, als Pivotelement ein zufälliges Element auszuwählen. Bei diesem randomisierten Quicksort ist die Wahrscheinlichkeit, dass das Pivotelement in jedem Teilungsschritt so gewählt

wird, dass sich die Worst-Case-Laufzeit ergibt, extrem gering. Man kann davon ausgehen, dass er praktisch nie auftritt.

Durch die geschickte Wahl des Pivotelementes kann aber nur das mittlere schlechteste Verhalten verbessert werden. Vollständig ausgeschlossen werden kann das Eintreten des Worst Case jedoch nicht.

Ein Vergleich des Sortierens vom zwei Listen mit 300 Elementen per Bubblesort und Quicksort zeigt die klare Überlegenheit von Quicksort: Es benötigt ca. 2500 Vergleiche, während Bubblesort 45000 benötigt.

6.3 Heapsort

Heapsort ist ein 1964 von Robert W. Floyd und J. W. J. Williams entwickeltes, relativ schnelles Sortierverfahren. Es handelt sich um eine Verbesserung von Selectionsort. Seine Komplexität ist bei einem Array der Länge n in der Landau-Notation ausgedrückt O(n·log(n)). Heapsort arbeitet zwar in-place (auch in situ), ist jedoch nicht stabil.

Die Voraussetzung dafür, dass man ein Array von sortierbaren Werten mit Heapsort sortieren kann, ist, dass dieses einen binären Heap repräsentiert. Ist dies nicht der Fall, so muss man es zuerst in einen Heap überführen.

Man beachte, dass für die zweite Hälfte jedes Arrays die Heap-Eigenschaft bereits erfüllt ist, denn jeder Knoten in der zweiten Hälfte des Arrays entspricht im Heap einem Blatt, hat also keinen Nachfolgeknoten, dessen Wert größer sein könnte als der eigene (im Folgenden wird der Einfachheit halber der Knoten mit dem größten Wert der größte Knoten genannt).

Um ein Array in einen Heap zu überführen, beginnt man deshalb in der Mitte des Arrays. Man versickert nun sukzessive alle davor liegenden Knoten, bis das erste Element versickert wurde. Versickern (auch: absinken) heißt, dass man einen Knoten mit dem größeren seiner beiden Nachfolgeknoten vertauscht, falls dieser größer ist als er selbst, und damit so lange fortfährt, bis er keinen Nachfolgeknoten mehr hat, der größer ist als er selbst.

Um die Rechnung zu vereinfachen, wird im Folgenden vorausgesetzt, dass das erste Element des Arrays den Index 1 hat. Die Nachfolger eines Knotens mit dem Index i liegen dann an den Positionen $2 \cdot i$ und $2 \cdot i + 1$.

Man kann zeigen, dass der Aufbau des Heaps, in Landau-Notation ausgedrückt, in $O(n)$ Schritten ablaufen kann. Das Versickern eines Elements von der Wurzel benötigt im ungünstigsten Fall (Worst Case) $O(\log(n))$ Schritte. Insgesamt garantiert Heapsort eine Laufzeit von $O(n \cdot \log(n))$.

Für genügend große n (>16000) ist ein Heapsort im Durchschnitt schneller als ein optimierter Quicksort. Hinzu kommt das bessere Worst-Case-Verhalten von $O(n \cdot \log(n))$ gegenüber $O(n^2)$ beim Quicksort, so dass bei großen Sortiermengen Heapsort zu bevorzugen ist.

6.3.1 Baum

Ein Baum ist in der Graphentheorie ein spezieller Graph, mit dem sich eine Monohierarchie modellieren lässt. Je nachdem, ob die Kanten des Baums eine ausgezeichnete Richtung besitzen, lassen sich graphentheoretische Bäume unterteilen in ungerichtete Bäume und gewurzelte Bäume, und für gewurzelte Bäume in Out-Trees, bei denen die Kanten von der Wurzel ausgehen, und In-Trees, bei denen Kanten in Richtung Wurzel zeigen.

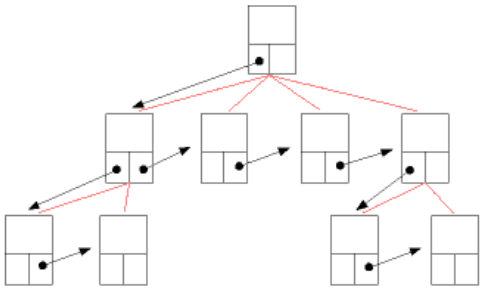


Abbildung 4: Implementierung eines allgemeinen Baumes durch einen Binärbaum

In der Informatik werden Bäume häufig als Datenstruktur eingesetzt. In diesem Fall werden sie aber anders repräsentiert als allgemeine Graphen. Durch Entfernen einer Kante zerfällt ein Baum in zwei Teilbäume und bildet damit einen so genannten Wald.

Es existiert eine Vielzahl von Begriffen, die Bäume näher spezifizieren. So gibt es zum Beispiel:

- Binärbäume
- Binomialbäume,
- balancierte Bäume und
- vollständige Binärbäume.

Gewurzelte Bäume, insbesondere Out-Trees, werden häufig als Datenstruktur verwendet. Bei beschränkter Ordnung können diese so implementiert werden, dass jeder Knoten einen festen Satz an Variablen oder ein Array für die Referenzen auf seine Kinder enthält. Häufig besitzen die Knoten auch eine Referenz auf ihren Elternknoten (back pointer). Ein Baum unbeschränkter Ordnung kann implementiert werden, indem man statt Arrays dynamische Listen verwendet. In Programmiersprachen ohne dynamische Listen hat sich auch ein Verfahren bewährt, bei dem ein allgemeiner Baum durch einen Binärbaum implementiert wird (Bild 4):

Die rötliche Linie zeigt dabei den realisierten allgemeinen Baum, während die Pfeile die tatsächlichen Zeigerstrukturen repräsentieren. Das Grundprinzip besteht darin, dass ein Zeiger jeweils auf den am weitesten links stehenden Sohn zeigt, während der andere auf den rechten Bruder verweist. Hierbei ist zwar ein direkter Zugriff auf einzelne bestimmte Sohn-Knoten nicht mehr möglich, da man sich über die Geschwister voranarbeiten muss. Dafür ist diese Implementierung sehr speichereffizient.

Als Binärbaum bezeichnet man in der Graphentheorie eine spezielle Form eines Graphen. Genauer gesagt handelt es sich um einen gewurzelten Baum, bei dem jeder Knoten höchstens zwei Kindknoten besitzt. Oft wird verlangt, dass sich die Kindknoten eindeutig in linkes und rechtes Kind einteilen lassen.

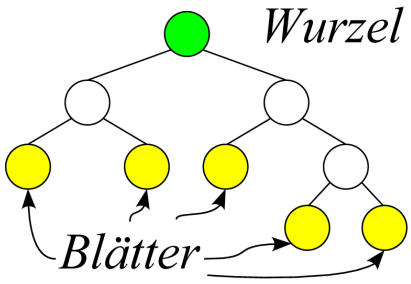


Abbildung 5: Ein voller, aber nicht vollständiger Binärbaum

Eine verbale Definition: Ein Baum ist entweder leer, oder er besteht aus einem linken oder rechten Teilbaum, die wiederum Bäume sind!

Ein Binärbaum heißt geordnet, wenn jeder innere Knoten ein linkes und eventuell zusätzlich ein rechtes Kind besitzt (und nicht etwa nur ein rechtes Kind). Man bezeichnet ihn als voll oder strikt, wenn jeder Knoten entweder Blatt ist (also kein Kind besitzt), oder aber zwei (also sowohl ein linkes wie ein rechtes) Kinder besitzt. Man bezeichnet ihn als vollständig, wenn alle Blätter die gleiche Tiefe besitzen. Induktiv lässt sich zeigen, dass ein vollständiger Binärbaum der Höhe n, den man häufig auch als B_n bezeichnet, genau

- $2^{n+1} - 1$ Knoten,
- $2^n - 1$ innere Knoten,
- 2^i Knoten in Tiefe i, insbesondere also
- 2^n Blätter

besitzt, wobei mit Höhe n die Länge des Pfades zu einem tiefsten Knoten bezeichnet wird. Eine Darstellung eines Binärbaumes, in dem die Knoten mit rechtwinkligen Dreiecken und die Kanten mit Rechtecken dargestellt werden, nennt man pythagoreischen Binärbaum.

Es gibt verschiedene Möglichkeiten, die Knoten von Binärbäumen zu durchlaufen. Diesen Prozess bezeichnet man auch als Linearisierung oder Traversierung. Man unterscheidet hier in

- pre-order (WLR): wobei zuerst die Wurzel (W) betrachtet wird und anschließend zuerst der linke (L), dann der rechte (R) Teilbaum durchlaufen wird,
- in-order (LWR): wobei zuerst der linke (L) Teilbaum durchlaufen wird, dann die Wurzel (W) betrachtet wird und anschließend der rechte (R) Teilbaum durchlaufen wird und
- post-order (LRW): wobei zuerst der linke (L), dann der rechte (R) Teilbaum durchlaufen wird und anschließend die Wurzel (W) betrachtet wird.
- level-order Beginnend bei der Wurzel, werden die Ebenen von links nach rechts durchlaufen.

Rekursive Implementierung:

```
Funktion Preorder(Baum)
   W <- Baum.Wurzel
      //W:= Wurzel des übergebenen Baumes
   If Baum.Links <> NULL
      //Existiert ein linker Unterbaum?
   L <- Preorder(Baum.Links)
      //  dann: L:= Preorder von linkem Unterbaum
   If Baum.Rechts <> NULL
      //Existiert ein rechter Unterbaum?
   R <- Preorder(Baum.Rechts)
      //  dann: R:= Preorder von rechtem Unterbaum
   Return W°L°R
      //Rückgabe: Verkettung aus W, L und R


Funktion Inorder(Baum)
   W <- Baum.Wurzel
      //W:= Wurzel des übergebenen Baumes
   If Baum.Links <> NULL
      //Existiert ein linker Unterbaum?
   L <- Inorder(Baum.Links)
      //  dann: L:= Inorder von linkem Unterbaum
   If Baum.Rechts <> NULL
      //Existiert ein rechter Unterbaum?
   R <- Inorder(Baum.Rechts)
      //  dann: R:= Inorder von rechtem Unterbaum
   Return L°W°R
      //Rückgabe: Verkettung aus L, W und R


Funktion Postorder(Baum)
   W <- Baum.Wurzel
      //W:= Wurzel des übergebenen Baumes
   If Baum.Links <> NULL
      //Existiert ein linker Unterbaum?
   L <- Postorder(Baum.Links)
      //  dann: L:= Postorder von linkem Unterbaum
   If Baum.Rechts <> NULL
      //Existiert ein rechter Unterbaum?
   R <- Postorder(Baum.Rechts)
      //  dann: R:= Postorder von rechtem Unterbaum
   Return L°R°W
      //Rückgabe: Verkettung aus L, R und W
```

6.3.2 Heap

In der Informatik ist ein Heap (manchmal auch als Haufen oder Halde bezeichnet) eine zumeist auf Bäumen basierende abstrakte Datenstruktur. In einem Heap können Objekte oder Elemente abgelegt und aus diesem wieder entnommen werden. Sie dienen damit der Speicherung von Mengen. Den Elementen ist dabei ein Schlüssel zugeordnet, der die Priorität der Elemente festlegt. Häufig werden auch die Elemente selbst als Schlüssel verwendet.

Über die Menge der Schlüssel muss daher eine totale Ordnung festgelegt sein, über welche die Reihenfolge der eingefügten Elemente festgelegt wird. Beispielsweise könnte die Menge der ganzen Zahlen zusammen mit der Kleiner-Relation ($<$) als Schlüssel-Menge fungieren.

Der Begriff Heap wird häufig als bedeutungsgleich zu einem partiell geordneten Baum verstanden. Gelegentlich versteht man einschränkend nur eine besondere Implementierungsform eines partiell geordneten Baums, nämlich die Einbettung in ein Array, als Heap.

Verwendung finden Heaps vor allem dort, wo es darauf ankommt, schnell ein Element mit höchster Priorität aus dem Heap zu entnehmen, beispielsweise bei Vorrangwarteschlangen.

Man unterscheidet Heaps in Min-Heaps und Max-Heaps. Bei Min-Heaps bezeichnet man die Eigenschaft, dass die Schlüssel der Kinder eines Knotens stets größer als der Schlüssel ihres Vaters sind, als Heap-Bedingung. Dies bewirkt, dass an den Wurzeln des Baumes stets ein Element mit minimalem Schlüssel im Baum zu finden ist. Umgekehrt verlangt die Heap-Bedingung bei Max-Heaps, dass die Schlüssel der Kinder eines Knotens stets kleiner als die ihres Vaters sind. Hier befindet sich an der Wurzel des Baumes immer ein Element mit maximalem Schlüssel.

Mathematisch besteht der Unterschied zwischen beiden Varianten nur in ihrer gegensätzlichen Ordnung der Elemente. Da die Definition von auf- und absteigend rein willkürlich ist, ist es also eine Auslegungsfrage, ob es sich bei einer konkreten Implementierung um einen Min-Heap oder um einen Max-Heap handelt.

Oft kommt es auch vor, dass ein Heap die Heapeigenschaft in beiden Richtungen abbilden soll. In diesem Fall werden einfach zwei Heaps geschaffen, wobei der eine nach der Kleiner- und der andere nach der Größer-Relation anordnet. Eine derartige Datenstruktur wird dann Doppelheap oder kurz Deap genannt.

Es existieren zahlreiche Arten von Heaps mit unterschiedlich gutem Laufzeitverhalten für die verschiedenen Operationen, die sie zur Verfügung stellen. Beispiele für Heaps sind:

- Binärer Heap
- Binomial-Heap
- Fibonacci-Heap
- Treap

Heaps haben ein breites Spektrum an Anwendungen. Häufig ist vor allem der Einsatz in Vorrangwarteschlangen, wie sie bei Servern oder Betriebssystemen zur Festlegung der Ausführungsreihenfolge von Aufgaben benötigt werden.

Daneben gibt es aber auch Anwendungen, die nicht explizit den Einsatz von Warteschlangen verlangen. Allgemein stellt ein Heap eine ideale Datenstruktur für Greedy-Algorithmen dar, die schrittweise lokale optimierte Entscheidungen treffen. So wird zum Beispiel bei dem Sortieralgorithmus Heapsort ein Binärer Heap zum Sortieren eingesetzt. Fibonacci-Heaps finden beim Algorithmus von Dijkstra bzw. Prim Anwendung.

6.3.3 Beispiel für die Überführung in einen Max-Heap

Man möchte ein Array mit dem Inhalt [H|E|A|P|S|O|R|T] in einen Max-Heap überführen, wobei gilt: A<B<C<...<Y<Z.

```
1 2 3 4 5 6 7 8
H E A P S O R T    Wir beginnen links von der Mitte, d. h. bei P:
      ^       ^    sein Nachfolger ist T. Da T>P ist, tauschen wir beide.
H E A T S O R P    Wir fahren mit dem A fort. Seine Nachfolger sind
    ^     ^ ^      O und R. Es gilt sowohl R>O als auch R>A.
                   Also tauschen wir R und A.
H E R T S O A P    Dann vergleichen wir E mit seinen Nachfolgern T und S.
  ^   ^ ^          Es gilt T>S und T>E. Deshalb müssen wir T und E
                   vertauschen.
H T R E S O A P    Wir müssen nun E weiter versickern, denn der neue
      ^       ^    Nachfolger von E ist P, und P>E.
H T R P S O A E    Nun vergleichen wir H mit seinen
```

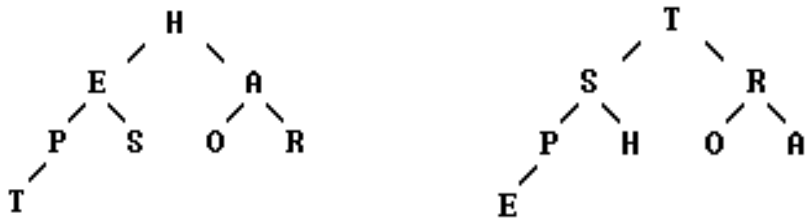


Abbildung 6: Repräsentation des Arrays [H|E|A|P|S|O|R|T] als Binärbaum (links) und in einen Max-Heap überführter Binärbaum, entspricht [T|S|R|P|H|O|A|E] (rechts)

```
^ ^ ^             Nachfolgern T und R. T>R und T>H.
T H R P S O A E   Wir versickern das H weiter.
  ^   ^ ^         S>P und S>H.
T S R P H O A E   Wir haben das Array nun in einen Max-Heap überführt.
```

6.3.4 Prinzip der Sortierung und Implementierung

Der eigentliche Sortieralgorithmus nutzt die Tatsache aus, dass die Wurzel eines Heaps stets der größte Knoten ist. Da im fertig sortierten Array der größte Wert ganz rechts stehen soll, vertauscht man das erste mit dem letzten Arrayelement. Das Element am Ende des Arrays ist nun an der gewünschten Position und bleibt dort. Den Rest des Arrays muss man wieder in einen Heap überführen, indem man das erste Element versickert. Anschließend vertauscht man das erste mit dem vorletzten Element, d. h. die beiden größten Werte sind wie gewünscht am Ende des Arrays. Man versickert das nun erste Element, usw.

```
// Java
// vertauscht in einem Array die Einträge mit Index x und y
   private static void vertausche(int[] a, int x, int y) {
      int z = a[x]; a[x] = a[y]; a[y] = z;
   }

   // versickert im Array mit Länge n das Element mit Index i
   private static void versickere(int[] a, int n, int i) {
      while (i <= n/2) {
       int ln = 2*i;
          // linker Nachfolger
       if (ln < n)
          // hat der Knoten einen rechten Nachfolger, und
          if (a[ln+1] > a[ln])
             // ist der größer als der linke?
          ln++;
             // dann benutze den rechten, sonst den linken Nachfolger
       if (a[ln] > a[i]) {
          vertausche(a, i, ln);
         // vertausche mit größerem Nachfolger
          i = ln; // versickere weiter
       }
       else {
             // beide Nachfolger sind kleiner als das Element, kann nicht
          i = n;
        // weiter versickern: Beende Schleife
       }
      }
   }

// überführt ein Array in einen Heap
```

```
   private static void überführeInHeap(int[] a, int n) {
      for (int i = n/2; i >= 1; i--) {
            // starte von der Mitte aus rückwärts
         versickere(a, n, i);
      }
   }

// sortiert ein Array von ganzen Zahlen
   public static void heapSort(int[] a, int n) {
      überführeInHeap(a, n);
         // stelle Heap-Eigenschaft her
      for (int i = n; i >= 1; i--) {
         vertausche(a, 1, i);
        // vertausche 1. mit letztem unsortierten Element
         versickere(a, i-1, 1);
        // stelle Heap-Eigenschaften für Rest-Array her
      }
   }/Lücke
```

6.4 Insertion Sort

Insertionsort (engl. insertion - Einfügen) ist ein einfaches stabiles Sortierverfahren. Es ist weit weniger effizient als andere anspruchsvollere Sortierverfahren. Dafür hat es jedoch folgende Vorteile:

- einfach zu implementieren
- effizient bei (ziemlich) kleinen Eingabemengen
- effizient bei Eingabemengen, die schon vorsortiert sind
- stabil (d.h., die Reihenfolge von schon sortierten Elementen ändert sich nicht)
- minimaler Speicherverbrauch, da er ortsfest arbeitet

Der Algorithmus entnimmt der unsortierten Eingabemenge ein beliebiges (z. B. das Erste) Element und fügt es an richtiger Stelle in die (anfangs leere) Ausgabemenge ein. Geht man in der Reihenfolge der ursprünglichen Menge vor, so ist es außerdem stabil. Wird auf einem Array gearbeitet, so müssen die Elemente nach dem neu eingefügten Element verschoben werden. Dies ist die eigentlich teure Operation von Insertionsort, da das Finden der richtigen Einfügeposition über eine binäre Suche vergleichsweise effizient erfolgen kann.

Der folgende Pseudocode sortiert die Eingabemenge aufsteigend. Um eine absteigende Sortierung zu erreichen, ist der zweite Vergleich in Zeile 4 entsprechend zu ändern.

```
a: Liste (Array) mit der unsortierten Eingabemenge

insertion_sort(a)
1  von j <-- 2 bis länge[a]
2     führe aus aktueller_wert <- a[j]
3         i <-- j - 1
4         solange i > 0 und a[i] > aktueller_wert
5             führe aus a[i + 1] <-- a[i]
6                 i <-- i - 1
7         a[i + 1] <-- aktueller_wert
```

Die Anzahl der Vergleiche und Verschiebungen des Algorithmus ist von der Anordnung der Elemente in der unsortierten Eingangsmenge abhängig. Für den

Average-Case ist eine Abschätzung der Laufzeit daher nicht so einfach möglich. Im Best Case ist die Komplexität jedoch linear (O(n)), d.h. sogar besser als bei den komplizierten Verfahren (QuickSort, MergeSort, HeapSort etc.). Im Worst Case dagegen quadratisch (O(n^2)).

Wenn zur Bestimmung der richtigen Position eines Elementes die binäre Suche benutzt wird, kann man die Anzahl der Vergleiche, im Worst Case, durch

$$\log(n!) = n \log n n \log e + O(\log n) = n \log n 0,4426n + O(\log n) \qquad (23)$$

abschätzen. Die Anzahl der Schiebeoperationen sind im Average-Case $(n * (n-1))/4 = O(n^2)$.

Der Worst Case wäre ein absteigend sortiertes Array a, da wir bei jedem Element i die Methode move(i,0,a) aufrufen müssen. Die Anzahl der Schiebeoperationen beträgt in diesem Fall $((n * (n+1))/2)n = O(n^2)$.

6.5 Shell Sort

Shell sort is a sorting algorithm that requires asymptotically fewer than O(n^2) comparisons and exchanges in the worst case. Although it is easy to develop an intuitive sense of how this algorithm works, it is very difficult to analyze its execution time, but estimates range from O($n \log^2 n$) to O($n^{1.5}$) depending on implementation details.

Shell sort is a generalization of insertion sort, with two observations in mind:

- Insertion sort is efficient if the input is "almost sorted".
- Insertion sort is inefficient, on average, because it moves values just one position at a time.

Shell sort improves insertion sort by comparing elements separated by a gap of several positions. This lets an element take "bigger steps" toward its expected position. Multiple passes over the data are taken with smaller and smaller gap sizes. The last step of Shell sort is a plain insertion sort, but by then, the array of data is guaranteed to be almost sorted.

Consider a small value that is initially stored in the wrong end of the array. Using an O(n^2) sort such as bubble sort or insertion sort, it will take roughly n comparisons and exchanges to move this value all the way to the other end of the array. Shell sort first moves values using giant step sizes, so a small value will move a long way towards its final position, with just a few comparisons and exchanges.

For example, consider a list of numbers like [13 14 94 33 82 25 59 94 65 23 45 27 73 25 39 10]. If we started with a step-size of 5, we could visualize this as breaking the list of numbers into a table with 5 columns. This would look like this:

13	14	94	33	82
25	59	94	65	23
45	27	73	25	39
10				

We then sort each column (using an insertion sort), which gives us

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 swaps

Abbildung 7: Shell Sort in action

10 14 73 25 23
13 27 94 33 39
25 59 94 65 82
45

When read back as a single list of numbers, we get [10 14 73 25 23 13 27 94 33 39 25 59 94 65 82 45]. Here, the 10 which was all the way at the end, has moved all the way to the beginning.

This list is then again sorted using a 3-gap sort, and then 1-gap sort (simple insertion sort). While transforming the list into a table makes it easier to visualize, the algorithm itself does its sorting in-place by incrementing the index by the step size. i += step_size instead of i++

The Shell sort is named after its inventor, Donald Shell, who published it in 1959. Some older textbooks and references incorrectly call this the "Shell-Metzner" sort after Marlene Metzner Norton, but according to Metzner, "I had nothing to do with the sort, and my name should never have been attached to it."

6.5.1 Gap Sequence

The gap sequence is an integral part of the shellsort algorithm. Any increment sequence will work, so long as the last element is 1. The algorithm begins by performing a gap insertion sort, with the gap being the first number in the gap sequence. It continues to perform a gap insertion sort for each number in the sequence, until it finishes with a gap of 1. When the gap is 1, the gap insertion sort is simply an ordinary insertion sort, guaranteeing that the final list is sorted.

The gap sequence that was originally suggested by Donald Shell was to begin with $N/2$ and to halve the number until it reaches 1. While this sequence provides significant performance enhancements over the quadratic algorithms such as insertion sort, it can be changed slightly to further decrease the average and worst-case running times. Weiss' textbook demonstrates that this sequence allows a worst case $O(n^2)$ sort, if the data is initially in the array as (small_1, large_1, small_2, large_2, ...) - that is, the upper half of the numbers are placed, in sorted order, in the even index locations and the lower end of the numbers are placed similarly in the odd indexed locations.

Perhaps the most crucial property of Shellsort is that the elements remain k-sorted even as the gap diminishes. For instance, if a list was 5-sorted and then 3-sorted, the list is now not only 3-sorted, but both 5- and 3-sorted. If this were not true, the algorithm would undo work that it had done in previous iterations, and would not achieve such a low running time.

Depending on the choice of gap sequence, Shellsort has a proven worst-case running time of $O(n^2)$ (using Shell's increments that start with $1/2$ the array size and divide by 2 each time), $O(n^{3/2})$ (using Hibbard's increments of $2^k - 1$), $O(n^{4/3})$, or $O(n \log^2 n)$, and possibly unproven better running times. The existence of an $O(n \log n)$ worst-case implementation of Shellsort remains an open research question.

Teil III
Objektoriertes Programmieren

7 Grundlagen

Grundlegende Merkmale der objektorienterten Programmierung sind: Kapselung der Daten, Vergabe von Zugriffsrechten und Vererbung.

Die objektorientierte Programmierung, kurz OOP, ist ein auf dem Konzept der Objektorientierung basierendes Programmierparadigma, welches Flexibilität und Wiederverwendbarkeit von Programmen fördert. Die Grundidee der objektorientierten Programmierung ist, Daten und Funktionen, die auf diese Daten angewendet werden können, möglichst eng in einem sogenannten Objekt zusammenzufassen und nach außen hin zu kapseln, so dass Methoden fremder Objekte diese Daten nicht versehentlich manipulieren können. Im Gegensatz dazu beschreibt das vor der OOP vorherrschende Paradigma eine strikte Trennung von Funktionen (Programmcode) und Daten, dafür aber eine schwächere Strukturierung der Daten selbst.

Gängige moderne Programmiersprachen wie Java, C++ und C# unterstützen sowohl die OOP als auch den prozeduralen Ansatz, der in den gängigen Programmiersprachen der 1970er und 1980er Jahre wie Pascal, Fortran oder C vorherrschend war. Auch die Skriptsprache Python dagegen setzt z.B. auf OOP. Die objektorientierte Programmierung wurde bereits Ende der sechziger Jahre als Lösungsansatz für die Modularisierung und die Wiederverwendbarkeit von Code entwickelt. Die erste objektorientierte Programmiersprache war Simula-67.

8 Objektorientierung am Beispiel der Implementierung in Java

- Kapselung der Daten: Für jeden Datentyp wird eine "Class" entworfen, die die Daten und Funktionen zur Verwendung auf diesen Daten enthält.
- Zugriffsrechte: Jeder Klasse, jedem Feld (=Variable) und jeder Methode (=Funktion) werden Zugriffsrechte mittels vorangestelltem **public, private, package** oder **protected** zugeteilt. Dies entscheidet, ob auf die Daten und Funktionen von außerhalb der Klasse zugegriffen werden darf.

  Dadurch ist ein effektiver Entwurf großer Programme möglich, da z.B. durch das Verbot der direkten Manipulation der Variablen (z.B. private int x) und gleichzeitige Definition einer Funktion zur Manipulation der Variablen (z.B. public void setx(int n) und public int getx()) in die Funktion Sicherheitsabfragen eingeführt werden können die z.B. Divisionen durch 0 verhindern oder Ähnliches.
- Vererbung: Durch den Befehl **A extends B** erbt eine Klasse A von einer anderen Klasse B alle Variablen und Funktionen, d.h. kann diese ebenfalls verwenden. A nennt man dann Ableitung, Erweiterung oder Subklasse von B. B heißt Oberklasse, Superklasse oder Basisklasse von A.

- Gleichnamige Variablen in Subklassen machen die Variablen der Superklasse nur unsichtbar, lassen sie aber nicht verschwinden (in der Subklasse wird die Subklassenvariable verwendet). Anders als bei lokalen Variablen kann das Superklassenfeld weiter verwendet werden mit **super.variable** Will man explizit das Subklassenfeld ansprechen tut man dies durch **this.variable** . Dieser Mechanismus heißt **Schattenvariable**.
- Gleichnamige Methoden in Subklassen verhindern den Aufruf der Superklassenfunktion. ("Überschriebene Funktion")

8.1 Applet & Servlet

8.1.1 Applet

A Java applet is an applet delivered in the form of Java bytecode. Java applets can run in a Web browser using a Java Virtual Machine (JVM), or in Sun's AppletViewer, a stand alone tool to test applets. Java applets were introduced in the first version of the Java language in 1995. Java applets are usually written in the Java programming language but they can also be written in other languages that compile to Java bytecode such as Jython.

Applets are used to provide interactive features to web applications that cannot be provided by HTML. Since Java's bytecode is platform independent, Java applets can be executed by browsers for many platforms, including Windows, Unix, Mac OS and Linux.There are open source tools like applet2app which can be used to convert an applet to a stand alone Java application/windows executable. This has the advantage of running a Java applet in offline mode without the need for internet browser software.

A Java Servlet is sometimes informally compared to be llikeä server-side applet, but it is different in its language, functions, and in each of the characteristics described here about applets.

Java applets are executed in a sandbox by most web browsers, preventing them from accessing local data. The code of the applet is downloaded from a web server and the browser either embeds the applet into a web page or opens a new window showing the applet's user interface. The applet can be displayed on the web page by making use of the deprecated applet HTML element, or the recommended object element. This specifies the applet's source and the applet's location statistics.

A Java applet extends the class java.applet.Applet, or in the case of a Swing applet, javax.swing.JApplet. The class must override methods from the applet class to set up a user interface inside itself (Applet is a descendant of Panel which is a descendant of Container).

Vor- & Nachteile

Vorteile A Java applet can have any or all of the following advantages:

- it is simple to make it work on Windows, Mac OS and Linux, i.e. to make it cross platform

- the same applet can work on "all" installed versions of Java at the same time, rather than just the latest plug-in version only. However, if an applet requires a later version of the JRE the client will be forced to wait during the large download.
- it runs in a sandbox, so the user does not need to trust the code, so it can work without security approval
- it is supported by most web browsers
- it will cache in most web browsers, so will be quick to load when returning to a web page
- it can have full access to the machine it is running on if the user agrees
- it can improve with use: after a first applet is run, the JVM is already running and starts quickly, benefitting regular users of Java
- it can run at a comparable (but generally slower) speed to other compiled languages such as C++
- it can be a real time application
- it can move the work from the server to the client, making a web solution more scalable with the number of users/clients

Nachteile A Java applet is open to any of the following disadvantages:

- it requires the Java plug-in, which isn't available by default on all web browsers
- it can't start up until the Java Virtual Machine is running, and this may have significant startup time the first time it is used
- if it is uncached, it must be downloaded (usually over the internet), and this takes time
- it is considered more difficult to build and design a good user interface with applets than with HTML-based technologies
- if untrusted, it has severely limited access to the user's system - in particular having no direct access to the client's disc or clipboard
- some organizations only allow software installed by the administrators. As a result, many users cannot view applets by default.
- applets may require a specific JRE.
- if the JRE crashes it will take the browser down with it–this happens on occasion.
- if there is a server error, then it may not be able to retrieve mandatory files.

8.1.2 Servlet

The Java Servlet API allows a software developer to add dynamic content to a Web server using the Java platform. The generated content is commonly HTML, but may be other data such as XML. Servlets with JavaServer Pages are the Java counterpart to dynamic web content technologies such as CGI/PHP or ASP.NET/VBScript, JScript, C#. Servlets can maintain state across many server transactions by using HTTP cookies, session variables or URL rewriting.

The Servlet API, contained in the Java package hierarchy javax.servlet, defines the expected interactions of a web container and a servlet. A web container is essentially the component of a web server that interacts with the servlets. The web container is responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access rights.

A Servlet is an object that receives requests (ServletRequest) and generates a response (ServletResponse) based on the request. The API package javax.servlet.http defines HTTP subclasses of the generic servlet (HttpServlet) request (HttpServletRequest) and response (HttpServletResponse) as well as an (HttpSession) that tracks multiple requests and responses between the web server and a client. Servlets may be packaged in a WAR file as a Web application.

Moreover, servlets can be generated automatically by JavaServer Pages (JSP), or alternately by template engines such as WebMacro. Often servlets are used in conjunction with JSPs in a pattern called "Model 2", which is a flavor of the model-view-controller pattern.

The Servlet life cycle consists of the following steps:

1. The Servlet class is loaded by the container during start-up.

2. The container calls the init() method. This method initializes the servlet and must be called before the servlet can service any requests. In the entire life of a servlet, the init method is called only once.

3. After initialization, the servlet can service client-requests. Each request is serviced in its own separate thread. The container calls the service() method of the servlet for every request. The service() method determines the kind of HTTP request (GET, POST etc) and accordingly calls the methods doGet(), doPost(), doTrace() etc. The developer of the servlet must provide implementation for these methods. If an implementation for doPost() has not been provided, it means that the servlet cannot handle POST requests. A developer must never overload the service() method.

4. Finally, the container calls the destroy() method which takes the servlet out of service. The destroy() method like init() is called only once in the life-cycle of a Servlet.

There is only one ServletContext in every application. This object can be used by all the servlets to obtain application level information or container details. Every servlet, on the other hand, gets its own ServletConfig object. This object provides initialization parameters for a servlet. A developer can obtain the reference to ServletContext using the ServletConfig object.

8.2 Thread

A thread in computer science is short for a thread of execution. Threads are a way for a program to split itself into two or more simultaneously (or pseudo-simultaneously) running tasks. Threads and processes differ from one operating system to another, but in general, the way that a thread is created and shares its resources is different from the way a process does.

Multiple threads can be executed in parallel on many computer systems. This multithreading generally occurs by time slicing, wherein a single processor switches between different threads, in which case the processing is not literally simultaneous, for the single processor is only really doing one thing at a time. This switching can happen so fast as to give the illusion of simultaneity to an end user. For instance, a typical PC today contains only one processor core, but you can run multiple programs at once, such as a word processor alongside an audio playback program; though the user experiences these things as simultaneous, in truth, the processor is quickly switching back and forth between these separate processes. On a multiprocessor or multi-core system, threading can be achieved via multiprocessing, wherein different threads and processes can run literally simultaneously on different processors or cores.

Many modern operating systems directly support both time-sliced and multiprocessor threading with a process scheduler. The operating system kernel allows programmers to manipulate threads via the system call interface. Some implementations are called a kernel thread, whereas a lightweight process is a specific type of kernel thread that shares the same state and information.

Absent that, programs can still implement threading by using timers, signals, or other methods to interrupt their own execution and hence perform a sort of ad hoc time-slicing. These are sometimes called user-space threads.

8.2.1 Beispiel

This is an example of a simple multi-threaded program written in the Java programming language. The program calculates prime numbers until the user types the word "stop". Then the program prints how many prime numbers it found and exits. This example demonstrates how threads can access the same variable while working asynchronously. This example also demonstrates a simple "race condition". The thread printing prime numbers continues to do so for a short time after the user types "stop". Of course, this problem is easily corrected using standard programming techniques.

```
import java.io.*;

public class Example implements Runnable
{
     static Thread threadCalculate;
     static Thread threadListen;
     long totalPrimesFound = 0;

     public static void main (String[] args)
     {
          Example e = new Example();

          threadCalculate = new Thread(e);
          threadListen = new Thread(e);

          threadCalculate.start();
          threadListen.start();
     }
```

```
    public void run()
    {
         Thread currentThread = Thread.currentThread();

  if (currentThread == threadCalculate)
       calculatePrimes();
  else if (currentThread == threadListen)
       listenForStop();
    }

    public void calculatePrimes()
    {
         int n = 1;

         while (true)
         {
              n++;
              boolean isPrime = true;

              for (int i = 2; i < n; i++)
                   if ((n / i) * i == n)
                   {
                        isPrime = false;
                         break;
                   }

                   if (isPrime)
           {
                totalPrimesFound++;
                System.out.println(n);
           }
  }
    }

    private void listenForStop()
    {
         BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
         String line = "";

         while (!line.equals("stop"))
         {
              try
      {
           line = input.readLine();
              }
  catch (IOException exception) {}
         }

         System.out.println("Found " + totalPrimesFound +
  " prime numbers before you said stop.");
  System.exit(0);
    }
}
```

9 Design Patterns - Entwurfsmuster

Der primäre Nutzen eines Entwurfsmusters liegt in der Beschreibung einer Lösung für eine bestimmte Klasse von Entwurfsproblemen. Weiterer Nutzen ergibt sich aus der Tatsache, dass jedes Muster einen Namen hat. Dies vereinfacht die Diskussion unter Entwicklern, da man abstrakt über eine Struktur sprechen kann. So sind etwa Software-Entwurfsmuster - im Gegensatz zu Idiomen - zunächst einmal unabhängig von der konkreten Programmiersprache.

Wenn der Einsatz von Entwurfsmustern dokumentiert wird, ergibt sich ein weiterer Nutzen dadurch, dass durch die Beschreibung des Musters ein Bezug zur dort vorhandenen Diskussion des Problemkontextes und der Vor- und Nachteile der Lösung hergestellt wird.

9.1 Unified Modelling Language

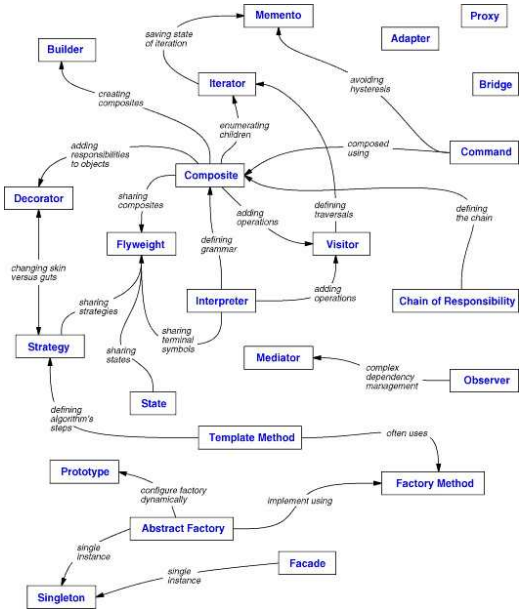


Abbildung 8: Interaktionsdiagramm verschiedener Design Patterns

Will man die Interaktion verschiedener Klassen untereinander angeben so benutzt man dazu die Unified Modeling Language.

9.2 Delegation

m Bereich der objektorientierten Programmierung bezeichnet der Begriff der Delegation eine Form der Interaktion von Klassen. Im Gegensatz zur Vererbung gibt eine Klasse bei der Delegation einen Teil ihrer Verantwortung an eine andere Klasse ab.

Bei der Delegation gibt eine Klasse die Verantwortlichkeit für einen Teil ihrer Funktionalität an eine andere Klasse ab. Dies kann zum Beispiel der Aufruf einer Methode einer Hilfsklasse sein, die auch von anderen Klassen verwendet wird und deren Funktionalität nicht originär Teil der aufrufenden Klasse ist. Ein anderes Beispiel ist der Verwendung des Entwurfsmusters Fassade, bei dem eine Klasse eine Fassade, das heißt eine Schnittstelle zu einem komplexen Untersystem darstellt. In diesem Fall besteht die Aufgabe der Fassade hauptsächlich in der Delegation von Funktionalität an andere Unterklassen des Subsystems.

Einen Spezialfall stellen reine Wrapper-Methoden dar, die Funktionsaufrufe an eine andere Klasse weiterleiten. Dies wird häufig verwendet, um dem Gesetz von Demeter zu genügen.

Gerade unerfahrene Programmierer bevorzugen oft die Vererbung gegenüber der Delegation, um die Funktionalität einer anderen Klasse einzubinden. Zum Beispiel könnte eine Klasse Auto dabei von einer Klasse Liste abgeleitet werden, da ja so die Begründung ein Auto aus mehreren Teilen besteht, die in Auto-Objekten enthalten sein müssen. Dieser Ansatz hat aber mehrere Probleme:

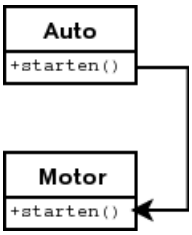


Abbildung 9: UML-Diagramm: Delegation
Die Klasse Auto delegiert einen Teil ihrer Verantwortung an die Klasse Motor

- Philophisch gesehen ist ein Auto keine Liste, es verwendet lediglich eine solche zur Verwaltung seiner Teile. Durch den falschen Einsatz der Vererbung kann ein falscher Eindruck erweckt werden.
- Die Klasse Auto stellt durch den Einsatz der Vererbung alle Methoden der Klasse Liste zur Verfügung. Jedoch ist semantisch nicht klar, was eine Methode hinzufuegen hier zu bedeuten hat. Besser wäre der Einsatz spezifischer Methoden.
- Eine solche Art der Vererbung kann leicht zu einer Verletzung des Liskovschen Substitutionsprinzipes führen, das heißt die abgeleitete Klasse erfüllt nicht den Vertrag der Basisklasse; sie verhält sich anders.

Aus diesem Grund wäre bei diesem Beispiel die Verwendung der Delegation oder der Aggregation sinnvoller gewesen.

9.3 Object Composition

In computer science, object composition (not to be confused with function composition) is a way and practice to combine simple objects or data types into more complex ones. Compositions are a critical building block of many basic data structures, including the tagged union, the linked list, and the binary tree, as well as the object used in object-oriented programming.

Composition is contrasted with subtyping, which is the process of adding detail to a general data type to create a more specific data type. In composition, the composite type "has an" object of a simpler type, while in subtyping, the subtype "is an" instance of its parent type. Composition does not form a subtype but a new type.

A real-world example of composition may be seen in an automobile: the objects wheel, steering wheel, seat, gearbox and engine may have no functionality by themselves, but if you composite them, they may form an automobile object, which has a higher function, greater than the sum of its parts in a trite sense.

Composited objects are called fields, items, members or attributes, and the resulting composition a structure, record, tuple, user-defined type (UDT), or composite type. The terms usually vary across languages. Fields are given a unique name so that each one can be distinguished from the others. Sometimes an issue of ownership arises: when a composition is destroyed, should objects belonging to it be destroyed as well? If not, the case is sometimes called aggregation. For more, see the aggregation section below.

In UML, composition is depicted as a filled diamond. It always implies a multiplicity of 1 or 0..1, as no more than one object at a time can have lifetime responsibility for another object. The more general form of composition, that of aggregation, is depicted as an unfilled diamond.

9.3.1 Beispiel

This is an example of composition in C:

```
typedef struct {
   int flange;
   char *name;
   enum { male, female } sex;
} Person;
```

In this example, the primitive types int, char and enummale, female are combined to form the composite type of Person. Each object of type Person then "has an" age, name, and sex.

If a Person type were instead created by subtyping, it might be a subtype of Organism, and it could inherit some attributes from Organism (every organism has an age), while extending the definition of Organism with new attributes (not every organism has a sex, but every person does).

9.3.2 Aggregation

Aggregation differs from ordinary composition in that it does not imply ownership. In composition, when the owning object is destroyed, so are the contained objects. In aggregation, this is not necessarily true. For example, a university owns various departments (e.g., chemistry), and each department has a number of professors. If the university closes, the departments will no longer exist, but the professors in those departments will continue to exist. Therefore, a University can be seen as a composition of departments, whereas departments have an aggregation of professors. In addition, a Professor could work in more than one department, but a department could not be part of more than one university.

Composition is usually implemented such that an object contains another object. For example, in C++:

```
class Department;

class University
{
     ...
     private:
     Department faculty[20];
     ...
};
```

In aggregation, the object may only contain a reference or pointer to the object:

```
class Professor;

class Department
{
     ...
     private:
     Professor* members[5];
     ...
};
```

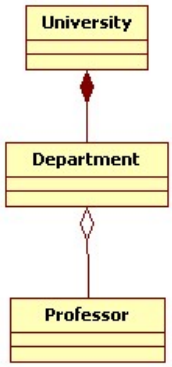


Abbildung 10: UML-Diagramm Aggregation

Sometimes aggregation is referred to as composition when the distinction between ordinary composition and aggregation is unimportant.

9.4 Singelton

Das Einzelstück (engl. Singleton) ist ein in der Softwareentwicklung eingesetztes Entwurfsmuster und gehört zur Kategorie der Erzeugungsmuster (engl. Creational Patterns). Es stellt sicher, dass zu einer Klasse nur genau ein Objekt erzeugt werden kann und ermöglicht einen globalen Zugriff auf dieses Objekt.

Das Einzelstück findet Verwendung, wenn

- nur ein Objekt zu einer Klasse existieren darf und ein einfacher Zugriff auf dieses Objekt benötigt wird oder
- wenn das einzige Objekt durch Unterklassenbildung spezialisiert werden soll.

Anwendungsbeispiele sind:

- Ein zentrales Protokoll-Objekt, das Ausgaben in eine Datei schreibt.
- Druckaufträge, die zu einem Drucker gesendet werden, sollen nur in einen einzigen Puffer geschrieben werden.

In Klammer stehen die Bezeichnungen aus der Abb. 11.

- Einzelstück (Singleton)
  - erzeugt und verwaltet einziges Objekt zu einer Klasse
  - bietet globalen Zugriff auf dieses Objekt über eine Instanzoperation
  - die Instanzoperation ist eine Klassenmethode, d. h. statisch gebunden
  - (das Attribut Instanz (instance) ist ein Klassenattribut, d. h. ein statisches Attribut)

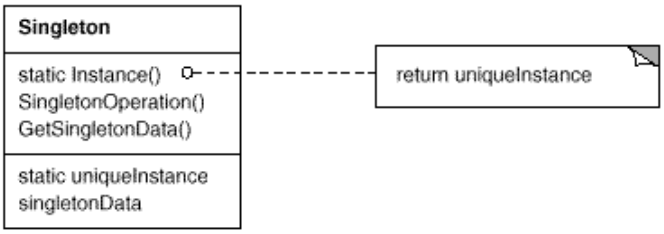


Abbildung 11: UML-Diagramm: Singleton

9.4.1 Vor- & Nachteile

Vorteile:

- Das Muster bietet eine Verbesserung gegenüber globalen Variablen.
- Das Einzelstück kann durch Unterklassenbildung spezialisiert werden.
- Sollten später mehrere Objekte benötigt werden, ist eine Änderung leicht(er) möglich.

Nachteile:

- Es besteht die große Gefahr, durch exzessive Verwendung von Singletons quasi ein objektorientiertes Äquivalent zu globalen Variablen zu implementieren.
- Der "Scope" eines Singletons - d.h. in welcher Umgebung ist das Singleton tatsächlich "einzeln"? - ist i. d. R. durch die Ablaufumgebung definiert, die nicht mit dem Bereich zusammenfallen muss, in dem das Einzelstück technisch oder fachlich "einzeln" sein soll. Z.B. ist in Java eine einfache static-Variable "einzeln je ClassLoader"; in verteilten (z.B. cluster-fähigen) Systemen oder komplexen nebenläufigen Applikationen wie Application-Servern können dadurch wieder mehrere Instanzen nebeneinander existieren. Analoges gilt für die so genannten App-Domains in .Net. Im Zusammenhang mit Generischen Typen kann die Semantik von static-Variablen nocheinmal komplexer sein (und ist etwa in Java und C# unterschiedlich!).
- Die Testability von Singletons ist schlecht. Das Mocken eines Singleton-Objekts, z.B. eines Netzwerkzugriffs ist aufwändig und in manchen Fällen - z.B. wenn für Testzwecke Fehler erzeugt werden sollen - fast unmöglich.
- In Systemen mit parallelen Abläufen (Threads) muss sichergestellt sein, dass (a) nicht durch parallele Initialisierung kurzfristig mehr als eine Instanz existiert; und (b) dass das Singleton-Objekt später auch die Verwendung in vielen parallelen Abläufen erlaubt, also thread-safe ist.
  - wird in der Regel durch einen Semaphor um die Initialisierung erreicht (z. B. Java: mit synchronized). Aus Effizienzgründen verwendet man dazu in .Net das Double-checked Locking Idiom. In Java ist das wegen tiefliegender Ursachen im Speicherzugriffsmodell nicht möglich.

  - erfordert ein "thread-safe Design" der Singleton-Klasse.

- Die Konfiguration des Singletons ist - zumindest bei Lazy-Initialization (s.u.) - nur über andere Singletons möglich, z.B. Environment-Variablen, aus einem Registry, aus "well-known" Files o.ä.

- Eine Ressource-Deallokation von Ressourcen, die das Singleton verwendet, ist schwierig. So ist z.B. bei einem Singleton für ein Logging-System oft unklar, wann die Log-Datei geschlossen werden soll.

- In einigen objektorientierten Programmiersprachen gibt es keine Möglichkeit, Klassenmethoden zu schreiben.

- In DLLs (dynamic link libraries) lassen sich (zumindest in C++) Singletons nur eingeschränkt verwenden. Da DLLs nicht wie zum Beispiel Libraries zum Programm gelinkt werden, sondern von Haus aus gelinkt sind, wird ein Singleton, das in einer DLL und dem Hauptprogramm verwendet wird, in beiden Modulen ein eigenes Objekt sein. Das kann man (umständlich) vermeiden, indem das Hauptprogramm die eigene Instanz des Singleton an die DLL übergibt (dies ist ein Spezialfall des oben erwähnten "Scope"-Problems).

Wegen der vielen Nachteile wird das Singleton-Muster mitunter schon als Anti-Pattern bewertet. Für Fälle, wo tatsächlich technisch ein passender Bereich für ein Singleton existiert (z.B. wenn nur ein einziges GUI von einem Programm ausgesteuert wird), machen Singletons aber Sinn - insbesondere wenn sie sich auf andere "einmalige Strukturen" wie z.B. eine Abstract Factory beziehen. Trotzdem: Das korrekte Design von Singletons ist schwierig - i. d. R. schwieriger als Designs ohne Singletons.

9.4.2 Beispiel: Lazy Creation

Von Lazy Creation spricht man, wenn das einzige Objekt der Klasse erst erzeugt wird, wenn es benötigt wird.

Die Erstellung des einmalig existierenden Objekts wird folgendermaßen erreicht:

- Der Konstruktor der Singleton-Klasse ist privat. So ist es von außen nicht möglich, ein weiteres Objekt dieser Klasse zu erzeugen.

- Als Ersatz wird eine neue Zugriffsmethode angelegt, die eine Referenz auf das einzige Objekt zurückgeben kann.

- Die Variable, in der das Objekt gespeichert wird, erhält den Modifikator statisch (static). Sie ist außerdem synchronisiert um die Sicherheit bei nebenläufiger Ausführung zu gewährleisten. Eine alternative Implementierung, die ohne Synchronisierung auskommt, ist unter Eager Creation bekannt.

```
/* Implementierung in Java */

public final class Singleton {

    /**
    * Privates Klassenattribut,
```

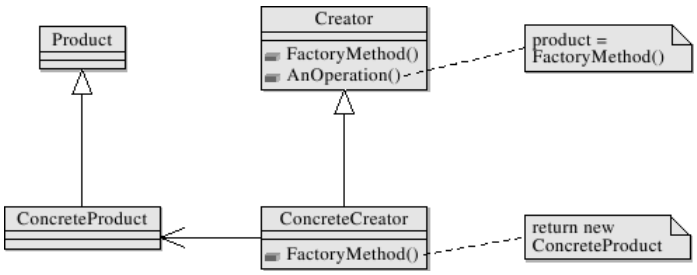


Abbildung 12: UML-Diagramm: Factory Method

```
    * wird beim erstmaligen Gebrauch (nicht beim Laden)
    * der Klasse erzeugt
    */

    private static Singleton instance;

    /** Konstruktor ist privat, darf nicht von außen
    * instanziiert werden.
    */

    private Singleton() {}

    /**
    * Statische Methode "getInstance()" liefert die
    * einzige Instanz der Klasse zurück.
    * Ist synchronisiert und somit thread-sicher.
    */

    public synchronized static Singleton getInstance() {
         if (instance == null) {
              instance = new Singleton();
         }
         return instance;
    }
}
```

9.5 Factory Method

Die Fabrikmethode (englisch Factory Method) ist ein Entwurfsmuster aus dem Bereich der Softwareentwicklung und gehört zur Kategorie der Erzeugungsmuster (Creational Patterns). Das Muster definiert eine Schnittstelle zur Erzeugung eines Objektes, wobei es den Unterklassen überlassen bleibt, von welcher Klasse das zu erzeugende Objekt ist. Es ist eines der sogenannten GoF-Muster (Gang of Four, siehe Viererbande). Dieses Muster wird manchmal auch als virtueller Konstruktor (virtual constructor) bezeichnet.

Der Begriff Fabrikmethode wird in der Praxis auch oft für eine statische Methode verwendet, die ein neues Objekt erzeugt. In diesem Fall ist keine Verwendung von Unterklassen bzw. Polymorphismus vorgesehen.

Die Fabrikmethode findet Anwendung, wenn:

- eine Klasse die von ihr zu erzeugenden Objekte nicht kennen kann bzw. soll oder
- Unterklassen bestimmen sollen, welche Objekte erzeugt werden.

Typische Anwendungsfälle sind Frameworks und Klassenbibliotheken.

- Produkt
  - Basistyp (Klasse oder Schnittstelle) für das zu erzeugende Produkt
- Erzeuger
  - definiert die Fabrikmethode, um ein solches Produkt zu erzeugen (mitunter wird für die Fabrikmethode eine Implementierung vorgegeben, die ein SStandard-Produktërzeugt)
- KonkretesProdukt
  - implementiert die Produkt-Schnittstelle (Subtyp von Produkt)
- KonkreterErzeuger
  - überschreibt die Fabrikmethode um konkrete(re) Produkte zu erzeugen
  - determiniert damit das konkrete Produkt (z.B. indem er den Konstruktor einer konkreten Klasse aufruft)

9.5.1 Vor- & Nachteile

Vorteile:

- Fabrikmethoden entkoppeln ihre Aufrufer von Implementierungen konkreter Produkt-Klassen. Das ist insbesondere wertvoll, wenn Frameworks sich während der Lebenszeit einer Applikation weiterentwickeln - so können zu einem späteren Zeitpunkt Instanzen anderer Klassen erzeugt werden, ohne dass sich die Applikation ändern muss.
- In C++, Java und ähnlichen Sprachen kann eine Fabrikmethode im Gegensatz zu einem Konstruktor einen aussagefähigeren Namen haben, z.B. Color.CreateRGB(...) vs. Color.CreateHSB(...).

Nachteile

- Die Verwendung dieses Erzeugungsmusters läuft auf Unterklassenbildung hinaus.
- Es muss eine eigene Klasse vorhanden sein, die die statische Methode aufnehmen kann (das hat schon zu Forderungen geführt, dass in Java- oder C#-Schnittstellen statische Methoden erlaubt werden sollten).

9.6 Observer

Der Beobachter (englisch Observer) ist ein Entwurfsmuster aus dem Bereich der Softwareentwicklung und gehört zu der Kategorie der Verhaltensmuster (Behavioural Patterns). Es ermöglicht die Weitergabe von Änderungen eines Objekts an abhängige Objekte. Das Muster ist eines der sogenannten GoF-Muster (siehe Viererbande).

Dieses Entwurfsmuster ist auch unter dem Namen publish-subscribe bekannt, frei übersetzt "veröffentlichen und abonnieren".

Akteure:

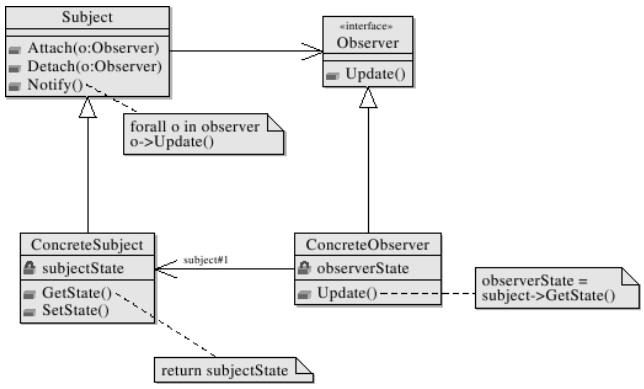


Abbildung 13: UML-Diagramm: Observer

- Subjekt (Beobachtbares Objekt, auch Publisher, also Veröffentlicher, genannt)
  - kennt Liste von Beobachtern, aber keine konkreten Beobachter
  - bietet Schnittstelle zur An- und Abmeldung von Beobachtern
  - bietet Schnittstelle zur Benachrichtigung von Beobachtern über Änderungen
- KonkretesSubjekt (Konkretes, beobachtbares Objekt)
  - speichert relevanten Zustand
  - benachrichtigt alle Beobachter bei Zustandsänderungen über deren Aktualisierungsschnittstelle
  - Schnittstelle zur Erfragung des aktuellen Zustands
- Beobachter (auch Subscriber, also Abonnent, genannt)
  - definiert Aktualisierungsschnittstelle
- KonkreterBeobachter
  - verwaltet Referenz auf ein konkretes Subjekt, dessen Zustand es beobachtet
  - speichert dessen Zustand konsistent
  - implementiert Aktualisierungsschnittstelle unter Verwendung der Abfrageschnittstelle des konkreten Subjekts

9.6.1 Verwendung

Eine oder auch mehrere Komponenten stellen den Status eines Objektes grafisch dar. Sie kennen die gesamte Schnittstelle dieses Objektes. Ändert sich der Status des Objektes, müssen die Komponenten darüber informiert werden. Andererseits soll das Objekt aber von den Komponenten unabhängig bleiben - ihre Schnittstelle also nicht kennen.

Beispiel: Messergebnisse werden gleichzeitig in einem Balkendiagramm, einem Liniendiagramm und einer Tabelle dargestellt. Messwerte ändern sich permanent. Die Komponenten der Diagramme sollen diese Änderungen permanent darstellen, das gemessene Objekt soll dabei aber keine Kenntnis über die Struktur dieser Komponenten besitzen.

Lösung: Das beobachtete Objekt bietet einen Mechanismus, um Beobachter an- und abzumelden und diese über Änderungen zu informieren. Es kennt alle seine Beobachter nur über die (überschaubare) Schnittstelle Beobachter. Es meldet jede Änderung völlig unspezifisch an jeden angemeldeten Beobachter, braucht also die weitere Struktur dieser Komponenten nicht zu kennen.

Die Beobachter implementieren ihrerseits eine (spezifische) Methode, um auf die Änderung zu reagieren. In der Regel werden die für eine Komponente relevanten Teile des Status abgefragt.

Allgemein finden Beobachter Anwendung, wenn

- eine Abstraktion mehrere Aspekte hat, die von einem anderen Aspekt derselben Abstraktion abhängen,
- die Änderung eines Objekts Änderungen an anderen Objekten nach sich zieht oder
- ein Objekt andere Objekte benachrichtigen soll, ohne diese im Detail zu kennen.

9.6.2 Vor- & Nachteile

Vorteile:

- Subjekte und Beobachter können unabhängig variiert werden.
- Subjekt und Benutzer sind auf abstrakte und minimale Art lose gekoppelt. Das beobachtete Objekt braucht keine Kenntnis über die Struktur seiner Beobachter zu besitzen, sondern kennt diese nur über die Beobachter-Schnittstelle.
- Ein abhängiges Objekt erhält die Änderungen automatisch.
- Multicasts werden unterstützt.

Nachteile:

- Änderungen am Objekt führen bei großer Beobachteranzahl zu hohen Änderungskosten. Einerseits informiert das Subjekt jeden Beobachter, auch wenn dieser die Änderungsinformation nicht benötigt. Zusätzlich können die Änderungen weitere Änderungen nach sich ziehen und so einen unerwartet hohen Aufwand haben.
- Ruft ein Beobachter während der Bearbeitung einer gemeldeten Änderung wiederum Änderungsmethoden des Subjektes auf, kann es zu Endlosschleifen kommen.
- Der Mechanismus liefert keine Information darüber, was sich geändert hat. Die daraus resultierende Unabhängigkeit der Komponenten kann sich allerdings auch als Vorteil herausstellen.

- Bei der gerade durchgeführten Observierung eines Objektzustands kann es notwendig sein, einen konsistenten Subjektzustand zu garantieren. Dies kann durch synchrone Aufrufe der Notifizierungsmethode des Beobachters sichergestellt werden. In einem Multithreading System sind evtl. Lockingmechanismen oder Threads mit queuing zur Beobachter-Notifizierung erforderlich.

Java bietet fertige Observer und Observableklassen an, die der Entwickler verwenden kann.

Die Nachteile des Entwurfsmusters entfernt Java, indem es das Beobachterkonzept durch Listener und Events ersetzt. Interessiert sich eine Klasse für eine andere, implementiert sie deren Listener-Interface und meldet sich beim Subjekt an. Ändert sich das Subjekt, informiert es seine Beobachter durch den Aufruf der Listenerschnittstelle, die die Änderung beschreibt.

9.7 Framework

In software development, a framework is a defined support structure in which another software project can be organized and developed. A framework may include support programs, code libraries, a scripting language, or other software to help develop and glue together the different components of a software project.

Frameworks are designed with the intent of facilitating software development, by allowing designers and programmers to spend more time on meeting software requirements rather than dealing with the more tedious low level details of providing a working system. For example, a team using Apache Struts to develop a banking web site can focus on how account withdrawals are going to work rather than how to control navigation between pages in a bug-free manner. However, there are common complaints that using frameworks adds to "code bloat", and that a preponderance of competing and complementary frameworks means that one trades time spent on rote programming and design for time spent on learning frameworks.

Outside of computer applications, a framework can be considered as the processes and technologies used to solve a complex issue. It is the skeleton upon which various objects are integrated for a given solution.

9.8 Strategy Pattern

Die Strategie (engl. Strategy) ist ein Entwurfsmuster aus dem Bereich der Softwareentwicklung und gehört zu der Kategorie der Verhaltensmuster (Behavioural Patterns). Das Muster definiert eine Familie austauschbarer Algorithmen. Es ist eines der sogenannten GoF-Muster.

Akteure:

- Strategie
  - definiert Schnittstelle für alle unterstützten Algorithmen
- KonkreteStrategie
  - implementiert einen Algorithmus
  - Verwendung über Strategieschnittstelle

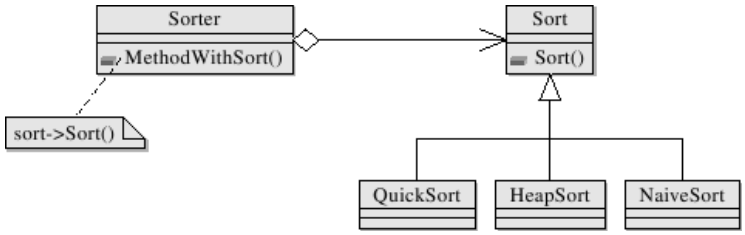


Abbildung 14: UML-Diagramm: Strategy Pattern

  - eventuell Zugriff auf Daten des Kontexts über Kontextschnittstelle
- Kontext
  - Konfiguration mit konkretem Strategieobjekt
  - hält Referenz auf konkretes Strategieobjekt
  - nutzt konkretes Strategieobjekt über Strategieschnittstelle
  - definiert eventuell Schnittstelle für Kontextinformationen

9.8.1 Vor- & Nachteile

Vorteile:

- Es wird eine Familie von Algorithmen definiert.
- Strategien bieten eine Alternative zur Unterklassenbildung.
- Strategien helfen, Mehrfachverzweigungen zu vermeiden, und verbessern dadurch die Wiederverwendung.
- Strategien ermöglichen die Auswahl aus verschiedenen Implementationen und erhöhen dadurch die Flexibilität.

Nachteile:

- Klienten müssen die unterschiedlichen Strategien kennen, um zwischen ihnen auswählen und den Kontext initialisieren zu können.
- Gegenüber der Implementation der Algorithmen im Kontext erzeugen Strategien zusätzlichen Kommunikationsaufwand zwischen Strategie und Kontext.
- Des Weiteren wird die Anzahl von Objekten erhöht.

9.8.2 Beispiel

Als Beispiel kann ein Steuerprogramm dienen, das die Berechnung von Steuersätzen möglichst in Strategie-Objekte auslagern sollte, um einfach länderabhängig konfigurierbar zu sein. Ein anderes Beispiel wäre die Speicherung eines Dokuments oder einer Grafik in verschiedenen Dateiformaten. Auch ein Packer, der verschiedene Kompressionsalgorithmen unterstützt, kann mit Hilfe von Strategie implementiert sein. Bei Java wird das Entwurfsmuster zum Beispiel zur Delegierung des Layouts von AWT-Komponenten an entsprechende LayoutManager (BorderLayout,FlowLayout etc.) verwendet.

9.8.3 Verwendung

Strategie-Objekte werden ähnlich wie Klassenbibliotheken verwendet. Im Gegensatz dazu handelt es sich jedoch nicht um externe Programmteile, die als ein Toolkit genutzt werden können, sondern um integrale Bestandteile des eigentlichen Programms, die deshalb als eigene Objekte definiert wurden, damit sie durch andere Algorithmen ausgetauscht werden können.

Meistens wird eine Strategie durch Klassen umgesetzt, die eine bestimmte Schnittstelle implementieren. In Sprachen wie Smalltalk, in denen auch der Programmcode selbst in Objekten abgelegt werden kann, kann eine Strategie aber auch durch solche Code-Objekte realisiert werden.

Die Verwendung von Strategien bieten sich an, wenn

- viele verwandte Klassen sich nur in ihrem Verhalten unterscheiden,
- unterschiedliche, austauschbare Varianten eines Algorithmus benötigt werden,
- Daten innerhalb eines Algorithmus vor Klienten verborgen werden sollen oder
- verschiedene Verhaltensweisen innerhalb einer Klasse fest integriert (meist über Mehrfachverzweigungen) sind und
  - die verwendeten Algorithmen wiederverwendet werden sollen oder
  - die Klasse flexibler gestaltet werden soll.

Teil IV
Compilerbau

10 Compiler

Ein Compiler (auch Kompilierer oder Übersetzer) ist ein Computerprogramm, das ein in einer Quellsprache geschriebenes Programm - genannt Quellprogramm - in ein semantisch äquivalentes Programm einer Zielsprache (Zielprogramm) umwandelt. Üblicherweise handelt es sich dabei um die Übersetzung eines von einem Programmierer in einer Programmiersprache geschriebenen Quelltextes in Assemblersprache, Bytecode oder Maschinensprache. Das Übersetzen eines Quellprogramms in ein Zielprogramm durch einen Compiler wird auch als Kompilierung bezeichnet.

Die Bezeichnung Compiler (engl. "compile": zusammenstellen/-tragen, "pile" = Haufen/Pfahl/Stapel) ist eigentlich irreführend. Ursprünglich bezeichnete das Wort Compiler Programme, die Unterprogramme zusammenfügen (etwa mit heutigen Linkern vergleichbar). Dies geht an der heutigen Kernaufgabe eines Compilers vorbei.

Zur Steuerung des Übersetzens kann der Quelltext neben den Anweisungen der Programmiersprache zusätzliche spezielle Compiler-Anweisungen enthalten.

Verwandt mit einem Compiler ist ein Interpreter, der ein Programm nicht in die Zielsprache übersetzt, sondern Schritt für Schritt direkt ausführt.

10.1 Aufbau eines Compilers

Moderne Compiler werden in verschiedene Phasen gegliedert, die jeweils verschiedene Teilaufgaben des Compilers übernehmen. Einige dieser Phasen können als eigenständige Programme realisiert werden (s. Precompiler, Präprozessor). Sie werden sequentiell ausgeführt. Im Wesentlichen lassen sich zwei Phasen unterscheiden: das Frontend (auch Analysephase), das den Quelltext analysiert und daraus einen attributierten Syntaxbaum erzeugt, sowie das Backend (auch Synthesephase), das daraus das Zielprogramm erzeugt.

10.1.1 Frontend (auch Analysephase)

Im Frontend wird der Code analysiert, strukturiert und auf Fehler geprüft. Es ist auch selbst wieder in Phasen gegliedert:

Lexikalische Analyse
Die lexikalische Analyse zerteilt den eingelesenen Quelltext in zusammengehörende Token verschiedener Klassen, z. B. Schlüsselwörter, Bezeichner, Zahlen und Operatoren. Dieser Teil des Compilers heißt Scanner oder Lexer.

Ein Scanner benutzt gelegentlich einen separaten Screener, um Whitespace (Leerraum, also Leerzeichen, Tabulatorzeichen, Zeilenenden, usw.) und Kommentare zu überspringen.

Syntaktische Analyse
Die syntaktische Analyse überprüft, ob der eingelesene Quellcode ein korrektes

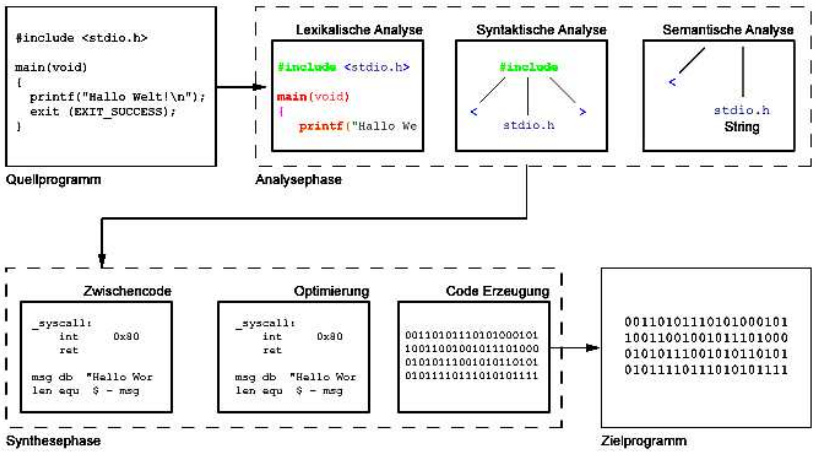


Abbildung 15: Phasen eines Compilers

Programm der zu übersetzenden Quellsprache ist, d. h. der Syntax (Grammatik) der Quellsprache entspricht. Dabei wird die Eingabe in einen Syntaxbaum umgewandelt. Dieser Teil wird auch als Parser bezeichnet.

Semantische Analyse
Die semantische Analyse überprüft die statische Semantik, also über die syntaktische Analyse hinausgehende Bedingungen an das Programm. Zum Beispiel muss eine Variable in der Regel deklariert worden sein, bevor sie verwendet wird, und Zuweisungen müssen mit kompatiblen (verträglichen) Datentypen erfolgen. Dies kann mit Hilfe von Attributgrammatiken realisiert werden. Dabei werden die Knoten des vom Parser generierten Syntaxbaums mit Attributen versehen, die Informationen enthalten. So kann zum Beispiel eine Liste aller deklarierten Variablen erstellt werden. Die Ausgabe der semantischen Analyse nennt man dann dekorierter oder attributierter Syntaxbaum.

10.1.2 Backend (auch Synthesephase)

Das Backend erzeugt aus dem vom Frontend erstellten attributierten Syntaxbaum den Programmcode der Zielsprache.

Zwischencodeerzeugung
Viele moderne Compiler erzeugen aus dem Syntaxbaum einen Zwischencode, der schon relativ maschinennah sein kann und führen auf diesem Zwischencode z. B. Programmoptimierungen durch. Das bietet sich besonders bei Compilern an, die mehrere Quellsprachen oder verschiedene Zielplattformen unterstützen. Hier kann der Zwischencode auch ein Austauschformat sein.

Nicht direkt Code für den Prozessor des Laufzeitsystems zu erzeugen, sondern zunächst nur Zwischencode, für einen idealen (auch virtuell genannten)

Prozessor (also einer Hardware, die oft nur durch Software simuliert existiert), kann vorteilhaft sein:

- z. B. weil man Portabilität anstrebt (siehe Java VM), oder
- weil der Übersetzer durch diese Aufteilung des Übersetzungsprozesses einfacher wird (siehe p-Code), oder
- weil man allgemeine Optimierungen (effizienzsteigernde Codetransformationen) bereits auf dem Zwischencode vornehmen kann, oder
- weil der Zielprozessor noch nicht bequem genug zu programmieren ist, z. B. weil man gerne Fließkommabefehle hätte, der Prozessor aber keine FPU hat - ein weiterer Übersetzungsschritt fügt dann Code ein, der diese Befehle mit den vorhandenen Ganzzahlbefehlen simuliert

Programmoptimierung
Der Zwischencode ist Basis vieler Programmoptimierungen (siehe 10.2).

Codegenerierung
Bei der Codegenerierung wird der Programmcode der Zielsprache entweder direkt aus dem Syntaxbaum oder aus dem Zwischencode erzeugt. Falls die Zielsprache eine Maschinensprache ist, kann das Ergebnis direkt ein ausführbares Programm sein oder eine sogenannte Objektdatei, die durch das Linken mit der Laufzeitbibliothek und evtl. weiteren Objektdateien zu einer Bibliothek oder einem ausführbaren Programm führt.

10.2 Programmoptimierung

Üblicherweise bietet ein Compiler Optionen für verschiedene Optimierungen mit dem Ziel, die Laufzeit des Zielprogramms zu verbessern oder dessen Speicherplatzbedarf zu minimieren.

Die Optimierungen erfolgen teilweise in Abhängigkeit von den Eigenschaften der Hardware, z. B. wie viele und welche Register der Prozessor des Computers zur Verfügung stellt.

Optimierungen optimieren oft nur bestimmte Aspekte eines Programms. Es ist möglich, dass ein Programm nach einer Optimierung langsamer ausgeführt wird, als das ohne sie der Fall gewesen wäre. Dies kann zum Beispiel eintreten, wenn eine Optimierung für ein Programmkonstrukt längereren Code erzeugt, der zwar an sich schneller ausgeführt werden würde, aber mehr Zeit benötigt, um erst einmal in den Cache geladen zu werden. Er ist damit erst bei häufigerer Benutzung vorteilhaft.

Einige Optimierungen führen dazu, dass der Compiler Programmkonstrukte aus der Quellsprache in Zielsprachenkonstrukte übersetzt, für die es gar keine direkten Entsprechungen in der Quellsprache gibt. Ein Nachteil solcher Optimierungen ist, dass es damit kaum noch möglich ist, den Programmablauf mit einem interaktiven Debugger zu verfolgen.

Optimierungen können sehr aufwendig sein. Vielfach muss in modernen Compilern daher abgewogen werden, ob es sich lohnt, einen Programmteil zu optimieren. Es können Tests und Anwendungsszenarien durchgespielt werden (s. Profiler), um herauszufinden, wo sich komplexe Optimierungen lohnen.

Im folgenden werden einige Optimierungsmöglichkeiten eines Compilers betrachtet. Das größte Optimierungspotenzial besteht allerdings oft in der Veränderung des Quellprogramms selbst, zum Beispiel darin, einen Algorithmus durch einen effizienteren zu ersetzen. Dieser Vorgang kann meistens nicht automatisiert werden, sondern muss durch den Programmierer erfolgen. Einfachere Optimierungen können dagegen an den Compiler delegiert werden, um den Quelltext lesbarer zu halten.

10.2.1 Einsparung von Maschinenbefehlen

In vielen höheren Programmiersprachen benötigt man beispielsweise eine Hilfsvariable, um den Inhalt zweier Variablen zu vertauschen:

Höhere Programmiersprache	Maschinenbefehle ohne Optimierung	Maschinenbefehle mit Optimierung
t = a	a → Register 1 Register 1 → t	a → Register 1
a = b	b → Register 1 Register 1 → a	b → Register 2
b = t	t → Register 1 Register 1 → b	Register 1 → b Register 2 → a

Mit der Optimierung werden statt 6 nur noch 4 Assemblerbefehle benötigt, außerdem wird der Speicherplatz für die Hilfsvariable t nicht gebraucht. D. h. diese Vertauschung wird schneller ausgeführt und benötigt weniger Hauptspeicher. Dies gilt jedoch nur, wenn ausreichend Register im Prozessor zur Verfügung stehen. Die Speicherung von Daten in Registern statt im Hauptspeicher ist eine häufig angewendete Möglichkeit der Optimierung.

10.2.2 Statische Formelauswertung zur Übersetzungszeit

Die Berechnung des Kreisumfangs mittels

```
pi = 3.1415
u  = 2 * pi * r
```

kann ein Compiler bereits zum Übersetzungszeitpunkt zu $u = 6.2830 * r$ auswerten. Dies spart die Multiplikation $2 * pi$ zur Laufzeit des erzeugten Programms. Diese Vorgehensweise wird als Konstantenfaltung (engl. "constant folding") bezeichnet. (Compiler für die Sprache Ada müssen diese Compile-Zeit-Berechnungen sogar in beliebiger Genauigkeit durchführen.)

10.2.3 Optimierung von Schleifen

Insbesondere Schleifen versucht man zu optimieren, indem man z. B.

- möglichst viele Variablen in Registern hält (normalerweise mindestens die Schleifenvariable).
- statt eines Index, mit dem auf Elemente eines Feldes (englisch array) zugegriffen wird, Zeiger auf die Elemente verwendet. Dadurch wird der Aufwand beim Zugriff auf Feldelemente geringer.
- Berechnungen innerhalb der Schleife, die in jedem Durchlauf dasselbe Ergebnis liefern, nur einmal vor der Schleife ausführt.

- zwei Schleifen, die über denselben Wertebereich gehen, zu einer Schleife zusammenfasst. Damit fällt der Verwaltungsaufwand für die Schleife nur einmal an.
- die Schleife teilweise oder (bei Schleifen mit konstanter, niedriger Durchlaufzahl) komplett auflöst (englisch loop unrolling), sodass die Anweisungen innerhalb der Schleife mehrfach direkt hintereinander ausgeführt werden, ohne dass jedesmal nach den Anweisungen eine Prüfung der Schleifenbedingung und ein Sprung zum Schleifenbeginn erfolgen.
- die Schleife (vor allem bei Zählschleifen mit for) umgedreht wird, da beim Herunterzählen auf 0 effiziente Sprungbefehle (Jump-Not-Zero) benutzt werden können.
- die Schleife umformt, damit die Überprüfung der Abbruchbedingung am Ende der Schleife durchgeführt wird (Schleifen mit Anfangsüberprüfung haben stets eine bedingte und eine unbedingte Sprunganweisung, während Schleifen mit Endüberprüfung nur eine bedingte Sprunganweisung haben).
- wenn eine Schleife (nach einigen Optimierungen) einen leeren Rumpf besitzt, sie ganz entfernen kann. Dies kann allerdings dazu führen, dass Warteschleifen, die ein Programm absichtlich verlangsamen sollen, entfernt werden. Allerdings sollten für diesen Zweck, soweit möglich, sowieso möglichst Funktionen des Betriebssystems benutzt werden.

10.2.4 Peephole Optimizations

Peephole optimizations are usually performed late in the compilation process after machine code has been generated. This form of optimization examines a few adjacent instructions (like looking through a peephole at the code) to see whether they can be replaced by a single instruction or a shorter sequence of instructions. For instance, a multiplication of a value by two might be more efficiently executed by shifting the value left or by adding the value to itself (this example is also an instance of strength reduction).

11 Backus-Naur-Form

Die Backus-Naur-Form oder Backus-Normalform, kurz BNF, ist eine kompakte formale Metasprache zur Darstellung kontextfreier Grammatiken (Typ-2-Grammatiken in der Chomsky-Hierarchie). Hierzu zählt die Syntax gängiger höherer Programmiersprachen. Sie wird auch für die Notation von Befehlssätzen und Kommunikationsprotokollen verwendet.

Ursprünglich war sie nach John Backus benannt, später wurde sie (auf Anregung von Donald E. Knuth) auch nach Peter Naur benannt. Beide waren Informatikpioniere, die sich mit der Erstellung der Algol-60-Regeln und insbesondere mit der Kunst des Compilerbaus beschäftigten. Durch die Backus-Naur-Form im Algol 60 Report wurde es erstmals möglich, die Syntax einer Programmiersprache formal exakt, also ohne die Ungenauigkeiten natürlicher Sprachen, darzustellen.

Es gibt viele Varianten der Backus-Naur-Form. Die erweiterte Backus-Naur-Form (EBNF) ist eine gebräuchliche Variante, die unter anderem eine kompakte Notation von sich wiederholenden Elementen erlaubt.

11.1 Grundlagen

Ein Programm besteht zunächst aus sichtbaren, also auf der Tastatur vorhandenen, Zeichen. Daneben treten noch Leerzeichen und Zeilentrenner auf. Die sichtbaren Zeichen werden zu den Terminalsymbolen (engl. terminals) gerechnet.

In computer science, terminal and nonterminal symbols are those symbols that are used to construct production rules in a formal grammar. Whereas terminal symbols form the parts of strings generated by the grammar, nonterminal symbols map to the names of productions in the grammar, and generate strings by substitution either of other nonterminals or of terminals (or some combination of these).

For instance, to represent a (possibly signed) integer, the following (expressed in a variant of BNF) may be described:

BNF verwendet so genannte Ableitungsregeln (Produktionen), in denen Nichtterminalsymbole (engl. nonterminals) definiert werden. Dabei dient das Zeichen | (vertikaler Strich) als Alternative, die Zeichenfolge ::= wird zur Definition verwendet und die Nichtterminalsymbole, die auch syntaktische Variablen genannt werden, werden mit spitzen Klammern <...> umschlossen:

Alternative:

```
<Ziffer außer Null> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Eine Ziffer außer Null ist also entweder eine 1 oder eine 2 oder eine 3 usw. Es lassen sich auch Terminalfolgen definieren, also eine Sequenz. Als Elemente dürfen Terminalsymbole und Nichtterminalsymbole auftreten:

Sequenz:

```
<Ziffer>              ::= 0 | <Ziffer außer Null>
<Zweistellige Zahl>   ::= <Ziffer außer Null> <Ziffer>
<Zehn bis Neunzehn>   ::= 1 <Ziffer>
<Zweiundvierzig>      ::= 42
```

Eine Ziffer ist also eine 0 oder eine Ziffer außer Null. Eine zweistellige Zahl ist eine Ziffer außer Null gefolgt von einer Ziffer. Zweiundvierzig ist eine 4 gefolgt von einer 2. Wiederholungen muss man in BNF über Rekursionen definieren. Eine Ableitungsregel kann dazu auf der rechten Seite das Symbol auf der linken Seite enthalten, etwa:

```
<Ziffernfolge> ::= <Ziffer> | <Ziffer> <Ziffernfolge>
```

Lies: Eine Ziffernfolge ist eine Ziffer oder eine Ziffer gefolgt von einer Ziffernfolge. Eine Ziffernfolge passt also zu den Symbolfolgen 0,1, 2, 10,9870,8970635 usw., jedoch auch zu 00, 000, ... Eine positive Zahl darf nicht mit 0 beginnen. Dies leistet die folgende Regel:

```
<Positive Zahl> ::= <Ziffer außer Null> | <Ziffer außer Null> <Ziffernfolge>
```

Option

```
<Zahl> ::= [ - ] <Positive Zahl>
```

Das Minuszeichen ist optional. Die Definition ist äquivalent zu

```
<Zahl> ::= <Positive Zahl> | - <Positive Zahl>
```

Eine Zahl ist eine positive Zahl, oder ein Minuszeichen, gefolgt von einer positiven Zahl.

11.2 BNF und Programmiersprachen

Um die Syntax von Programmiersprachen wie ALGOL, Pascal, Java in BNF darzustellen, muss man noch die Schlüsselwörter (IF, SWITCH) zu den Terminalsymbolen rechnen. In einem Compiler werden sie in einer Vorphase, der lexikalischen Analyse, erkannt und als besondere Zeichen weitergegeben. Kommentare werden von der lexikalischen Analyse erkannt (und oft entfernt), manchmal auch weitere Elemente wie Gleitkommazahlen, Bezeichner und Zeichenketten.

Damit lässt sich dann die gesamte Syntax z.B. eines PASCAL-Programms in BNF darstellen:

```
<Programm>               ::= 'PROGRAM' <Bezeichner> 'BEGIN' <Satzfolge> 'END' .
<Bezeichner>             ::= <Buchstabe> <Restbezeichner>
<Restbezeichner>         ::= | <Buchstabe oder Ziffer> <Restbezeichner>
<Buchstabe oder Ziffer>  ::= <Buchstabe> | <Ziffer>
<Buchstabe>              ::= A | B | C | D | ... | Z | a | b | ... | z   *)
<Ziffer>                 ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Satzfolge>              ::= ...
...

*) gekürzt
```

Eine Syntaxanalyse besteht aus der Rückführung eines Programmtexts auf das Nichtterminalsymbol <Programm>. Ein Programm muss also mit dem Wort PROGRAM beginnen, auf das ein Bezeichner folgt. Bezeichner beginnen mit einem Buchstaben, gefolgt von beliebig vielen Buchstaben oder Ziffern.

Die Rückführung auf <Programm> gelingt bei

```
PROGRAM Ggt BEGIN ... END.

PROGRAM DiesisteinlangerBezeichnermit123 BEGIN ... END .
```

nicht jedoch bei

```
Ggt BEGIN ... END.          (beginnt nicht mit PROGRAM)

PROGRAM 123 BEGIN ... END. (123 ist kein Bezeichner, Bezeichner
                             müssen mit einem Buchstaben beginnen)
```

11.3 Beispiel

Hier eine BNF für eine deutsche Postanschrift:

```
<Post-Anschrift>  ::= <Personenteil> <Straße> <Stadt>
<Personenteil>    ::= [ <Titel> ] <Namensteil> <EOL>
<Vornamenteil>    ::= <Vorname> | <Initial> .
<Namensteil>      ::= <Vornamenteil> <Nachname> | <Vornamenteil> <Namensteil>
<Straße>          ::= <Straßenname> <Hausnummer> <EOL>
<Stadt>           ::= <Postleitzahl> <Stadtname> <EOL>

(Definition von Straßenname, Hausnummer etc. fehlen)
```

Die Ausformulierung lautet:

- Eine Postanschrift besteht aus einem Personenteil, gefolgt von einer Straße, gefolgt von der Stadt.
- Der Personenteil besteht aus einem optionalen Titel und einem Namensteil, gefolgt von einem Zeilenende.
- Der Vornamenteil besteht aus einem Vornamen oder einem Initial, auf den dann ein Punkt folgt.
- Der Namensteil besteht aus einem Vornamen-Teil, einem Nachname oder aus einem Vornamen-Teil und wiederum aus einem Namensteil. (Diese Regel zeigt die Benutzung von Rekursion in BNFs und stellt den Fall dar, dass eine Person mehrere Vornamen und/oder Initialen besitzt.)
- Eine Straße besteht aus einem Straßenname, gefolgt von einer Hausnummer, gefolgt von einem Zeilenende.
- Eine Stadt besteht aus einer Postleitzahl, gefolgt von einem Stadtname, gefolgt von einem Zeilenende.

Man beachte, dass einiges (wie die Postleitzahl oder Hausnummer) nicht weiter spezifiziert ist. Es wird angenommen, dass diese lexikalischen Details vom Kontext abhängen oder anderweitig spezifiziert sind.

Dieses Beispiel ist keine reine Form aus dem "ALGOL 60 report". Die eckigen Klammern "[ ]" stellen eine Option dar. Sie wurden einige Jahre später in der Definition von IBMs PL/I eingeführt, sind aber allgemein anerkannt.

12 Erweiterte Backus-Naur-Form

Die Erweiterte Backus-Naur-Form, kurz EBNF, ist eine Erweiterung der Backus-Naur-Form (BNF), die ursprünglich von Niklaus Wirth zur Darstellung der Syntax der Programmiersprache Pascal eingeführt wurde. Sie ist eine formale Metasyntax (Metasprache), die benutzt wird, um kontextfreie Grammatiken darzustellen.

Die EBNF ist von der ISO als ISO/IEC 14977:1996(E) standardisiert. Gelegentlich werden auch andere erweiterte Varianten der BNF nicht ganz korrekt ebenfalls als EBNF bezeichnet.

12.1 Grundlagen

Ein Text, etwa Quelltext eines Computerprogramms, besteht zunächst aus Terminalsymbolen, das heißt, aus sichtbaren Zeichen Buchstaben, Ziffern, Satzzeichen, Leerzeichen, etc.

Die EBNF definiert Produktionsregeln, in denen Symbolfolgen jeweils einem Nichtterminalsymbol zugeordnet werden, etwa

```
ZifferAußerNull   = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
Ziffer            = "0" | ZifferAußerNull ;
```

In dieser Produktionsregel wird das Nichtterminalsymbol Ziffer definiert, das stets auf der linken Seite steht. Der vertikale Strich stellt eine Alternative dar, die Terminalsymbole werden in Anführungszeichen eingeschlossen und mit einem Semikolon als Endezeichen abgeschlossen. Eine Ziffer ist also eine 0 oder eine ZifferAußerNull, die wiederum 1 oder 2 oder 3 usw. bis 9 sein kann.

Eine Produktionsregel kann auch eine Folge von Terminal- oder Nichtterminalsymbolen enthalten, etwa:

```
Zwoelf                       = "1" "2" ;
Zweihundertundeins           = "2" "0" "1" ;
Dreihundertzwölf             = "3" Zwoelf ;
ZwoelfTausendzweihunderteins = Zwoelf Zweihundertundeins ;
```

Ausdrücke, die ausgelassen oder wiederholt werden dürfen, können mit geschweiften Klammern dargestellt ... werden:

```
NatuerlicheZahl = ZifferAußerNull { Ziffer } ;
```

Hier passen die Texte 1, 2, ...,10,...,12345,... . Zu beachten ist, dass alles, was innerhalb der geschweiften Klammern steht, beliebig oft, jedoch auch keinmal vorkommen darf.

Eine Option kann durch eckige Klammen [...] dargestellt werden:

```
GanzeZahl = "0" | [ "-" ] NatuerlicheZahl ;
```

Eine ganze Zahl ist also die Null (0) oder eine natürliche Zahl, der optional ein Minuszeichen vorangestellt werden kann. Hier passen also alle ganzen Zahlen wie 0, -3, 1234 etc.

12.2 Erweiterungen nach ISO

Nach dem Standard ISO 14977 ist es möglich, Nichtterminalsymbole auch aus mehr als einem Wort bestehen zu lassen, indem die Einzelteile durch ein Komma verbunden werden.

```
Ganze Zahl = "0" | [ "-" ] , Natuerliche Zahl ;
```

Außerdem ist die Möglichkeit vorgesehen, eine definierbare Anzahl an Wiederholungen zu erlauben.

```
LeerzeichenAlsTab = 4 * " " , "Yes" ;
```

Hier wird vor der Zeichenfolge “Yes” viermal das “ ” Zeichen erwartet.

12.3 Motivation zur Erweiterung der BNF

Die BNF benötigt teilweise umständliche Konstrukte, um optionale Elemente, also Elemente, die ausgelassen werden dürfen, sowie sich wiederholende Elemente darzustellen. In der Spezifikation von PL/1 wurden bereits eckige Klammern “[...]” für Optionen verwendet. Niklaus Wirth hat in der Definition der Sprache Pascal zusätzlich geschweifte Klammern “{...}” für Wiederholungen in die BNF eingeführt und nannte dies extended BNF (erweiterte BNF).

Alle Formulierungen in einer EBNF-Syntax lassen sich auch in BNF ausdrücken. Die EBNF wurde von Wirth aus Gründen der besseren Lesbarkeit und kompakteren Schreibweise geschaffen.

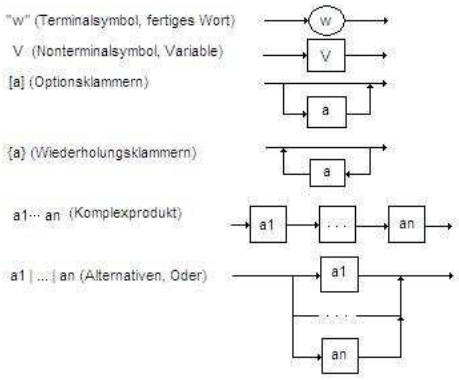


Abbildung 16: Übersetzung einer EBNF in ein Syntaxdiagramm

12.4 Syntaxdiagramm

Ein Syntaxdiagramm wird in der Theoretischen Informatik benutzt, um die Syntax einer Regelmenge graphisch darzustellen. Insbesondere können damit Formale Sprachen bis zur Klasse der kontextfreien Sprachen und damit aufgrund der Teilmengeneigenschaft auch die Syntax von Programmiersprachen in einem Syntaxdiagramm dargestellt werden.

Jede Erweiterte Backus-Naur-Form kann mit Hilfe der Übersetzung in Abbildung 16 eins zu eins in ein Syntaxdiagramm gewandelt werden.

13 Endlicher Automat

Ein endlicher Automat (EA, auch Zustandsmaschine, engl. finite state machine - FSM) ist ein Modell des Verhaltens, bestehend aus Zuständen, Zustandsübergängen und Aktionen. Ein Automat heißt endlich, wenn seine Zustandsmenge (später S genannt) endlich ist, er also eine endliche Menge von Zuständen besitzt, die er einnehmen kann. Ein EA ist ein Speziallfall aus der Menge der Automaten. Ein Zustand speichert die Information über die Vergangenheit, d.h. er reflektiert die Änderungen der Eingabe seit dem Systemstart bis zum aktuellen Zeitpunkt. Ein Zustandsübergang zeigt eine Änderung des Zustandes des EA und wird durch logische Bedingungen beschrieben, die erfüllt sein müssen, um den Übergang zu ermöglichen. Eine Aktion ist die Ausgabe des EA, die in einer bestimmten Situation erfolgt. Es gibt vier Typen von Aktionen:

- **Eingangsaktion**: Ausgabe wird beim Eintreten in einen Zustand generiert
- **Ausgangsaktion**: Ausgabe wird beim Verlassen eines Zustandes generiert
- **Eingabeaktion**: Ausgabe wird abhängig vom aktuellen Zustand und Eingabe generiert
- **Übergangsaktion**: Ausgabe wird abhängig von einem Zustandsübergang generiert

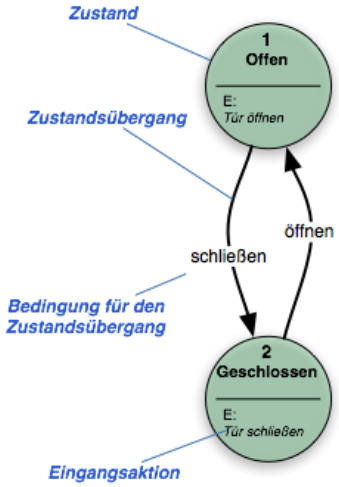


Abbildung 17: Beispiel eines EA

Ein EA kann als Zustandsübergangsdiagramm wie in Abbildung 17 dargestellt werden. Zusätzlich werden mehrere Typen von Übergangstabellen (bzw. Zustandsübergangstabellen) benutzt. Die folgende Tabelle zeigt eine sehr verbreitete Form von Übergangstabellen: die Kombination aus dem aktuellen Zustand (B) und Eingabe (Y) führt zum nächsten Zustand (C). Die komplette Information über die möglichen Aktionen wird mit Hilfe von Fußnoten angegeben. Eine Definition des EA, die auch die volle Ausgabeinformation beinhaltet, ist mit Zustandstabellen möglich, die für jeden Zustand einzeln definiert werden (siehe auch virtueller EA).

Übergangstabelle

	Zustand A	Zustand B	Zustand C
Bedingung X	...	...	...
Bedingung Y	...	Zustand C	...
Bedingung Z	...	...	...

Die Definition des EA wurde ursprünglich in der Automatentheorie eingeführt und später in der Computertechnik übernommen.

Zustandsmaschinen werden hauptsächlich in der Entwicklung digitaler Schaltungen, Modellierung des Applikationsverhaltens (Steuerungen), generell in der Softwaretechnik sowie Wort- und Spracherkennung benutzt.

13.1 Klassifizierung

Generell werden zwei Gruppen von EA unterschieden: Akzeptoren und Transduktoren.

13.1.1 Akzeptoren

Sie akzeptieren und erkennen die Eingabe und signalisieren durch ihren Zustand das Ergebnis nach außen. In der Regel werden Symbole (Buchstaben) als Einga-

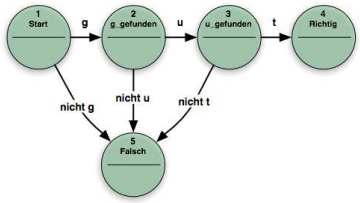


Abbildung 18: EA vom Typ Akzeptor: erkennt das Wort "gut"

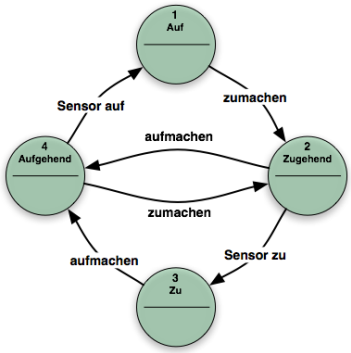


Abbildung 19: EA vom Typ Transduktor: Moore-Modell

be benutzt. Das Beispiel in der Abbildung 18 zeigt einen EA, der das Wort "gut" akzeptiert. Akzeptoren werden vorwiegend in der Wort- und Spracherkennung eingesetzt.

13.1.2 Transduktoren (Transducer)

Transduktoren generieren Ausgaben in Abhängigkeit von Zustand und Eingabe mit Hilfe von Aktionen. Sie werden vorwiegend für Steuerungsaufgaben eingesetzt, wobei grundsätzlich zwei Typen unterschieden werden:

- **Moore-Automat**

  Im Moore-Modell werden nur Eingangsaktionen benutzt, d.h. die Ausgabe (Γ) hängt nur vom Zustand (S) ab ($S \rightarrow \Gamma$). Das Verhalten eines Moore-Automaten ist dadurch, verglichen mit dem Mealy-Modell, einfacher und leichter zu verstehen. Das Beispiel in Abbildung 19 zeigt einen Moore-Automaten, der eine Aufzugtür steuert. Die Zustandsmaschine kennt zwei Befehle, "aufmachen" und "zumachen", die von einem Benutzer eingegeben werden können. Die Eingangsaktion (E:) im Zustand "Aufgehend" startet einen Motor, der die Tür öffnet, und die Eingangsaktion im Zustand "Zugehend" startet den Motor in entgegengesetzter Richtung. In den Zuständen "Auf" und "Zu" werden keine Aktionen durchgeführt. Sie signalisieren nur die Situation nach außen (bzw. zu anderen EA).

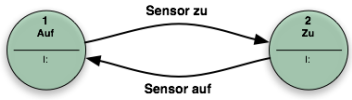


Abbildung 20: EA vom Typ Transduktor: Mealy-Modell

- **Mealy-Automat**

  Im Mealy-Modell werden Eingabeaktionen benutzt, d.h. die Ausgabe (Γ) hängt von Zustand (S) und Eingabe (Σ) ab (S x $\Sigma \rightarrow \Gamma$). Der Einsatz von Mealy-Automaten führt oft zu einer Verringerung der Anzahl zu berücksichtigender Zustände. Die Funktion des EA ist dadurch komplexer und oft schwieriger zu verstehen. Das Beispiel in der Abbildung 20 zeigt einen Mealy-EA, der das gleiche Verhalten wie der EA im Moore-Beispiel aufweist. Es gibt zwei Eingabeaktionen (I:): "starte den Motor, um die Tür zu schließen, wenn die Eingabe 'zumachen' erfolgt" und "starte den Motor in entgegengesetzter Richtung, um die Tür zu öffnen, wenn die Eingabe 'aufmachen' erfolgt".

Moore- und Mealy-Automaten sind gleichwertig. Der eine kann in den jeweils anderen überführt werden. In der Praxis werden meistens Mischmodelle benutzt.

Eine weitere Klassifizierung der EA wird durch die Unterscheidung zwischen deterministischen (DEA) und nicht-deterministischen (NEA) Automaten gemacht. In den deterministischen Automaten existiert für jeden Zustand genau ein Übergang für jede mögliche Eingabe. Bei den nicht-deterministischen Automaten kann es keinen oder auch mehr als einen Übergang für die mögliche Eingabe geben.

Ein EA, der nur aus einem Zustand besteht wird als kombinatorischer EA bezeichnet. Er benutzt nur Eingabeaktionen.

13.2 Die Logik des EA

Der nächste Zustand und die Ausgabe des EA ist eine Funktion der Eingabe und des aktuellen Zustandes. Abbildung 21 zeigt den Ablauf der Logik.

13.3 Das mathematische Modell

Es gibt unterschiedliche Definitionen, je nach Typ des EA. Ein Akzeptor (oder auch deterministischer Automat) ist ein 5-Tupel (Σ, S, s0, δ, F), wobei:

- Σ ist das Eingabealphabet (eine endliche nicht leere Menge von Symbolen),
- S ist eine endliche nicht leere Menge von Zuständen,
- s0 ist der Anfangszustand und ein Element aus S,
- δ ist die Zustandsübergangsfunktion: δ: S x $\Sigma \rightarrow$ S,
- F ist die Menge von Endzuständen und eine (möglicherweise leere) Untermenge von S.

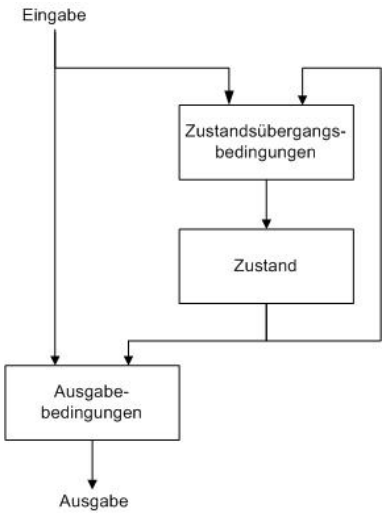


Abbildung 21: Die Logik eines EA

Ein Transduktor ist ein 6-Tupel (Σ, Γ, S, s0, δ, ω), wobei:

- Σ ist das Eingabealphabet (eine endliche nicht leere Menge von Symbolen),
- Γ ist das Ausgabealphabet (eine endliche nicht leere Menge von Symbolen),
- S ist eine endliche nicht leere Menge von Zuständen,
- s0 ist der Anfangszustand und ein Element aus S,
- δ ist die Zustandsübergangsfunktion: δ: S x $\Sigma \rightarrow$ S,
- ω ist die Ausgabefunktion.

Falls die Ausgabefunktion eine Funktion von Zustand und Eingabealphabet ist (ω: S x $\Sigma \rightarrow \Gamma$), dann handelt es sich um ein Mealy-Modell. Falls die Ausgabefunktion nur vom Zustand abhängt (ω: S $\rightarrow \Gamma$), dann ist es ein Moore-Automat.

13.4 Optimierung & Implementierung

Ein EA wird optimiert, indem die Zustandsmaschine mit der geringsten Anzahl von Zuständen gefunden wird, die die gleiche Funktion erfüllt. Dieses Problem kann zum Beispiel mit Hilfe von Färbungsalgorithmen gelöst werden.

In digitalen Schaltungen werden EA mit Hilfe von PLCs, logischen Gattern, Flip-Flops oder Relais gebaut. Eine Hardwareimplementation benötigt normalerweise ein Register, um die Zustandsvariable zu speichern, eine Logikeinheit, die die Zustandsübergänge bestimmt und eine zweite Logikeinheit, die für die Ausgabe verantwortlich ist.

In der Softwareentwicklung werden meist folgende Konzepte verwendet, um Applikationen mit Hilfe von Zustandsmaschinen zu modellieren bzw. implementieren:

- ereignisgesteuerter EA
- virtueller EA

13.5 Darstellung endlicher Automaten

Die allgemeinen Regeln für das Zeichnen eines Zustandsübergangsdiagramms sind wie folgt:

- Kreise stellen Zustände dar. Im Kreis steht der Name des Zustands.
- Pfeile zwischen Zuständen stellen die Transitionen dar. Auf jedem Pfeil steht, welche Bedingungen den Übergang ermöglichen.

Es gibt Softwarewerkzeuge, um die Graphen der EA in ansprechender Form zu generieren. Beispielsweise kann man die Pakete Xy-pic (engl.) oder GasTeX (engl.) für LaTeX verwenden. Der StateWORKS Editor (engl.) ermöglicht sowohl das Zeichnen von einzelnen EA sowie von Systemen von EA. Ein weiterer Editor ist AutoEdit. Eine Komplettlösung zum designen, testen und bereitstellen von EA bietet steed.net incl. voller Integration in VisualStudio 2005. steed.net geht sogar noch weiter und bietet eine Lösung zum Verteilen ganzer EA-Systeme im Netzwerk.

Literatur

[1] Wikipedia

[2] E. W. Dijkstra: A note on two problems in connexion with graphs. In: Numerische Mathematik. 1 (1959), S. 269271

[3] Th.H.Cormen/C.E.Leiserson/R.Rivest/C.Stein: Algorithmen - Eine Einführung. Oldenbourg Verlag 2004. ISBN 3-486-27515-1

[4] Richard M. Karp: Reducibility Among Combinatorial Problems. In Complexity of Computer Computations, Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y.. New York: Plenum, p.85-103. 1972.

[5] Michael R. Garey und David S. Johnson: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco, 1978. ISBN 0716710455

[6] Stephen A. Cook: The Complexity of Theorem Proving Procedures. In Annual ACM Symposium on Theory of Computing (STOC), pp 151–158, 1971.

[7] Thomas Ottmann, Peter Widmayer: Algorithmen und Datenstrukturen. Spektrum Akademischer Verlag, Heidelberg 2002 (4. Aufl.), ISBN 3-8274-1029-0

[8] S. Mertens: A complete anytime algorithm for balanced number partitioning, oai:arXiv.org:cs/9903011

[9] Leena Suhl und Taieb Mellouli: "Optimierungssysteme: Modelle, Verfahren, Software, Anwendungen", Springer, Berlin, 2006, ISBN 3-540-26119-2

[10] J. Kallrath: "Gemischt-Ganzzahlige Optimierung: Modellierung in der Praxis", Vieweg, Wiesbaden, 2002

[11] Richard Kipp Martin: Large Scale Linear and Integer Optimization - A Unified Approach, Kluwer Academic Publishers, 1999

[12] R.J. Dakin: A tree-search algorithm for mixed integer programming problems. In: The Computer Journal, Volume 8, S. 250-255, 1965

[13] Ralph Gomory: Early Integer Programming. In: Operations Research, Volume 50, Nummer 1, S. 78-81, 2002

[14] A.H. Land und A. G. Doig: An automatic method of solving discrete programming problems. In: Econometrica Volume 28, S. 497- 520, 1960

[15] George Nemhauser und Laurence Wolsey: Integer and Combinatorial Optimization. Wiley Interscience, 1988, ISBN 0-471-35943-2

[16] Richard Crandall, Carl Pomerance Prime Numbers - A Computational Perspective Springer, New York, ISBN 0-387-25282-7

[17] Carl Pomerance: A Tale of Two Sieves, Notices of the AMS, 43 (1996) 1473-1485

[18] A. K. Lenstra & H. W. Lenstra, Jr.: The development of the number field sieve, Lecture Notes in Mathematics, 1554

[19] M. A. Weiss, Data Structures & Problem Solving using Java, Addison Wesley. ISBN 0-201-74835-5

[20] Donald E. Knuth: Backus Normal Form versus Backus Naur Form, Communications of the ACM 7 (December 1964)

[21] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilers - Principles, Techniques, and Tools, Addison Wesley, 2003, ISBN 0-201-10194-7

[22] Reinhard Wilhelm, Dieter Maurer: Übersetzerbau - Theorie, Konstruktion, Generierung, Springer-Verlag, 1997, ISBN 3-540-61692-6

[23] Wagner, Lucas: The History of Apple's "Syntax" Poster

[24] Wagner, F., Modeling Software with Finite State Machines: A Practical Approach, Auerbach Publications, 2006, ISBN 0-8493-8086-3

[25] Hopcroft, John E. und Ullman, Jeffrey D.: Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979, ISBN 0-201-02988-X

[26] Hopcroft, John E. und Ullman, Jeffrey D.: Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie, ISBN 3-89319-181-X

[27] Sander, Stucky und Herschel: Automaten, Sprachen, Berechenbarkeit, ISBN 3-519-02937-5

[28] Kohavi, Z.: Switching and Finite Automata Theory (engl.), McGraw-Hill, 1978

[29] Gill, A.: Introduction to the Theory of Finite-state Machines (engl.), McGraw-Hill, 1962

[30] Wuttke, H.-D. und Henke, K.: Schaltsysteme - Eine automatenorientierte Einführung, Pearson Studium, ISBN 3-8273-7035-3

Teil V
Anhang

- UML Tutorial: Part 1 – Class Diagrams; Robert C. Martin
- The State and Strategy Patterny, Design Patterns in Java; Bob Tarr
- Design Patterns - How Design Patterns Solve Design Problems
- An O(ND) Difference Algorithm and Its Variations; Eugene W. Myersx
- A Compact Guide to Lex & Yacc; Tom Niemann