Algorithmische Diskrete Mathematik

Skript zur Vorlesung im Sommersemester 2010

Ulf Lorenz, Lea Rausch, Thorsten Ederer, Thomas Opfer

Letzte Änderung: 8. Juli 2010

lorenz@mathematik.tu-darmstadt.de

Inhaltsverzeichnis

Vorwort

Dieses Skript über *Algorithmische Diskrete Mathematik* ensteht im Verlauf der gleichnamigen Veranstaltung von PD Dr. Ulf Lorenz an der Technischen Universität Darmstadt im Sommersemester 2010 mit tatkräftiger Unterstützung der Studenten. Obwohl bereits viele teils gute Skripte zu diesem Themengebiet existieren, haben wir uns entschieden den Stoff noch einmal aufzubereiten und eigene Schwerpunkte zu setzen. Wir hoffen, dass die Anlehnung an den Foliensatz der Veranstaltung die Studenten bei der Vorlesungsnachbereitung und Klausurvorbereitung unterstützt.

Die aktuelle Version des Skriptes können Sie von der zugehörigen Veranstaltungsseite (siehe `http://www3.mathematik.tu-darmstadt.de/hp/optimization/lorenz-ulf/`) herunterladen. Sollten Sie Fehler finden oder Verbesserungsvorschläge haben, teilen Sie uns diese bitte per E-Mail (`lorenz@mathematik.tu-darmstadt.de`) mit.

Als weiterführende Literatur kann eine Vielzahl von Büchern über diskrete Mathematik oder Algorithmen herangezogen werden. Beispielhaft seien das schlanke deutsche Lehrbuch *Diskrete Mathematik* [1] oder das größere englische Werk *Introduction to algorithms* [2] genannt, die sich inhaltlich gut als Vorlesungsergänzung eignen.

Literaturverzeichnis

[1] M. Aigner, *Diskrete Mathematik*, Vieweg+Teubner Verlag, 2006

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to algorithms*, The MIT press, 2001

[3] S. Grothklags, U. Lorenz, B. Monien, *From State-of-the-Art Static Fleet Assignment to Flexible Stochastic Planning of the Future*, Springer-Verlag Berlin Heidelberg 2009

Kapitel 1

Einführung

Die diskrete Mathematik als Teilgebiet der Mathematik befasst sich vorwiegend mit mathematischen Operationen über endlichen oder zumindest abzählbar unendlichen Mengen. Im Gegensatz zu anderen Gebieten wie der Analysis, die sich mit kontinuierlichen Funktionen oder Kurven über nicht abzählbaren, unendlichen Mengen beschäftigt, besitzen die in der diskreten Mathematik behandelten Objekte für gewöhnlich keine Stetigkeitseigenschaften. Was kann man mit endlichen Mengen anfangen? Man kann sie zumindest mal abzählen. Oft ist auch eine einfache Struktur in Form von Relationen auf diesen endlichen Mengen gegeben. Auf diese Weise entstehen z.B. die so genannten Graphen, die auch in diesem Skript eine tragende Rolle spielen.

An der TU Darmstadt ist die Algorithmische Diskrete Mathematik zudem eine Vorläuferveranstaltung zur Vertiefung in der Optimierung. Was in grauer Vorzeit als Kombinatorik begann und vor ca. 50 Jahren in die diskrete Mathematik mündete, war der Gedanke, dass es manchmal nicht genügt zu einem kombinatorischen Problem eine theoretische Lösung zu finden, sondern dass es auch wichtig ist, konstruktive und schnelle Algorithmen zur Verfügung zu haben, die die Lösung liefern.

Die einfache Verstädlichkeit von Problemstellungen und Lösungsansätzen, sowie die Entwicklung moderner Computer haben dazu beigetragen, dass die diskrete Mathematik sehr erfolgreich Einzug in viele Anwendungsfelder gefunden hat. Beispiele sind die Suche eines Navigationssystems nach einer guten Verbindung (Abbildung 1.1), die optimale Routenplanung in einem Gasnetzwerk oder der kosteneffiziente Betrieb eines Flugverkehrsnetzes. Auch wenn die Problemstellungen einfach erscheinen, stellt das Auffinden von geeigneten Lösungsverfahren eine sehr anspruchsvolle Aufgabe dar.

Ein so genannter Graph ist eine abstrakte Darstellung einer Menge von Objekten, die paarweise verbunden sein können. Wir werden die Begrifflichkeiten später natürlich noch formal festziehen. Das Beispiel zeigt eine Straßenkarte , die für einen Computer nur schwer zu analysieren ist, da sie zunächst interpretiert werden muss. Ein Mensch schaut drauf und es ist ihm schnell klar, wo sein Startpunkt, und sein Ziel ist, und wie er sich einen Weg vom Start zum Ziel suchen kann. Ein Computer kann das zunächst nicht. Er „sieht“ nur farbige Punkte. Um es auch einem Computer zu ermöglichen, uns die schnellsten Wege von einem gegebenen Startpunkt zu einem ebenfalls vorgegebenen Ziel zu berechnen, ist es nötig die Karte zu abstrahieren und die relevanten Informationen dieser Karte, nämlich der Zusammenhang von Straßen und Kreuzungen, herauszufiltern. Man sieht nun ein paar dicke Punkte, die die Kreuzungen darstellen, sowie Linien, die Straßenzüge zwischen Kreuzungen repräsentieren. Bezeichnen wir nun die Kreuzungen als Knoten und die Straßenzüge als Kanten und vergessen wir, dass Straßen und Kreuzungen grau sind, asphaltiert sind u.s.w., haben wir es nur noch mit einem mathematischen, sehr einfachen Gebilde zu tun, dem Graphen. Einem Computer beizubringen, wie er kürzeste Wege für uns in einem Graphen findet, das werden wir ihm im Verlauf der Vorlesung noch beibringen.

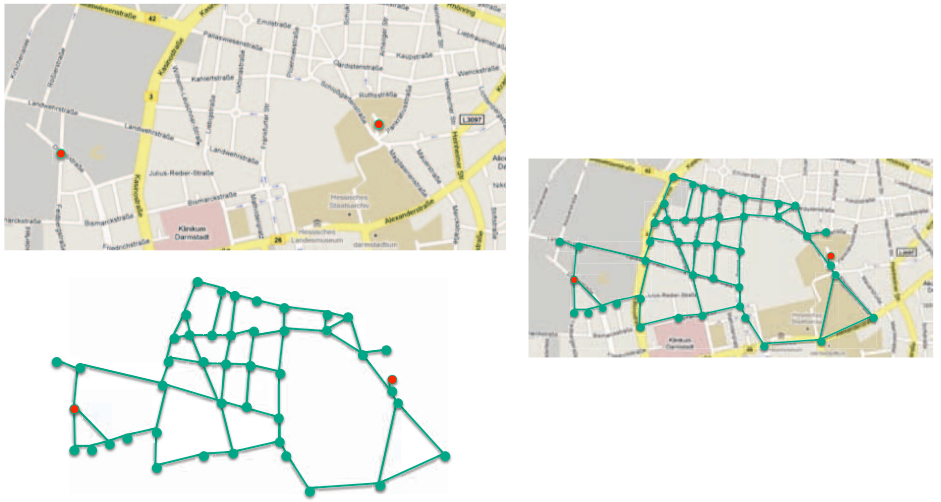


Abbildung 1.1: Bilderfolge zur Erstellung eines Graphen aus einer Straßenkarte.

Dieses einführende Kapitel ist in zwei Abschnitte gegliedert. Im ersten Teil wird ein Optimierungsproblem aus der Industrie präsentiert. Es soll in erster Linie als Einblick in dieses Themengebiet dienen. Die mathematischen Methoden werden dabei nur angeschnitten und können in den Vorlesungen Optimierung 1 bis 3 vertieft werden. Im zweiten Teil werden erste Algorithmen am Beispiel eines einfachen diskreten Problems vorgestellt, für das nur grundlegende Analysiskenntnisse erforderlich sind.

1.1 Anwendungsbeispiel: Optimierung in der Flugindustrie

Die Flugindustrie ist ein Wirtschaftszweig, für den mathematische Methoden der diskreten Mathematik und insbesondere der Optimierung unerlässlich geworden sind. Die Planung von international operierenden Unternehmen kann nicht mehr allein durch Erfahrung und Expertenwissen gemeistert werden, da schon die Problembeschreibungen zu groß werden um für Menschen vollständig erfassbar zu sein. Aspekte wie die Flugplanung, Marktanalyse, Personalverwaltung sowie Netzwerkdesign und -kontrolle werden heutzutage modelliert und computergestützt ausgewertet. Im folgenden wird ein Flottenzuweisungsproblem vorgestellt, wie es ähnlich auch wirklich verwendet wird. Dieses Teilkapitel soll zum einen dazu dienen, zu zeigen wie wichtig Geschwindigkeit bei Berechnungen ist und wie man sie bekommt. Zum anderen ist das Beispiel sehr detailliert dargestellt, um auch dem ungeübten Leser einfach mal einen visuellen Eindruck zu geben, wie ein an die Realität angelehntes mathematisches Optimierungsmodell am Ende aussieht. Es wird nicht erwartet, dass die Formalien verstanden werden, denn dies ist hier gar nicht nötig. Es reicht, den Fließtext zu lesen.

Das Problem der *Flottenzuweisung* [3] ist, bei gegebenem Flugplan für die verfügbaren Flugzeuge zu entscheiden, welcher Flugzeugtyp welcher Flugstrecke zugeteilt wird. Ziel ist aus Sicht der Fluggesellschaft die Gewinnmaximierung. Dabei müssen verschiedene notwendige Bedingungen eingehalten werden. Beispielsweise ist die Anzahl der Flugzeuge beschränkt, sie haben eine bestimmte Passagierkapazität und einen unterschiedlichen Treibstoffverbrauch. Um mit Computerunterstützung den Firmengewinn zu maximieren, ist es in vielen Fällen unbedingt erforderlich, auch die schnellsten und besten Algorithmen bei den Berechnungen einzusetzen. Das liegt zum einen daran, dass schnellere Verfahren eine größere Detailgenauigkeit bei der Prozessoptimierung zulassen, zum anderen aber auch daran, dass sich die verschiedenen Firmen in Konkurrenz zueinander befinden. Wenn eine Planung bzw. eine ad-hoc-Umplanung nach einer Störung zu lange dauert, kann das dazu führen, dass sich potentielle Kunden an Konkurrenten wenden.

Für die mittelfristige Planung wird typischerweise ein zyklischer Zeitabschnitt (z.B. eine Woche) betrachtet und durch ein sogenanntes *Time-Space-Netzwerk* (Abbildung 1.2) modelliert. Die Abbildung zeigt drei verschiedene Flughäfen (AAA, BBB und CCC), auf denen verschiedene Ereignisse im Laufe der Zeit stattfinden. Die Zeit vergeht in der Graphik von links nach rechts. Ein Pfeil, der Knoten verschiedener Zeitleisten verbindet, ist dann die Darstellung eines Fluges (auf einer vorgesehenen Flugstrecke) und ein Pfeil zwischen zwei Knoten derselben Zeitleiste repräsentiert eine so genannte Bodenkante. In dieser Zeit befindet sich ein Flugzeug am Boden. Die erste Modellierung, die wir zur Beschreibung unseres Problems heranziehen ist also wiederum ein Graph, dessen Knoten die Abreise- und Ankunftsereignisse und dessen Kanten die Flüge repräsentieren. Man spricht in diesem Fall von einem *gerichteten Graphen*, da für jede Kante der Startpunkt vom Endpunkt unterschieden wird.

Die mathematische Optimierung nutzt solche Graphmodelle gerne als Grundmodelle und Ausgangspunkt für mathematische Optimierungsmodelle. Um ein mathematisches Optimierungsproblem zu formulieren, werden im nächsten Schritt Parameter des Modells bestimmt, sowie Bedingungen, die das Modell einhalten muss, so genannte Nebenbedingungen, dazu eine Zielfunktion, die uns hilft gute von schlechten Lösungen zu unterscheiden und eine Menge von Variablen, in denen wir am Ende der Berechnungen unsere Lösung ablesen können.

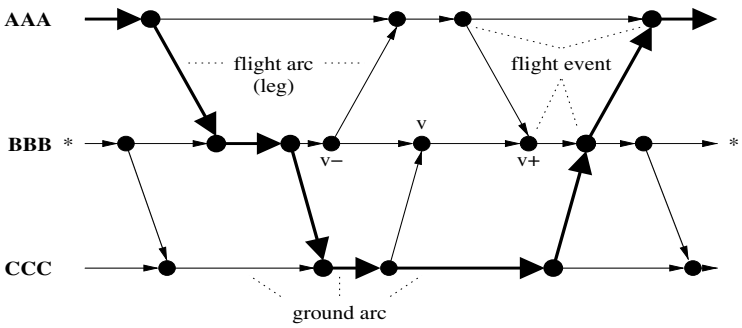


Abbildung 1.2: Diagramm eines Raumzeit-Netzwerks. [3]

Die Eingabedaten/Parameter des Problems sollen wie folgt bezeichnet werden:

$\mathscr{F}$	Menge der verfügbaren Flotten
N_f	Anzahl der verfügbaren Flugzeuge in Flotte $f \in \mathscr{F}$
$\mathscr{L}$	Menge der Flugstrecken (Flugplan)
$\mathscr{F}_l$	Menge der Flotten, die auf Flugstrecke $l \in \mathscr{L}$, $\mathscr{F}_l \subseteq \mathscr{F}$ eingesetzt werden können
s_l^{dep}	Abgangs-Flughafen von Flugstrecke $l \in \mathscr{L}$
s_l^{arr}	Ankunfts-Flughafen von Flugstrecke $l \in \mathscr{L}$
$t_{l,f}^{dep}$	Abflug-Zeitpunkt für Flugstrecke $l \in \mathscr{L}$, wenn dort Flotte $f \in \mathscr{F}_l$ eingesetzt wird
$t_{l,f}^{arr}$	Lande-Zeitpunkt für Flugstrecke $l \in \mathscr{L}$, wenn dort Flotte $f \in \mathscr{F}_l$ eingesetzt wird
$p_{l,f}$	Profit von Flugstrecke $l \in \mathscr{L}$, wenn dort Flotte $f \in \mathscr{F}_l$ eingesetzt wird
$g(k,l,f)$	Minimale Bodenzeit am Flughafen s_k^{arr}, wenn ein Flugzeug der Flotte f zwischen den Flugstrecken k und l kurz Halt macht; dann gilt $s_k^{arr} = s_l^{dep}$

Um das Verbindungs-Netzwerk zu modellieren, definiert man die Mengen der zulässigen Anschluss-Flugstrecken $C_{l,f}$ (bzw. Vorgänger-Flugstrecken $C_{l,f}^{-1}$) für ein Flugzeug der Flotte $f \in \mathscr{F}$, das die Flugstrecke $l \in \mathscr{L}$ hinter sich (bzw. vor sich) hat:

$$C_{l,f} = \left\{ k \in \mathscr{L} | f \in \mathscr{F}_k, s_l^{arr} = s_k^{dep}, t_{l,f}^{arr} + g(l,k,f) \leq t_{k,f}^{dep} \right\} \cup \{*\}$$
$$C_{l,f}^{-1} = \left\{ k \in \mathscr{L} | f \in \mathscr{F}_k, s_k^{arr} = s_l^{dep}, t_{k,f}^{arr} + g(k,l,f) \leq t_{l,f}^{dep} \right\} \cup \{*\}$$

Aus dem Modell wird ein sogenanntes *gemischt-ganzzahliges lineares Programm* erstellt. Ziel ist es, *Entscheidungsvariablen* (die diskrete oder kontinuierliche Werte annehmen dürfen) so zu bestimmen, dass eine *Kostenfunktion* (allgemein *Zielfunktion*) unter Einhaltung verschiedener *Nebenbedingungen* („*subject to*“) ein Extremum („*maximize*“) annimmt. Ein Programm für das Flottenzuweisungsproblem kann beispielsweise so aussehen:

$$\text{maximize} \quad \sum_{k \in \mathscr{L}} \sum_{f \in \mathscr{F}_k} p_{k,f} \left(\sum_{l \in C_{k,f}} x_{k,l,f} \right)$$

$$\begin{aligned}
\text{subject to} \quad \sum_{f \in \mathscr{F}_k} \sum_{l \in C_{k,f}} x_{k,l,f} &= 1 && \forall k \in \mathscr{L} \\
\sum_{k \in C_{l,f}^{-1}} x_{k,l,f} - \sum_{m \in C_{l,f}} x_{l,m,f} &= 0 && \forall l \in \mathscr{L}, f \in \mathscr{F}_l \\
\sum_{l \in \mathscr{L}} x_{*,l,f} &\leq N_f && \forall f \in \mathscr{F} \\
x_{k,l,f} &\in \{0,1\} && \forall k \in \mathscr{L}, f \in \mathscr{F}_k, l \in C_{k,f}
\end{aligned}$$

Ein solches Programm kann als der (stark) verallgemeinerte Fall der Suche nach der Extremstelle einer Kostenfunktion verstanden werden, wie man es in der elften Klasse lernt.

Um das Time-Space-Netzwerk zu modellieren, wird folgende Notation vereinbart:

$\mathcal{V}$	Menge aller Flugereignisse
$v_{l,f}^{dep}$	$= (t_{l,f}^{dep}, s_l^{dep}, f)$; Flugereignis, das dem Abflug von Flugstrecke l entspricht, wenn dort Flotte f eingesetzt wird
$v_{l,f}^{arr}$	$= (t_{l,f}^{arr}, s_l^{arr}, f)$; Flugereignis, das der Landung von Flugstrecke l entspricht, wenn dort Flotte f eingesetzt wird
v^+	Nachfolgendes Flugereignis $v \in \mathcal{V}$ am gleichen Flughafen und von der gleichen Flotte; $*$ falls v das letzte Flugereignis ist
v^-	Vorangehendes Flugereignis $v \in \mathcal{V}$ am gleichen Flughafen und von der gleichen Flotte; $*$ falls v das letzte Flugereignis ist
$\mathcal{V}_f^*$	Menge der Flugereignisse von Flotte f die kein Vorgänger-Flugereignis haben

Ein Programm zur Optimierung der Flottenzuweisung auf dem Raumzeit-Netzwerk ist dann:

$$\text{maximize} \quad \sum_{l \in \mathcal{L}} \sum_{f \in \mathcal{F}_l} p_{l,f}\, y_{l,f}$$

$$\begin{aligned}
\text{subject to} \quad \sum_{f \in \mathcal{F}_l} y_{l,f} &= 1 && \forall l \in \mathcal{L} \\
\sum_{v_{l,f}^{arr} = v} y_{l,f} - \sum_{v_{l,f}^{dep} = v} y_{l,f} + z_{v^-,v} - z_{v,v^+} &= 0 && \forall v \in \mathcal{V} \\
\sum_{v \in \mathcal{V}_f^*} z_{*,v} &\leq N_f && \forall f \in \mathcal{F} \\
y_{l,f} &\in \{0,1\} && \forall l \in \mathcal{L}, f \in \mathcal{F}_l \\
z_{v,v^+} &\geq 0 && \forall v \in \mathcal{V} \\
z_{*,v} &\geq 0 && \forall v \in \mathcal{V}^*
\end{aligned}$$

Die Zielfunktionskoeffizienten c_{fi} beschreiben die Kosten des Fluges i mit Flotte f und die Zielfunktion damit die Gesamtkosten. Man kann hier sehr schön das Wesen einer Nebenbedingung sehen: Die erste Nebenbedingung fordert, dass jeder Flug von genau einer Flotte übernommen wird. Die zweite Restriktion beschreibt die Flusserhaltung in jedem Knoten: es müssen genauso viele Flugzeuge den Flughafen verlassen, wie dort ankommen. Die dritte Bedingung fordert, dass Anschlussflüge von der gleichen Flotte übernommen werden. Die vierte Bedingung sichert, dass höchstens soviele Flugzeuge eines Typs unterwegs sind, wie auch existieren.

Ein mathematisches Gebilde, wie wir es hier dargestellt haben, nennt man auch ein mathematisches Programm. Die Größe eines Programms wird durch die Anzahl der Variablen und Nebenbedingungen charakterisiert. Eine realistische Instanz des vorgestellten Modells hat beispielsweise |Flüge|·|Flotten| ≈ 230000 Variablen und 2·|Flüge|·|Flotten| ≈ 500000 Restriktionen.

Heutzutage dominieren mathematische Methoden auch die Lösungsfindung, aber wegen der Größe konnte dieses Problem noch vor wenigen Jahren nicht exakt gelöst werden. Daher kamen Näherungsverfahren zum Einsatz. Sie basieren zum Beispiel auf so gennanter lokaler Suche. Dabei wird eine gegebene zulässige Lösung des Problems durch leichte Abänderung schrittweise verbessert, bis ein (möglicherweise nur lokales) Optimum erreicht ist. Man nimmt also bewusst in Kauf, die beste Lösung nicht zu finden, um den Rechenaufwand in Grenzen zu halten. Bei heuristischen Ansätzen weiß man im voraus oft nicht einmal, ob überhaupt eine zufriedenstellende Lösung gefunden werden kann. In der Praxis erweisen sie sich allerdings oft als erstaunlich erfolgreich.

Ein Beispiel für ein heuristisches Optimierungsverfahren ist das *Simulated Annealing*:

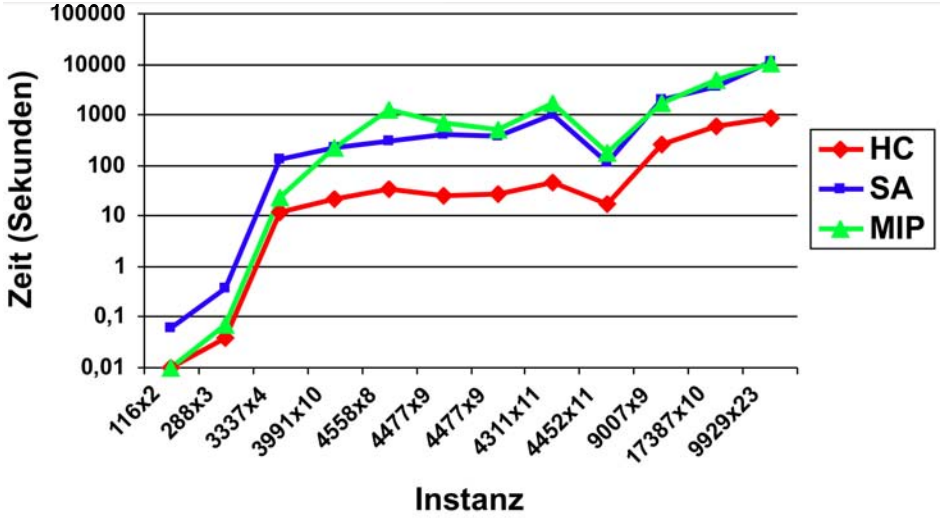
```
Generiere initiale Lösung
Wahrscheinlichkeit P = P0
do
    do
        Generiere eine lokal benachbarte Lösung
        Akzeptiere eine Verschlechterung mit Wahrscheinlichkeit P
    until Abbruchkriterium
    P = Update(P)
until Frozen
```

Obwohl die Herangehensweise völlig verschieden ist, zeigen exakte Verfahren und Heuristiken für verschiedene Problemgrößen vergleichbar gute Ergebnisse (Abbildung 1.3). Während die Lösungsqualität eines exakten Verfahrens nicht zu überbieten ist, findet man mit Heuristiken manchmal beinahe so gute Ergebnisse in erheblich kürzerer Zeit.

Da die Leistung von Computern (obwohl sie in den letzten Jahrzehnten enorm zugenommen hat) beschränkt ist, muss man andere Wege finden um Berechnungen zu beschleunigen. Ein sehr intuitiver Zugang ist die Parallelisierung von Algorithmen, d.h. die gleichzeitige Bearbeitung von Teilproblemen auf mehreren Prozessoren oder Computern.

Bei genauerer Betrachtung ergeben sich jedoch einige Probleme. Viele algorithmische Ansätze sind prinzipiell nicht parallelisierbar, im einfachsten Fall weil Zwischenergebnisse in die weitere Berechnung eingehen. Auch wenn ein Algorithmus prinzipiell in kleinere Häppchen zerteilt werden kann, die parallel verarbeitet werden können, kann die Implementierung sehr kompliziert sein. Beispielsweise muss man sich über eine gute Zerlegung des Problems und über die Synchronisation der Prozesse Gedanken machen. Unter Umständen kann ein paralleler Algorithmus auch schlecht skalieren, d.h. eine Verdopplung der Hardware kann statt einer 50-prozentigen Senkung der Laufzeit nur eine 10-prozentige Senkung, oder gar ein Eröhung der Laufzeit zur Lösungsfindung zur Folge haben.

Trotz dieser Schwierigkeiten ist Parallelität ein entscheidendes Ziel bei der Entwicklung und Implementierung von Algorithmen, da sich beeindruckende Laufzeitverbesserungen erzielen lassen (Abbildung 1.4).



Verfahren	HC	SA	MIP
∅-Lösungsqualität	98,5%	99,7%	>99,9%

Abbildung 1.3: Diagramm zum Vergleich verschiedener Optimierungsverfahren.
Ein exaktes Optimierungsverfahren (MIP: *mixed integer program*) wird mit zwei heuristischen Verfahren basierend auf lokaler Suche (SA: *simulated annealing*, HC: *hill climbing*) für verschiedene Problemgrößen verglichen. SA und MIP weisen eine sehr hohe Lösungsqualität auf. Ihre Laufzeit war bei großen Problemen vergleichbar und bei kleinen Problemen gewann die Heuristik. HC zeigt bei allen Problemgrößen die kürzeste Laufzeit, findet aber auch die schlechtesten Lösungen.

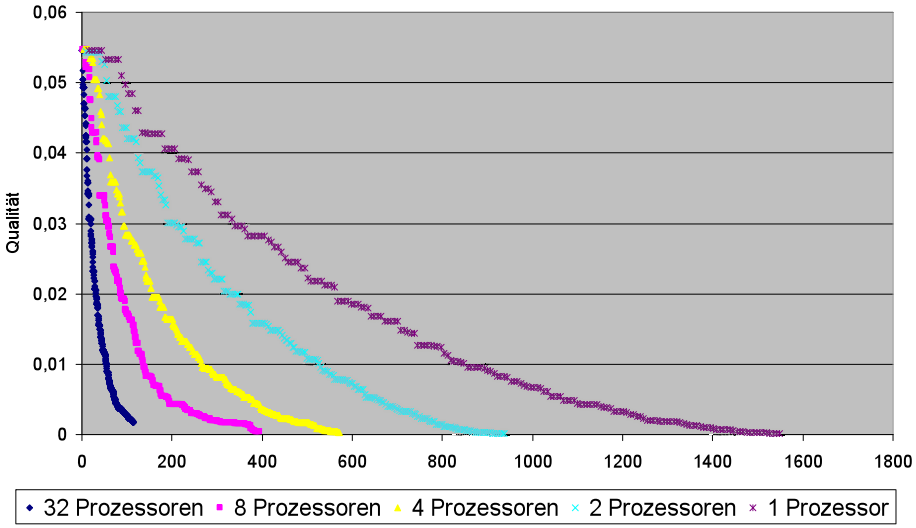


Abbildung 1.4: Diagramm zur Laufzeitverbesserung durch Parallelisierung.
Parallelisierung meint das gleichzeitige Ausführen von Teilaufgaben zur Lösungsfindung auf mehreren Computern, Prozessoren oder Prozessorkernen. Insbesondere durch den letzte Punkt ist in den vergangenen Jahren durch das Aufkommen von Mehrkernprozessoren die Parallelisierbarkeit von Algorithmen relevant geworden. Die Grafik zeigt die erzielte Qualität eines Ergebnisses aufgetragen über der Rechenzeit (beides in willkürlichen Einheiten) für verschiedene Prozessoranzahlen im Vergleich. Offenbar skaliert dieser Algorithmus gut, sodass eine Verdopplung der Prozessorzahl jeweils eine nennenswerte Leistungssteigerung bewirkt. Parallelisierung ermöglicht es dem Nutzer (sofern er über entsprechende Finanzen verfügt), schnell bessere Ergebnisse zu erhalten.

1.2 Algorithmische Beispiele: Das Maxsummenproblem

Als erstes Beispielproblem, um zu zeigen, wie sehr es sich lohnt, sich auf die Suche nach guten Algorithmen zu machen, soll das *Maxsummenproblem* dienen:

Definition 1.1. *Sei $a[1\,..\,n]$ ein Array ganzer Zahlen und $f(i,j) = a_i + \cdots + a_j$ eine Teilsumme aus diesem Array. Die größte Teilsumme* $\max\{\, f(i,j) \mid 1 \le i \le j \le n \,\}$ *heißt Maxsumme und ihre Berechnung heißt Maxsummenproblem.*

Trotz seiner einfachen Formulierung lassen sich daran gut algorithmische Ansätze vergleichen. Der Resourcenverbrauch eines Algorithmus soll in diesem Beispiel vereinfacht durch die Anzahl der benötigten Vergleiche $V(n)$ und die Anzahl der benötigten Additionen $A(n)$ charakterisiert werden. Die Summe der Additionen und Vergleiche soll Laufzeit $T(n) = A(n) + V(n)$ heißen.

Nachfolgend werden vier Algorithmen zum Lösen des Maxsummenproblems vorgestellt. Ein interessierter Leser kann sich an dieser Stelle kurz im Kopf die Maxsumme der beiden Listen $a = (\,3,\,-2,\,4,\,-5\,)$ und $a = (\,-10,\,-5,\,2,\,-7,\,3,\,6,\,-9,\,11\,)$ ausrechnen und später überlegen, welchen der vorgestellten Algorithmen er dabei verwendet hat.

Der naive Algorithmus

Der naive Ansatz ist, zunächst alle Elemente $f(i,j)$ der Menge auzuwerten und anschließend das Maximum dieser Elemente zu suchen. Die Berechnung wird an folgendem Beispiel deutlich:

Tabelle 1.1: Auswertung des naiven Algorithmus am Beispiel $a = (\,3,\,-2,\,4,\,-5\,)$

$f(1,1) = 3$	$f(1,2) = 3-2$	$f(1,3) = 3-2+4$	$f(1,4) = 3-2+4-5$
$f(2,2) = -2$	$f(2,3) = -2+4$	$f(2,4) = -2+4-5$	
$f(3,3) = 4$	$f(3,4) = 4-5$		
$f(4,4) = -5$			

Für die Auswertung jedes Eintrags wird eine Summe aus Einträgen der ersten Spalte gebildet. Jede Berechnung eines $f(i,j)$ erfordert $j-i$ Operationen. Aus den Ergebnissen kann man mit folgendem Pseudocode das größte Element finden:

```
maxsum := f (1,1)
for i := 1 to n do
    for j := i to n do
        falls f (i,j) > maxsum
            maxsum := f (i,j)
```

Dabei speichert `maxsum` das aktuelle Maximum der verglichenen Einträge und `f(i,j)` berechnet die Teilsummen. Die Schleifenvariablen laufen anschaulich eine dreieckige Fläche ab, die gerade den Einträgen der Beispieltabelle entspricht. Ist das aktuelle Element größer als das aktuelle Maximum, wird es als neues Maximum behalten.

Die Anzahl der benötigten Additionen ergibt sich als Summe der benötigten Rechenschritte aller Tabelleneinträge:

$$A_1(n) = \sum_{i=1}^{n}\sum_{j=i}^{n}(j-i) = \sum_{i=1}^{n}\sum_{k=1}^{n-i} k = \sum_{i=1}^{n}\frac{1}{2}(n-i)(n-i+1) = \frac{1}{6}n^3 - \frac{1}{6}n$$

Im ersten Schritt wurde die Variable substituiert. In den darauffolgenden Schritten wurden die Summenformeln für $\sum_a a$ und $\sum_a a^2$ angewandt.

Die Berechnung des Maximums bestimmt die Anzahl der benötigten Vergleiche:

$$V_1(n) = -1 + \binom{n}{2} + \binom{n}{1} = -1 + \sum_{i=1}^{n} i = -1 + \frac{1}{2}n\cdot(n+1) = \frac{1}{2}n^2 + \frac{1}{2}n - 1$$

Um das Maximum von L Tabelleneinträgen zu bestimmen, benötigt man $L-1$ Vergleiche. Diese Beziehung lässt sich kombinatorisch oder analytisch einsehen. Im ersten Fall berechnet man, auf wie viele Möglichkeiten man Anfang und Ende der Teilsumme auf das Array verteilen kann (nämlich $\binom{n}{2} + \binom{n}{1}$, unterteilt nach Anfang und Ende verschieden oder gleich), im zweiten Fall summiert man die Vergleiche für Teilsummen der Länge $1, 2, \ldots, n$ $(= \sum_{i=1}^{n} i)$.

Die Laufzeit des naiven Algorithmus ergibt sich somit zu:

$$T_1(n) = V_1(n) + A_1(n) = \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n - 1$$

Der normale Algorithmus

Wenn man die Auswertungen des naiven Algorithmus betrachtet, fällt auf, dass sich die meisten Berechnungen wiederholen. Bei genauerer Betrachtung der Tabelle erkennt man, dass ein $f(i,j+1)$ nicht immer neu aufsummiert werden muss, sondern als $f(i,j) + a_{j+1}$ berechnet werden kann. Eine Umgestaltung der Tabelle hebt diesen Zusammenhang hervor:

Tabelle 1.2: Auswertung des normalen Algorithmus am Beispiel $a = (\,3,\,-2,\,4,\,-5\,)$

$f(1,1) = a_1 = 3$	$f(1,2) = f(1,1) - 2$	$f(1,3) = f(1,2) + 4$	$f(1,4) = f(1,3) - 5$
	$f(2,2) = a_2 = -2$	$f(2,3) = f(2,2) + 4$	$f(2,4) = f(2,3) - 5$
		$f(3,3) = a_3 = 4$	$f(3,4) = f(3,3) - 5$
			$f(4,4) = a_4 = -5$

Die Zeilen lassen sich nun trivial von links nach rechts auswerten. Dazu muss man lediglich das alte Ergebnis weiterverwenden und den letzten Eintrag in der aktuellen Spalte bzw. den entsprechenden Eintrag des Arrays nachschauen.

Der Pseudocode des normalen Algorithmus sieht wie folgt aus:

```
maxsum := f (1,1)
for i := 1 to n do
    fij := 0
    for j := i to n do
        fij := fij + a[j]
        falls fij > maxsum
            maxsum := fij
```

Als Änderung zum naiven Algorithmus wurde die temporäre Variable `fij` eingeführt, die beim Durchlaufen einer Tabellenzeile mit den Array-Einträgen zusammengezählt wird. Somit entfallen die ursprünglichen Auswertungen von $f(i,j)$ in der inneren Schleife. Das Maximum der ausgewerteten Einträge findet man analog zum ersten Algorithmus, daher bleibt $V(n)$ unverändert. Für die Auswertung jedes $f(i,j)$ mit $j > i$ ist aber nur noch eine statt $j - i$ Additionen erforderlich:

$$A_2(n) = \sum_{i=1}^{n}(i-1) = -n + \sum_{i=1}^{n} i = \frac{1}{2}n^2 - \frac{1}{2}n$$

Daraus ergibt sich folgende Laufzeit für den normalen Algorithmus:

$$T_2(n) = V(n) + A(n) = n^2 - 1$$

Der *divide & conquer* Algorithmus

Der *divide & conquer* Algorithmus verfolgt einen anderen Ansatz. Die bisher betrachteten Algorithmen sind implizit davon ausgegangen, dass man alle $f(i,j)$ auswerten und nach dem größten Ergebnis suchen muss. Diese Vermutung stellt sich als falsch heraus: Man kann das größte $f(i,j)$ finden, ohne jemals alle $f(i,j)$ explizit ausgerechnet zu haben. Die Idee hinter *divide & conquer* Algorithmen besteht darin, das Originalproblem in Teilprobleme zu zerlegen (*divide*), die sich in der Regel leichter lösen lassen, und die Gesamtlösung anschließend aus den Teillösungen zusammenzusetzen (*conquer*).

Zur Vereinfachung der Analyse wird im Folgenden angenommen, dass die Eingabelänge eine Zweierpotenz $n = 2^k$ ist. (Die Erweiterung auf beliebige Eingaben ist nicht schwer. Man kann die Eingabelänge künstlich erweitern oder mit Fallunterscheidungen arbeiten.) Zur Zerlegung des Problems definieren wir drei Hilfsmengen:

$$\sigma(l,r) := \max\{\, f(i,j) \mid l \le i \le j \le r \,\}$$

$$s_1(l,r) := \max\{\, f(i,r) \mid l \le i \le r \,\}$$
$$s_2(l,r) := \max\{\, f(l,j) \mid l \le j \le r \,\}$$

Die Funktion $\sigma(l,r)$ mit zwei variablen Indizes ist die ursprüngliche Definition der Maxsumme zwischen den Elementen a_l und a_r. Die Funktionen $s_1(l,r)$ und $s_2(l,r)$ haben jeweils nur einen variablen Index und stellen das Maximum von Teilsummen „bis zu“ bzw. „ab“ einem bestimmten Element dar. Sie sollen *linksgerichtete Maxsumme* und *rechtsgerichtete Maxsumme* heißen.

Der entscheidende Schritt ist nun, zu erkennen, dass das Maxsummenproblem in drei Teilprobleme zerlegt werden kann:

$$\sigma(1,n) = \max\{\, \sigma(1,\frac{n}{2}),\ \sigma(\frac{n}{2}+1,n),\ s_1+s_2 \,\}$$

Diese Zerlegung ist möglich, da die Menge der Index-Paare (i,j) damit *partitioniert* wird, d.h. man zerlegt sie in disjunkte Teilmengen, deren Vereinigung wieder die Gesamtmenge ergibt. Anschaulich gibt es nur drei Möglichkeiten, wie die maximale Teilsumme bezüglich der Array-Mitte beschaffen sein kann: (1) sie beginnt und endet in der linken Hälfte, (2) sie beginnt in der linken und endet in der rechten Hälfte, (3) sie beginnt und endet in der rechten Hälfte. Es führt also zum Ziel, die maximalen Teilsummen dieser drei *Klassen* von Teilsummen zu bestimmen und die größte auszuwählen.

Am Beispiel $a = (-10, -5, 2, -7, 3, 6, -9, 11)$ wird die Zerlegung vorgeführt. Man teilt die Liste in der Mitte und berechnet auf den Hälften die Maxsummen $\sigma(1,4) = 7$ und $\sigma(5,8) = 11$. Hat man nun ein schnelles Verfahren, um die beiden maximalen Teilsummen der linken und der rechten Hälfte auszurechnen, lässt sich auch die maximale Teilsumme der gesamten Folge schnell berechnen. Man muss lediglich noch auf der linken Hälfte die linksgerichtete Maxsumme $s_1(1,4) = 0$ und auf der rechten Hälfte die rechtsgerichtete Maxsumme $s_2(5,8) = 11$ berechnen. Diese beiden Berechnungen gehen sehr schnell. Die Behauptung ist, dass die ganze Maxsumme sich dann als $\sigma(1,8) = \max\{7, 11, 0+11\} = 11$ berechnen lässt.

Der *divide & conquer* Algorithmus wendet diese Zerlegung rekursiv an, d.h. er wendet die Zerlegung für die Berechnung der Maxsummen der beiden Hälfen erneut an, auf die Viertel wieder, und so weiter. Die geschachtelte Zerlegung hört auf, wenn die Teilprobleme leicht genug geworden sind, also wenn nur noch Maxsummen von ein-elementigen Teilarrays gebildet werden. Somit findet im ganzen Algorithmus keine echte Berechnung einer Maxsumme mehr statt, sondern es werden nur noch gerichtete Maxsummen berechnet und nach obiger Rekursionsformel zu neuen Teilergebnissen zusammengesetzt, bis das Gesamtergebnis vorliegt. Der Ablauf des Algorithmus ist schematisch in Abbildung 1.5 dargestellt.

Der Pseudocode hat dementsprechend folgende Form:

```
sigma (l,r):
    falls l == r
        Gib a[l] aus.
    sonst
        sigma1 := sigma (l, (l+r)/2)
        sigma2 := sigma ((l+r)/2+1, r)
        s1_s2  := s1 (l, (l+r)/2) + s2 ((l+r)/2+1, r)
        maxsum := max (sigma1, sigma2, s1_s2)
        Gib maxsum aus.
```

Die rekursive Funktion `sigma(l,r)` prüft, ob sie auf einer ein-elementigen Menge aufgerufen wird. Ist das der Fall, liefert sie als Ergebnis den entsprechenden Eintrag des Arrays zurück. Ist das nicht der Fall, zerlegt sie das Problem in die besprochenen Teilprobleme, wobei sie sich selbst aufruft um manche dieser Teilprobleme zu lösen.

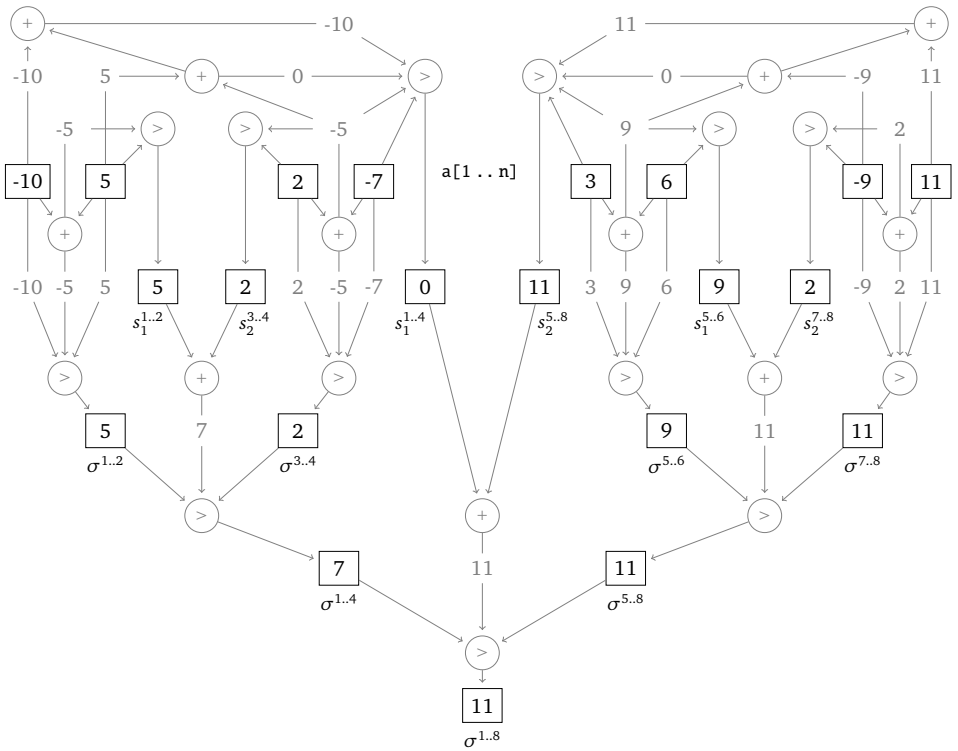


Abbildung 1.5: Schematischer Ablauf des *divide & conquer* Algorithmus.
Die eingerahmten Werte sind Eingaben, relevante Zwischenergebnisse und Ausgaben. Der „+"-Knoten steht für eine Addition, der „>"-Knoten für die Berechnung des Maximums (durch einen oder mehrere Vergleiche). Das Schema hat eine baumartige Struktur, da der Algorithmus rekursiv abläuft, d.h. das Originalproblem wird solange in Teilprobleme gleicher Art zerlegt, bis sie leicht genug zu lösen sind. Jede Maxsumme σ wird als Maximum zweier Teilmaxsummen σ und der Summe zweier gerichtet Maxsummen s_1 und s_2 berechnet. Das Lösen aller Teilprobleme erfordert weniger Operationen als das Lösen des ganzen Problems mit dem normalen Algorithmus.

Die Laufzeit des *divide & conquer* Algorithmus kann aus einer Rekursionsgleichung (*Rekurrenz*) bestimmt werden. Zunächst ist die Laufzeit auf einem Array der Länge 1 bekannt (dafür sind keine Operationen nötig, da der Wert abgelesen wird). Zudem kann die Laufzeit durch die Zerlegung ausgedrückt werden als die Laufzeit der beiden halben Probleme, den Zusatzaufwand durch die Berechnung der gerichteten Maxsummen (das sind $n-1$ Additionen und $n-2$ Vergleiche) und die abschließende Maximumsberechnung (noch 2 Vergleiche):

$$T_3(1) = 0 \qquad \text{und} \qquad T_3(n) = 2 \cdot T_3\left(\frac{n}{2}\right) + (2n-3) + 2 \quad \text{für } n > 1$$

Für Eingaben der Länge $n = 2^k$ (also $k = \log_2 n$) gilt dann:

$$\begin{aligned}
T_3(2^k) &= 2 \cdot T_3(2^{k-1}) + (2^{k+1} - 1) \\
&= 2 \cdot \left[2 \cdot T_3(2^{k-2}) + 2^{k+1} - 1\right] + (2^{k+1} - 1) \\
&= 4 \cdot T_3(2^{k-2}) + (2^{k+1} - 2) + (2^{k+1} - 1) \\
&= 4 \cdot \left[2 \cdot T_3(2^{k-3}) + 2^{k+1} - 1\right] + (2^{k+1} - 2) + (2^{k+1} - 1) \\
&= 8 \cdot T_3(2^{k-3}) + (2^{k+1} - 4) + (2^{k+1} - 2) + (2^{k+1} - 1) \\
&= \quad \ldots \\
&= 2^k \cdot T_3(2^{k-k}) + k \cdot 2^{k+1} - \sum_{i=1}^{k} 2^{i-1} \\
&= 0 + 2k \cdot 2^k - (2^k - 1) \\
&= (2k-1) \cdot 2^k + 1 \\
&= (\log_2 n - 1) \cdot n + 1
\end{aligned}$$

Durch Auflösen der Rekursionsbeziehung ergibt sich eine explizite Darstellung der Laufzeit.

Der *clevere* Algorithmus

Für das Maxsummenproblem gibt es noch einen besseren als den *divide & conquer* Algorithmus. Ein cleverer Algorithmus löst das Maxsummenproblem in einem einzigen Durchlauf des Arrays $a[1\,..\,n]$. Dazu betrachtet man folgende Hilfsfunktionen:

$$\text{maxtail}(k) := \max\{\sum_{l=i}^{k} a_l \mid 1 \le i \le k\}$$

$$\text{maxsum}(k) := \max\{\sum_{l=i}^{j} a_l \mid 1 \le i \le j \le k\}$$

Die Funktion `maxsum(k)` entsprich also der echten Maxsumme $\sigma(1,k)$ des *divide & conquer* Algorithmus und die Funktion `maxtail(k)` entspricht der linksgerichteten Maxsumme $s_1(1,k)$ des *divide & conquer* Algorithmus. Die Definitionen bringen somit nichts wirklich Neues, sondern erlauben nur eine geeignete Schreibweise.

Man kann zeigen, dass die Werte von `maxsum` sich leicht aus den Werten von `maxtail` berechnen lassen, welche sich wiederum leicht aus dem Array berechnen lassen. Daraus ergibt sich der Pseudocode für den *scanline* Algorithmus:

```
maxtail := a[1]
maxsum  := a[1]
for k := 2 to n do
    maxtail := max (maxtail+a[k], a[k])
    maxsum  := max (maxtail, maxsum)
```

Der Beweis der Formeln erfolgt über vollständige Induktion. Zu den Startwerten $\text{maxsum}(1) = a_1$ und $\text{maxtail}(1) = a_1$ leitet man dazu folgende Iterationsvorschriften ab:

$$\begin{aligned}
\text{maxtail}(k+1) &= \max\{\sum_{l=i}^{k+1} a_l \mid 1 \le i \le k+1\} \\
&= \max\left(\{a_{k+1} + \sum_{l=i}^{k} a_l \mid 1 \le i \le k\} \cup \{a_{k+1}\}\right) \\
&= \max\left(a_{k+1} + \text{maxtail}(k),\, a_{k+1}\right) \\
\text{maxsum}(k+1) &= \max\{\sum_{l=i}^{j} a_l \mid 1 \le i \le j \le k+1\} \\
&= \max\left(\{\sum_{l=i}^{j} a_l \mid 1 \le i \le j \le k\} \cup \{\sum_{l=i}^{k+1} a_l \mid 1 \le i \le k+1\}\right) \\
&= \max(\text{maxsum}(k),\, \text{maxtail}(k+1))
\end{aligned}$$

Durch Induktion über k sind die Formeln somit für alle Arraygrößen korrekt. Der Ablauf des Algorithmus ist in Abbildung 1.6 schematisch dargestellt.

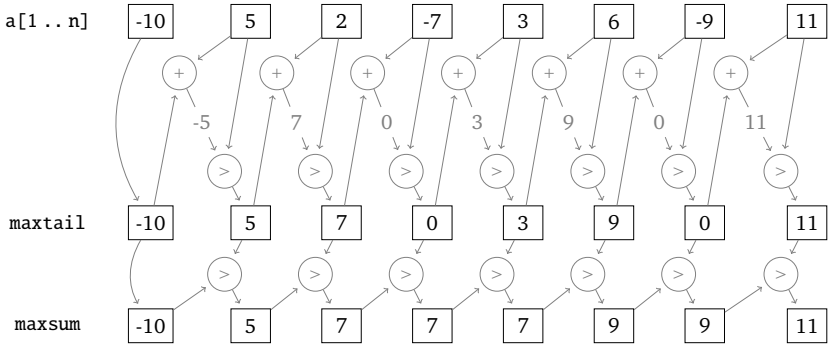


Abbildung 1.6: Schematischer Ablauf des *scanline* Algorithmus.
Die eingerahmten Werte sind Eingaben, relevante Zwischenergebnisse und Ausgaben. Der „+"-Knoten steht für eine Addition, der „>"-Knoten für die Berechnung des Maximums (durch einen oder mehrere Vergleiche). Die Werte von `maxtail` werden aus dem Vorgängerwert von `maxtail` und einem zugehörigen Eintrag des Arrays a durch eine Addition und einen Vergleich berechnet. Die Werte von `maxsum` werden aus ihrem Vorgängerwert von `maxsum` und einem zugehörigen Wert von `maxtail` durch einen Vergleich berechnet. Aufgrund dieser gerichteten Struktur müssen `maxtail` und `maxsum` nicht als Arrays im Speicher gehalten werden, sondern es genügt auf zwei Speicherplätzen (vgl. die Variablen `maxtail` und `maxsum` des Pseudocodes) zu operieren.

Die Laufzeit des *scanline* Algorithmus kann am Pseudocode abgelesen werden. Die Auswertung der Maximumsfunktion ist ein Vergleich. Pro Durchlauf der Schleife werden also zwei Vergleiche und eine Addition durchgeführt. Für die Laufzeit ergibt sich somit:

$$A_4(n) = n-1 \qquad V_4(n) = 2\cdot(n-1)$$
$$T_4(n) = A_4(n) + V_4(n) = 3n-3$$

Der Algorithmus braucht weniger als dreimal so viele Operationen wie das Array Elemente hat. Solche Algorithmen sind für viele Anwendungen wünschenswert, da sie die schöne Eigenschaft haben, auch für große Instanzen mit vielen Elementen schnell Lösungen errechnen zu können. Zur Lösung eines Problems mit 10facher Eingabelänge benötigt man nur einen 10mal so leistungsfähigen Rechner oder 10mal soviel Zeit.

Zusammenfassung

Die Beispiele zeigen, dass verschiedene Lösungsstrategien sich stark auf den Resourcenverbrauch (Anzahl Zeitschritte oder Anzahl Speicherstellen) eines Algorithmus auswirken können.

Hier noch einmal eine Übersicht der berechneten Laufzeiten:

naiver Algorithmus:	$T_1(n) = \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n - 1$
normaler Algorithmus:	$T_2(n) = n^2 - 1$
divide & conquer Algorithmus:	$T_3(n) = 2n \cdot \log_2 n - n + 1$
cleverer Algorithmus:	$T_4(n) = 3n - 3$

Welche Folgen diese Ergebnisse auf praktische Anwendungen haben, lässt sich am besten durch den Vergleich der Laufzeiten einiger repräsentativer Eingabelängen veranschaulichen:

Tabelle 1.3: Laufzeitvergleich der MaxSum-Algorithmen

	naiv	normal	divide & conquer	clever
n	$\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n - 1$	$n^2 - 1$	$2n \cdot \log_2 n - n + 1$	$3n - 3$
$2^2 = 4$	19	15	13	9
$2^4 = 16$	814	255	113	45
$2^6 = 64$	45759	4095	705	189
$2^8 = 256$	2829055	65535	3841	765
$2^{10} = 1024$	179418599	1048575	19457	3069
$2^{15} = 32768$	$> 5 \cdot 10^{12}$	$\approx 10^9$	950273	98301

Beispielsweise benötigt der *clevere* Algorithmus weniger Zeit für die Lösung des Maxsummenproblems auf einem Array mit $n = 1024$ Elementen als der *divide & conquer* Algorithmus für ein Array mit $n = 256$ Elementen und der normale Algorithmus für die Eingabelänge $n = 64$.

Kapitel 2

Komplexitätstheorie

Nach der Analyse des Max-Teilsummenproblems mitsamt den vorgestellten Algorithmen stellen sich ein paar drängende Fragen:

- Im Vergleich der Algorithmen im Einführungskapitel hat sich gezeigt, dass Laufzeit von Algorithmen sich dramatisch unterscheiden können. Muss man Analysen immer so detailiert machen und die Anzahl Schritte bis ins letzte auszählen? Das führt ja schon bei 3-Zeilern zu Komplikationen.
- Wieso werden Vergleiche und Additionen gezählt? Könnte man nicht auch Multiplikationen zählen? Warum wird eine Addition behandelt, wie ein Vergleich? Ist das wirklich sinnvoll?
- Wieso ist der Parameter, mit dessen Hilfe wir die Laufzeitfunktion bestimmen, gerade die Anzahl der Summanden? Führt das nicht dazu, dass jeder parametrisiert, was ihm gerade einfällt? Kann man nicht bessere Vergleichbarkeit erreichen?
- Woher weiß man, ob man einen guten Algorithmus gefunden hat?

Komplexität bezeichnet in der Mathematik und Informatik salopp gesagt die „Kompliziertheit“ von Problemen bzw. ihrer Lösungsverfahren. Was genau darunter zu verstehen ist, ist Teil dieses Kapitels. Es dient dazu Grundlagen zu legen, die eine vereinfachte, aber auch vereinheitlichte Analyse von Algorithmen erlauben.

Dieses Kapitel führt in mehreren Schritten die Grundlagen zur Laufzeitanalyse von Algorithmen ein. Im ersten Abschnitt wird die Landau-Notation (oft auch Groß-O-Notation oder O-Kalkül genannt) zur vereinfachten Klassifizierung von Laufzeiten vorgestellt. Der zweite Abschnitt befasst sich mit den Eingabegrößen eines Algorithmus und ihrer Kodierung in Computern.

2.1 Asymptotische Notation

Mit einem Blick auf Tabelle 1.3 sieht man sofort, dass der *clevere* Algorithmus schneller als seine Konkurrenten ist. Bei der Beurteilung sind die Zahlenwerte im Grunde bedeutungslos: man schaut lediglich danach, wie lang die Zahlen etwa sind, bzw. in welcher Spalte die Länge am wenigsten pro Zeile zunimmt. Durch Betrachtung der Laufzeitterme wird schnell klar, dass man sich nicht wirklich für die Konstanten oder Faktoren interessiert, und dass man manche Terme gegenüber anderen vernachlässigen möchte. Entscheidend ist diejenige funktionale Abhängigkeit, die bei großen Problemen die Laufzeit dominiert.

Vor diesem Hintergrund hat es sich als zweckmäßig erwiesen, Algorithmen nach ihrer *asymptotischen Laufzeit* zu klassifizieren. Mathematisch sauber kann man diese Idee wie folgt formulieren:

Definition 2.1. *Sei $g : \mathbb{N} \to \mathbb{R}^+$. Dann bezeichnet die Komplexitätsklasse*

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

die Menge aller Funktionen, die asymptotisch höchstens so schnell wachsen wie g.

Die asymptotische Notation vernachlässigt Konstanten und langsamer wachsende Terme. Für Polynome werden beispielsweise alle niedrigeren Ordnungen ignoriert:

Satz 2.1. *Für ein Polynom $f(n) = a_m \cdot n^m + a_{m-1} \cdot n^{m-1} + \cdots + a_1 \cdot n + a_0$ vom Grad m mit positivem Koeffizienten a_m bestimmt das größte Monom seine Komplexität: $f \in O(n^m)$*

Beweis. Es ist zu zeigen, dass das Polynom für beliebig große Argumente $n > n_0$ bis auf einen beliebigen konstanten Faktor c durch das Monom n^m nach oben abgeschätzt werden kann:

$$\begin{aligned} f(n) &\leq |a_m| \cdot n^m + |a_{m-1}| \cdot n^{m-1} + \cdots + |a_1| \cdot n + |a_0| \\ &\leq \left(|a_m| + |a_{m-1}|/n + \cdots + |a_1|/n^{m-1} + |a_0|/n^m \right) \cdot n^m \\ &\leq \left(|a_m| + |a_{m-1}| + \cdots + |a_1| + |a_0| \right) \cdot n^m \end{aligned}$$

Mit $c = |a_m| + |a_{m-1}| + \cdots + |a_1| + |a_0|$ und $n_0 = 1$ folgt die Behauptung. □

Definition 2.2. *Sei wieder $g : \mathbb{N} \to \mathbb{R}^+$. Dann gibt es folgende weitere Komplexitätsklassen:*

- $\Omega(g) := \{ f : \mathbb{N} \to \mathbb{R}^+ \mid \exists\, c > 0, n_0 \in \mathbb{N}\ \forall\, n \geq n_0 : f(n) \geq c \cdot g(n) \}$ *ist die Menge aller Funktionen, die asymptotisch mindestens so schnell wachsen wie g. Es gilt*

$$f \in \Omega(g) \iff g \in O(f)$$

- $\Theta(g) := \{ f : \mathbb{N} \to \mathbb{R}^+ \mid \exists\, c_0 > 0, c_1 > 0, n_0 \in \mathbb{N}\ \forall\, n \geq n_0 : c_0 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n) \}$ *ist die Menge aller Funktionen, die asymptotisch gleich schnell wachsen wie g. Es gilt*

$$f \in \Theta(g) \iff f \in O(g) \wedge g \in O(f)$$

- $o(g) := \{ f : \mathbb{N} \to \mathbb{R}^+ \mid \forall\, c > 0\ \exists\, n_0 \in \mathbb{N}\ \forall\, n \geq n_0 : f(n) < c \cdot g(n) \}$ *ist die Menge aller Funktionen, die asymptotisch langsamer wachsen als g. Es gilt*

$$f \in o(g) \iff f \in O(g) \wedge f \notin \Theta(g)$$

- $\omega(g) := \{ f : \mathbb{N} \to \mathbb{R}^+ \mid \forall\, c > 0\ \exists\, n_0 \in \mathbb{N}\ \forall\, n \geq n_0 : f(n) > c \cdot g(n) \}$ *ist die Menge aller Funktionen, die asymptotisch schneller wachsen als g. Es gilt*

$$f \in \omega(g) \iff g \in o(f)$$

Statt $f \in O(g)$ wird manchmal auch $f = O(g)$ geschrieben, ebenso für Ω, Θ, o und ω. Die Buchstaben heißen *Landau-Symbole*. Man spricht von der *Landau-Notation* oder auch *O-Notation*.

Beispiele

Ein Gefühl für Komplexitätsklassen entwickelt man am besten an Hand einiger Beispiele:

Sequentielle Suche Sei $f_1(n)$ die Anzahl der Vergleiche bei einer sequentiellen Suche nach dem Maximum in einer Zahlenfolge mit n Elementen. Dann gilt $f_1(n) \in O(n)$, da einmaliges Durchlaufen der Eingabe mit n Vergleichen auskommt. Andererseits muss jeder Algorithmus zumindest einmal jedes Element der Eingabe ansehen, deshalb gilt auch $f_1(n) \in \Omega(n)$. Insgesamt ergibt sich $f_1(n) \in \Theta(n)$, d.h. die Maximumsfunktion hat (echt) lineare Laufzeit.

Matrixmultiplikation Seien A und B quadratische $n \times n$ Matrizen, $C = A \cdot B$ und $f_2(n)$ die Anzahl der für die Multiplikation nötigen elementaren Operationen. Die Einträge von C ergeben sich aus $c_{i,j} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$, also werden n Multiplikationen und $n-1$ Additionen benötigt. Da n^2 viele Einträge von C berechnet werden, ergibt sich einerseits für den Gesamtaufwand des „offensichtlichen“ Algorithmus $f_2(n) \leq n^2 \cdot (n+n-1) = 2n^3 - n^2 \in O(n^3)$, und andererseits wird jeder Algorithmus $f_2(n) \in \Omega(n^2)$ Operationen zum Lesen der Eingabe benötigen, da er ja n^2 Einträge befüllen muss. Der schnellste zur Zeit bekannte Algorithmus benötigt $O(n^{2.376})$ Operationen.

Einige Ausdrücke in Landau-Notation lassen sich intuitiv bestätigen oder ausschließen, indem man die Bedeutung ausschreibt. Die mathematische Formulierung folgt aus der Definition:

$n \in o(n^2)$ „ n wächst (asymptotisch) langsamer als n^2 “
$\forall\, c > 0\, \exists\, n_0 \in \mathbb{N}\ \forall\, n \geq n_0 : n < c \cdot n^2$, zum Beispiel $n_0 = \left\lceil \frac{2}{c} \right\rceil$

$n^2 \in O(n^2)$ „ n^2 wächst (asymptotisch) nicht schneller als n^2 “
$\exists\, c > 0, n_0 \in \mathbb{N}\ \forall\, n \geq n_0 : n^2 \leq c \cdot n^2$, zum Beispiel $c = 1$ und $n_0 = 1$

$n^2 \notin o(n^2)$ „ n^2 wächst (asymptotisch) nicht langsamer als n^2 “
$\neg\, \forall\, c > 0\, \exists\, n_0 \in \mathbb{N}\ \forall\, n \geq n_0 : n^2 < c \cdot n^2$, zum Beispiel $c = 1$

Mit der O-Notation kann man die Laufzeit von Schleifen oft auf einen Blick erfassen:

Einfache Schleifen:

```
for i := 1 to n do
    for j := 1 to n do
        Führe eine Operation aus.
```

$O(n^2)$ Operationen

Schleifen mit mehreren Operationen:

```
for i := 1 to n do
    for j := i+1 to n do
        Führe f(n) Operationen aus.
```

$O(n^2 \cdot f(n))$ Operationen

In den beiden Fällen wurde nur eine O-Abschätzung gegeben, obwohl eine Θ-Abschätzung möglich wäre. In der Regel beschränkt man sich auf eine Abschätzung nach oben, bemüht sich aber, eine möglichst gute obere Schranke anzugeben. Abschätzungen nach oben sind meistens einfacher abzugeben.

Die folgende Tabelle verdeutlich das Wachstum von Funktionen verschiedener Größenordnungen:

Tabelle 2.1: Wachstumsvergleich verschieden komplexer Funktionen

$\log_2 n$	n	$n \cdot \log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

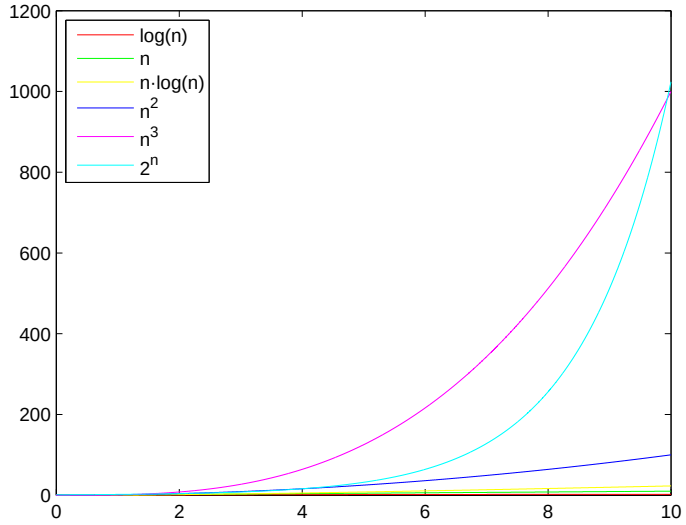


Abbildung 2.1: Funktionsgraphen zum Wachstumsvergleich einiger Funktionen.
Dargestellt sind die sechs Graphen der Funktionen aus Tabelle 2.1 mit gleicher Achsenskalierung. Die Funktionen sind in der Legende nach ihrem asymptotischen Wachstum geordnet. Es deutet sich an, dass die Größenordnung der Funktionswerte schon für relativ kleine Argumente stark variiert. Verläufe der Monome liegen ihrer Ordnung nach sortiert übereinander, logarithmische Funktionen wachsen langsamer als Polynome, exponentielle Funktionen wachsen schneller als jedes Polynom.

Die Relation $o(\dots)$ kann zur Klassifikation von Funktionen benutzt werden.
Zum Beispiel gilt für $0 < \epsilon < 1 < c$:

1	$= o(\log\log n)$	konstante Funktionen
$\log\log n$	$= o(\log n)$	doppelt logarithmische Funktionen
$\log n$	$= o(n^{\epsilon})$	logarithmische Funktionen
n^{ϵ}	$= o(n^{c})$	Wurzelfunktionen
n^{c}	$= o(n^{\log n})$	Polynome
$n^{\log n}$	$= o(c^{n})$	subexponentielle Funktionen
c^{n}	$= o(n^{n})$	exponentielle Funktionen
n^{n}	$= o(c^{c^{n}})$	überexponentielle Funktionen

Rechenregeln

Für die O-Notation gibt es Rechenregeln, von denen hier nur die wichtigsten angegeben werden.
Seien f und g Funktionen und c eine Konstante, dann gilt:

- $c \in O(1)$
- $c \cdot f(n) \in O(f(n))$
- $O(f) + O(f) \subseteq O(f)$
- $O(O(f)) = O(f)$
- $O(f) \cdot O(g) \subseteq O(f \cdot g)$
- $O(f + g) = O(\max\{f(n), g(n)\})$

Die letzte Regel läßt sich folgendermaßen einsehen:

Satz 2.2. *Für Funktionen f und g gilt die Landau-Regel*

$$O(f + g) = O(\max\{f(n), g(n)\})$$

Beweis. Die Mengengleichheit wird durch beide Inklusionen gezeigt.
Sei $h \in O(f + g)$, dann gibt es positive Konstanten c und n_0, sodass für alle $n \geq n_0$ gilt:

$$h(n) \leq c \cdot (f + g)(n) \leq c \cdot 2 \cdot \max\{f, g\}(n)$$

Also ist $h(n) \in O(\max\{f, g\})$.
Sei andererseits $h \in O(\max\{f, g\})$, dann gibt es positive Konstanten c und n_0, sodass für alle $n \geq n_0$ gilt:

$$h(n) \leq c \cdot \max\{f, g\}(n) \leq c \cdot (f + g)(n)$$

Also ist $h \in O(f + g)$. □

Das Master-Theorem

Bei der Analyse von rekursiven Algoithmen steht man häufig vor dem Problem, dass man aus einem rekursiven Algorithmus nur eine Rekursionsgleichung für die Laufzeit und keine explizite Darstellung ablesen kann. Im Einführungskapitel trat dieser Fall beim *divide and conquer* Algorithmus auf und die Rekurrenz konnte nur durch eine komplizierte Rechnung gelöst werden. Das geht oft auch einfacher.

Das Master-Theorem bietet eine Kochbuch-Lösung in asymptotischer Notation für die meisten für uns relevanten Rekurrenzen. Statt eine komplizierte Rekursionsgleichungen per Hand zu lösen, genügt also in der Regel schon eine Umformung nach folgendem Rezept:

Satz 2.3 (Master-Theorem)**.** *Seien $a \geq 1$ und $b > 1$ Konstanten. Sei $f : \mathbb{N} \to \mathbb{R}^+$ zunächst unbestimmt und $T : \mathbb{N}_0 \to \mathbb{R}^+$ rekursiv definiert durch*

$$T(n) = a \cdot T\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n) \qquad \text{mit } \left\lceil \frac{n}{b} \right\rceil = \left\lfloor \frac{n}{b} \right\rfloor \text{ oder } \left\lceil \frac{n}{b} \right\rceil = \left\lceil \frac{n}{b} \right\rceil$$

Diese Rekurrenz ist in folgenden (aber nicht nur diesen) Fällen lösbar:

1. *Falls gilt $\exists\, \epsilon > 0 : f(n) \in O(n^{\log_b a - \epsilon})$, dann*

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

2. *Falls $f(n) \in \Theta(n^{\log_b a})$, dann gilt*

$$T(n) \in \Theta(n^{\log_b a} \cdot \log n)$$

3. *Falls $\exists\, \epsilon > 0 : f(n) \in \Omega(n^{\log_b a + \epsilon})$ und*
   $\exists\, 0 < c < 1, n_0 \in \mathbb{N}\ \forall\, n > n_0 : a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$, dann gilt

$$T(n) \in \Theta(f(n))$$

Beweis. Siehe *Introduction to algorithms* [2], Kapitel 4.3 und 4.4. □

Auf den ersten Blick wirken die vorhergesagten Fälle zu speziell, aber an einigen Beispielen sieht man schnell die Stärke dieses Satzes ein:

- $T(n) = 9 \cdot T\left(\left\lceil \frac{n}{3} \right\rceil\right) + n$, dann ist $a = 9$, $b = 3$ und $f(n) = n$. Es folgt $n^{\log_b a} = n^{\log_3 9} = n^2$ und es gilt $f(n) \in O(n^{\log_3 9 - \epsilon})$. Aus Fall 1 folgt schließlich $T(n) \in \Theta(n^2)$
- $T(n) = T\left(\left\lceil \frac{2n}{3} \right\rceil\right) + 1$, dann ist $a = 1$, $b = \frac{3}{2}$ und $f(n) = 1$. Es folgt $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$ und es gilt $f(n) \in \Theta(1)$. Aus Fall 2 folgt schließlich $T(n) \in \Theta(\log n)$

Aber Achtung! Das Mastertheorem deckt bei weitem nicht alles ab. Der erste und der zweite Fall, sowie der zweite und der dritte Fall unterscheiden sich stark voneinander. Genauer gesagt fehlt zwischen den einzelnen Vorausssetzungen jeweils ein Faktor von n^{ϵ}.

2.2 Kodierungsschemata

Obwohl schon viel über Probleme gesprochen wurde, fehlt nach wie vor eine formale Beschreibung:

Definition 2.3. *Eine binäre Relation zwischen einer Menge I von Instanzen und einer Menge S von Lösungen heißt Problem.*

Am Beispiel des bekannten Maxsummenproblems lautet die Formulierung:

Problem Finde die größte Teilsumme eines Arrays.

Eingabe Ein Array ganzer Zahlen $a_1, \ldots, a_n$ und die Funktion $f(i,j) \coloneqq a_i + \cdots + a_j$.

Ausgabe Das maximale $f(i,j)$.

Die Menge der Instanzen besteht hier aus allen Folgen ganzer Zahlen mit endlicher Länge. Die Menge der Lösungen ist die Menge der ganzen Zahlen. Durch die Definition von f und der Forderung nach einem maximalen $f(i,j)$ werden den Zahlenfolgen endlicher Länge maximale Summen zugeordnet.

Man unterscheidet das abstrakte Problem von der konkreten Beschreibung des Problems und seiner Instanzen. Zur Bearbeitung des Problems auf einem Computer benötigt man beispielsweise eine geeignete *Kodierung*. Dazu nutzt man ein so genanntes Alphabet mit Symbolen bzw. Buchstaben, und man definiert Regeln, welche Bedeutung Verknüpfungen der Symbole bzw. Buchstaben haben. Alphabet und Regeln bilden ein *Kodierungsschema*.

Einige übliche Kodierungen mit resultierender Kodierungslänge, die in der Vorlesung ADM genutzt werden, sind:

Ganze Zahlen Eine ganze Zahl n wird in Binärdarstellung

$$n = \pm \sum_{i=0}^{k} x_i \cdot 2^i \qquad x_i \in \{0,1\} \text{ und } k = \lfloor \log_2(|n|) \rfloor$$

durch eine Bitfolge dargestellt. Die *Kodierungslänge* bezeichnet

$$\langle n \rangle = \lceil \log_2(|n|+1) \rceil + 1 = \lceil \log_2(|n|) \rceil + 2$$

Rationale Zahlen Für eine rationale Zahl r existiert eine Bruchdarstellung

$$r = \frac{p}{q}$$

mit einer ganzen Zahl p und einer natürlichen Zahl q. Die Kodierungslänge ist dann

$$\langle r \rangle = \langle p \rangle + \langle q \rangle$$

Vektoren Ein Vektor $x \in \mathbb{Q}^n$ in Komponentendarstellung

$$x = (x_1, \ldots, x_n)$$

hat die Kodierungslänge

$$\langle x \rangle = \sum_{i=1}^{n} \langle x_i \rangle$$

Matrizen Eine Matrix $A \in \mathbb{Q}^{m \times n}$ in Komponentendarstellung

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

hat die Kodierungslänge

$$\langle A \rangle = \sum_{i=1}^{m} \sum_{j=1}^{n} \langle a_{ij} \rangle$$

Die Summe der Kodierungslängen bestimmt den Speicherverbrauch:

Definition 2.4. *Die Anzahl der Bits, die zur vollständigen Beschreibung einer Instanz I benötigt werden, heißt Inputlänge $\langle I \rangle$.*

Ein Problem zu lösen, also allen Instanzen des Problems Lösungen zuzuordnen, kann unterschiedlich schwierig sein. In sehr komplizierten Fällen ist dies beweisbar nicht möglich. Ohne auf die Theorie der Unentscheidbarkeit eingehen zu wollen, wird an dieser Stelle nur ein Beispiel eines nicht lösbaren Problems gegeben, welches große Auswirkungen auf die Verifizierbarkeit von Algorithmen hat:

Random Access Machine Gegeben ist eine Kodierung einer Random Access Maschine (RAM, das entspricht in etwa einem herkömmlicher Computer mit unendlich viel Speicher), sowie ein $w \in \Sigma$. Frage: Hält die RAM bei Eingabe w?

Definition 2.5. *Ein Problem, für den es keinen Algorithmus gibt, der für alle Instanzen des Problems die richtige Antwort ausgibt, heißt nicht entscheidbar.*

Wichtig: Im Rahmen dieser Vorlesung sind alle Probleme lösbar. Die Frage ist nur in welcher Zeit und mit wieviel Speicherplatz.

Bevor wir uns an die Beantwortung der Frage machen, was im Sinne der Komplexitätstheorie ein schwieriges Problem ist, kann man ja erstmal nach der intuitiven Schwierigkeit von Problemen fragen. Was ist wohl schwieriger: Kopfrechnen, das Lösen eines Solitär-Brettspiels, den richtigen Zug in einer Schachstellung zu finden, den richtigen Zug beim Sokobanspiel zu finden, oder Kreuzworträtsel zu lösen?

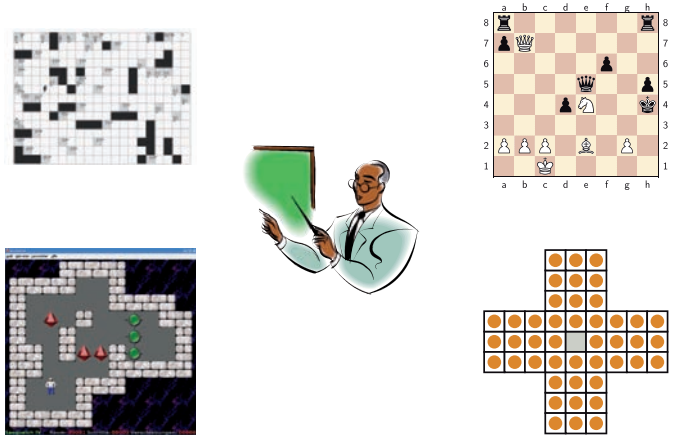


Abbildung 2.2: Grafiken einiger Spiele.
Im täglichen Leben muss man häufig komplizierte Denkleistungen erbringen und viele davon tarnen sich als Spiele. Wenn man einen Menschen mit den genannten Aufgaben konfrontiert, wird er einige davon als schwieriger empfinden als andere. Aber wie beurteilt man in der Mathematik, wie schwierig ein Spiel ist?

2.3 Algorithmen

Nachdem geklärt ist, was unter dem Begriff des Problems verstanden werden soll, gilt es nun möglichst formal zu klären, was ein Algorithmus ist.

Definition 2.6. *Eine Anleitung zur schrittweisen Lösung eines Problems heißt Algorithmus. Man sagt, ein Algorithmus A löst ein Problem Π, falls A für alle Instanzen $I \in \Pi$ des Problems eine Lösung in einer endlichen Anzahl von Schritten liefert.*

Der Begriff *Schritt* meint eine elementare Operation. Eine allgemeine Definition dafür gibt es aber nicht, da der verfügbare Befehlssatz im Allgemeinen auch von einer Maschine abhängt, auf der ein Algorithmus ausgeführt wird. Nun möchte man natürlich nicht Laufzeitanalysen für verschiedene Maschinen und Befehlssätze berechnen und tabellieren. Deshalb betrachtet man die Algorithmen auf einem *Rechnermodell*, das die relevanten Aspekte eines realen Rechners nachbildet, das sich aber deutlich leichter beschreiben lässt.

Ein Beispiel für ein solches Rechnermodell ist die *Registermaschine* (RAM, *random access machine*). Sie besteht aus einem Programm, einem Befehlszähler, einem Akkumulator und einem Speicher mit unendlich vielen, durchnummerierten Zellen. Die Zellen werden auch als Register bezeichnet und können beliebig große, natürliche Zahlen speichern. Der Befehlszähler b gibt an, welche *Programmzeile* gerade ausgeführt wird und der Akkumulator c bezeichnet ein spezielles Register, auf dem das Programm lesen und schreiben kann, und mit dessen Hilfe elementare Berechnungen durchgeführt werden können.

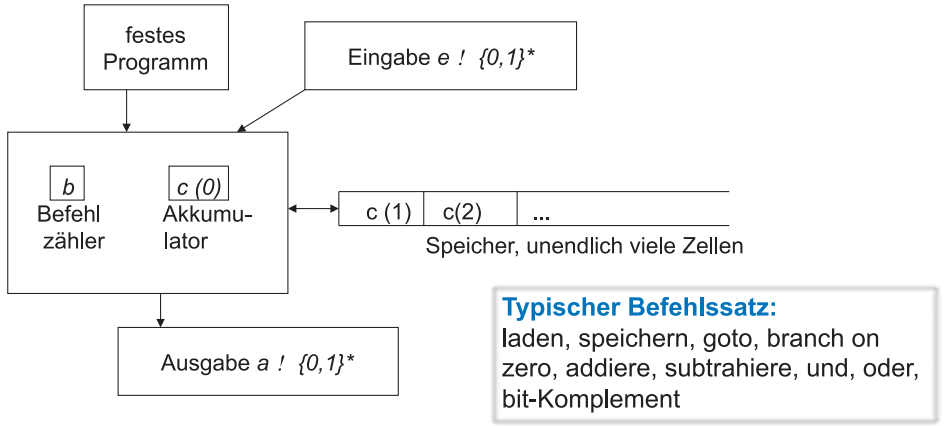


Abbildung 2.3: Schematischer Aufbau einer Registermaschine (RAM).
Die Registermaschine ist ein theoretisches Rechnermodell, das einem realen Computer ähnlich ist. Sie besteht aus einem Programm, einem Befehlszähler, einem Akkumulator und einem Speicher mit unendlich vielen, durchnummerierten Zellen. Beim Abarbeiten eines Programm werden Daten aus der Eingabe gelesen und in die Ausgabe geschrieben.

Dem Programm wird ein binärer Eingabetext e übergeben und es generiert daraus in endlich vielen Schritten einen binären Ausgabetext a. Normalerweise werden beide realisiert, indem man die Eingabe vor der Programmausführung in den Speicher legt und nach der Ausführung das Ergebnis aus dem Speicher ausliest.

Man unterscheidet zusätzlich zwei Modelltypen:

unit-cost Modell Jeder Befehl wird in einem Schritt abgearbeitet.

- **Typischer Befehlssatz** Grundrechenarten (+, -, *, /), Vergleiche (=, <, >), Lesen und Schreiben rationaler Zahlen, Programmfluss mittels Verzweigungen und Schleifen.
- **Einsatzgebiet** Dieses Modell ist einfach und wird meistens verwendet, insbesondere in der Vorlesung ADM fast ausschließlich.

log-cost Modell Jeder Befehl benötigt $\Theta(f(k))$ Zeit, wobei k die Anzahl der Bits der Operanden und f eine Kostenfunktion für die einzelnen Elementaroperationen ist.

- **Typischer Befehlssatz** Grundrechenarten (+, -, *, /), Bitweise Operationen (complement, and, or), Sprünge (goto, branch), Laden und Speichern.
- **Einsatzgebiet** Dieses Modell ist realistischer und wird für komplexere Probleme relevant. Es wird in der Optimierung wichtig, wenn es um die Laufzeitanalyse von Lösungsverfahren von Linearen Programmen geht.

2.3.1 Effizienzmaße

Im Einführungskapitel wurde eine Anzahl von Vergleichen und Additionen gezählt, und die Laufzeit, die aus Additionen und Vergleichen bestand, wurde in einem bestimmten Parameter gemessen, nämlich der Anzahl von vorgegebenen Zahlen in der Eingabe. Das geschah willkürlich, und bei anderen Problemen würde man sicher anders verfahren. In der Theoretischen Informatik hat man aber etwas gefunden, das es uns erlaubt die Analysen verschiedenster Algorithmen miteinander vergleichbar zu halten.

Man drückt dafür die Laufzeit von Algorithmen als Funktion ihrer Eingabelänge aus! Das ist ein echter Supertrick, denn zum einen ermöglicht es eine einheitliche Basis zur Algorithmenanalyse, zum anderen kommt auch niemand an den tiefgreifenden Erkenntnissen der Komplexitätstheorie vorbei. Jeder, der ein Programm scheibt, in welcher Programmiersprache auch immer (funktional, objekt-orientiert, prozedural ...), wird zu irgendeinem Zeitpunkt seine Eingaben kodieren müssen, und diese werden eine Länge haben. Es wird eine kodierte Ausgabe geben müssen und dazwischen eine Transformation der Eingabe hin zur Ausgabe. Dazu werden Berechnungen angestellt werden, die eine Laufzeit haben. Und wenn nun die Komplexitätstheorie zu einem Problem Π sagt, dass es keinen Algorithmus mit Laufzeit $O(n)$ gibt, dann wird es sicher keinen geben.

Die Laufzeit eines Algorithmus hängt im Allgmeinen allerdings nicht nur von der Eingabelänge ab, sondern kann abhängig vom Wert der Eingabe abweichen. In diesem Fall hat es sich eingebürgert, dass man aus den Laufzeiten aller Eingaben der Länge n einen repräsentativen Kennwert erstellt. In der Regel interessiert man sich für die schlechteste (*worst case*), die mittlere (*average*) oder die kürzeste (*best case*) Laufzeit, ermittelt über alle Eingaben einer bestimmten Länge.

Definition 2.7. *Sei $T_A(x)$ die Anzahl der Befehle, die Algorithmus A bei Eingabe x ausführt. Dann kann man seine Effizienz wie folgt charakterisieren:*

worst case Laufzeit $T_A^{wc}(n) := \max\{\, T_A(x) \mid \langle x \rangle \leq n \,\}$

average case Laufzeit $T_A^{ac}(n) := \sum_{\{\, x \mid \langle x \rangle = n \,\}} p_x \cdot T_A(x)$

(erfordert Kenntnis von Auftrittswahrscheinlichkeiten oder Annahme von Gleichverteilung)

best case Laufzeit $T_A^{bc}(n) := \min\{\, T_A(x) \mid \langle x \rangle \leq n \,\}$

Wenn man statt der Laufzeit den Platzbedarf analysiert, ergeben sich analoge Definitionen. Ist $S_A(x)$ die größte Adresse im Speicher, die A bei Eingabe x benutzt, so ist beispielsweise $S_A^{\text{wc}}(n) := \max\{\, S_A(x) \mid \langle x \rangle \leq n \,\}$.

Die so genannte *Komplexität* eines Problems und die *Komplexität eines Algorithmus* wird wie folgt definiert:

Definition 2.8 (Komplexität eines Algorithmus)**.** *Sei A ein deterministischer (RAM-)Algorithmus, der auf allen Eingaben hält. Die Laufzeit (Zeitkomplexität) von A ist eine Funktion $f : \mathbb{N} \to \mathbb{N}$, wobei $f(n)$ die maximale Anzahl (worst case) an Schritten von A auf einer Eingabe der Länge n ist.*

Definition 2.9 (Komplexität eines Problems)**.** *Die Laufzeit desjenigen Algorithmus mit kleinster Zeitkomplexität, der Problem P löst, heißt Zeitkomplexität des Problems. Der Speicherplatzbedarf des am wenigsten Speicherzellen benutzende Algorithmus, der Problem P löst, heißt Platzkomplexität des Problems.*

Um sich gegenseitig schneller über Probleme, Algorithmen und Laufzeiten verständigen zu können, werden üblicherweise noch die Begriffe Linearzeit, Polynomialzeit, und ähnliche verwendet.

- Für einen Linearzeitalgorithmus A gilt: $f_A(n) \leq c \cdot n$ für eine Konstante c
- Für einen Polynomzeitalgorithmus A gilt: $f_A(n) \leq c \cdot n^k$ für Konstanten c und k
- Ein Problem P ist „in Polynomzeit lösbar“, wenn es einen Algorithmus A, ein Polynom Π gibt, sodass $f_A(n) \leq \Pi(n)$

Man setzt dabei immer hinreichend große n voraus, auch wenn dies nicht explizit erwähnt wird.

Es folgen zwei weitere algorithmische Beispiele, an denen die neu eingeführten Begrifflichkeiten gezeigt werden.

Beispiel: Inkrementierung einer binär dargestellten Zahl

Betrachten wir eine Addition von 1 auf eine andere Zahl:

Problem Addition um 1 im Binärsystem.

Eingabe Binärdarstellung $x_{n-1} \dots x_0$ von x

Ausgabe Binärdarstellung von $x + 1$

Ein Algorithmus für dieses Problem könnte im Pseudocode so aussehen:

```
falls (x[n-1] ... x[0]) == (1 ... 1)
    Gib Ergebnis (1 0 ... 0) aus.
sonst
    Suche kritische Position, d.h. das kleinste i mit x[i] = 0.
    Gib bearbeitete Eingabe (x[n-1] ... x[i+1] 1 0 ... 0) aus.
```

Die üblichen Effizienzmaße ergeben dann folgende Ausdrücke:

Laufzeitanalyse Berechnet wird die Anzahl der veränderten Bits. (Das entspricht der Laufzeit einer Maschine, deren einzige elementare Operation ein *bit-flip* ist.)

worst case Bei Eingabe von $(11\ldots11)$ $\quad f^{\text{wc}}(n) = n$

best case Bei Eingabe von z.B. $(11\ldots10)$ $\quad f^{\text{bc}}(n) = 1$

average case Verschiedene Anzahlen von Operationen werden mit gewissen Auftrittswahrscheinlichkeiten (Gleichverteilung über alle möglichen Eingaben) gewichtet:

- 1 Operation mit Wahrscheinlichkeit $\frac{1}{2}$
- 2 Operationen mit Wahrscheinlichkeit $\frac{1}{4}$
- 3 Operationen mit Wahrscheinlichkeit $\frac{1}{8}$
- $\ldots$
- n Operationen mit Wahrscheinlichkeit $\frac{1}{2^n}$

$$\Rightarrow f^{\text{ac}}(n) = 1\cdot\frac{1}{2} + 2\cdot\frac{1}{4} + 3\cdot\frac{1}{8} + \cdots = \sum_{i=0}^{n-1}\frac{1}{2^{i+1}}\cdot(i+1) \le \sum_{i=0}^{\infty}\frac{1}{2^{i+1}}\cdot(i+1) = 2$$

Bei diesem Algorithmus liegt die *average case* Laufzeit nahe bei der *best case* Laufzeit. Es gibt aber auch Beispiele, wo der *average case* nahe beim *worst case* oder auch mittig zwischen den Fällen liegt.

Beispiel: Fibonacci-Zahlen

Als nächstes soll ein Vergleich zwischen der Laufzeit als Funktion der Eingabe und der Laufzeit als Funktion der Eingabegröße verdeutlicht werden:

Problem Berechnung der n-ten Fibonacci-Zahl.

Eingabe Natürliche Zahl n mit Kodierung $k = \langle n\rangle$

Ausgabe Laufzeit in Abhängigkeit von n oder k

Zur Erinnerung: Die Fibonacci-Folge $(F_i)_{i\in\mathbb{N}_0}$ ist rekursiv definiert. Die ersten beiden Fibonacci-Zahlen sind $F_0 = 0$ und $F_1 = 1$ und alle weiteren Folgeglieder ergeben sich als Summe ihrer beiden Vorgänger $F_n = F_{n-1} + F_{n-2}$. Ein erster (sehr langsamer) Algorithmus hat somit folgende Form:

```
fib(n):
    falls n <= 1
        Gib n aus.
    sonst
        Gib fib(n-1) + fib(n-2) aus.
```

Mit jedem zusätzlichen Folgeglied verdoppelt sich die Zahl der ausgeführten Operationen. Also ist $T(n) = O(2^n)$, wobei n der Index der Fibonacci-Folge ist. Die Kodierung dieses Index hat logarithmische Länge in n, also $k = \langle n\rangle$. Dann ist die Laufzeit $T(k) = 2^{2^k}$.

Ein zweiter Algorithmus geht wie folgt vor. Er errechnet erst F_2 aus F_0 unf F_1, dann F_3 aus den beiden Vorgängern u.s.w.

```
f0 := 0
f1 := 1
for i := 2 to n do
    tmp := f1
    f1  := f1 + f0
    f0  := tmp
falls n == 0
    Gib f0 aus.
sonst
    Gib f1 aus.
```

Nach einer kurzen Initialisierung sind zur Berechnung jeder Fibonacci-Zahl drei zusätzliche Operationen nötig. Die Laufzeit als Funktion der Eingabezahl ist also $T(n) = O(n)$ und die Laufzeit als Funktion der Eingabelänge mit $k = \langle n\rangle$ ist $T(k) = O(2^k)$.

Wie schon beim Maxsummenproblem gibt es auch beim Fibonacciproblem einen noch effektiveren Algorithmus mit etwas trickreicherer Herangehensweise. Zunächst stellt man fest, dass die Berechnung der Folgeglieder als lineares Gleichungssystem geschrieben werden kann:

$$\begin{pmatrix} F_{k+2} \\ F_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix} \qquad \text{bzw.} \qquad \vec{F}_{k+1} = A\cdot\vec{F}_k$$

Durch wiederholtes Anmultiplizieren von A lassen sich weitere Folgeglieder berechnen. Da die Spalten von A als $\vec{F}_1$ und $\vec{F}_0$ interpretiert werden können, folgt eine kompakte Darstellung:

$$\vec{F}_{k+n} = A^n\cdot\vec{F}_k \qquad \text{und} \qquad \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

Ein Algorithmus könnte die n-te Fibonacci-Zahl folglich als $(A^n)_{1,2}$ finden. Der Übergang zur Binärdarstellung des Index ermöglicht eine sehr schnelle Berechnung der einzelnen Potenzen:

$$n = \sum_{i=0}^{k} x_i\cdot 2^i \qquad \text{mit} \quad x_i \in \{0,1\} \text{ und } k = \lfloor\log_2(|n|)\rfloor$$

Durch Einsetzen zerfällt die Potenzierung in Teilprodukte:

$$A^n = A^{\sum_{i=0}^{k} x_i\cdot 2^i} = \prod_{i=0}^{k} A^{x_i\cdot 2^i} = \prod_{\substack{x_i=1 \\ 0\le i\le k}} A^{2^i}$$

Beispielsweise gilt für eine Eingabe $m = 13_{10} = 1001_2$, dass $A^{13} = A^{1+4+8} = A \cdot A^4 \cdot A^8$ ist. Die Berechnung dieser Potenzen erfolgt dann durch wiederholtes Quadrieren:

$$A^{2^i} = \overbrace{A^{(2\cdot\cdots\cdot 2)}}^{O(k)-\text{mal multiplizieren}} = \overbrace{\left(\ldots\left(A^2\right)^2\ldots\right)^2}^{O(k)-\text{mal quadrieren}}$$

Der Aufwand für die Potenzbildungen ist also $O(k)$ und der Aufwand für das Aufmultiplizieren ebenso. Die Laufzeit dieses Algorithmus ist somit $T(k) = O(k)$. Aber **Vorsicht**! Unsere Analyse beruht auf dem Unit-cost-Modell!!

Wenn man aber das log-cost-Modell heranzieht, ist die Überlegenheit des letzten Algorithmus gegenüber dem zweiten Algorithmus gar nicht mehr klar: Es lässt sich nämlich beobachten, dass die Fibonacci-Zahlen exponentiell mit dem Eingabeparameter n wachsen, und damit wachsen auch die Längen der Zahlen, die bei den Berechnungen auftreten, entsprechend an. Wie lang werden die Binärdarstellungen der Zahlen? Nun, wenn $F_n = \Theta(k^n)$ ist, für irgendein $k > 1$, dann ist $F_n = \Theta(2^{2^k})$ und damit $< F_n >= \Theta(2^k)$. Das heißt, man benötigt zwar nur wenige arithmetische Operationen im dritten Algorithmus, aber die einzelnen Operationen werden auch entsprechend aufwendig.

Ob tatsächlich der dritte Algorithmus schneller ist als der zweite, wenn man sie als Programme auf einem Computer ablaufen lässt, kann also anhand unserer Analyse nicht vorhergesagt werden; auch nicht, wenn wir von „hinreichend großen Eingabedaten“ ausgehen.

2.4 Komplexitätsklassen

Zunächst werden zwei spezielle Typen von Problemen spezifiziert:

Definition 2.10. *Ein Problem, das nur zwei mögliche Antworten („ja“ und „nein“) besitzt, heißt Entscheidungsproblem.*

Definition 2.11. *Ein Problem, bei dem aus einer (explizit oder implizit beschriebenen) Menge möglicher Lösungen Ω ein bezüglich einer Bewertungsfunktion $f : \Omega \to \mathbb{R}$ bestes Element $\bar{x}$ mit $f(\bar{x}) = \max\{ f(x) \mid x \in \Omega \}$ gesucht wird, heißt Optimierungsproblem.*

Beispiele für Entscheidungsprobleme sind „Ist n eine Primzahl?“ oder „Gibt es einen Lösungsweg beim Solitär-Brettspiel?“, ein Optimierungsproblem ist z.B. „Finde einen bestmöglichen Flugplan (bezüglich des Gewinns der Fluggesellschaft)“. Entscheidungsprobleme und Optimierungsprobleme lassen sich ineinander umformulieren.

2.4.1 Die Klassen P, NP, PSPACE

Mit Hilfe der O-Notation können wir Probleme in Klassen aufteilen. Man könnte z.B. die Menge aller Probleme mit Komplexität $O(n^3)$ zusammenfassen und sagen, das sei eine Klasse. Eine Klasse ist also eine Menge von Mengen. Oder man könnte die Menge aller Probleme so aufteilen, dass alle Probleme, die in Polynomzeit lösbar sind in einer Klasse zusammengefasst werden, alle Probleme, die mindestens Exponentielle Laufzeit der Lösungsalgorithmen erfordern in eine andere Klasse kommen und alle anderen Probleme in einer dritten Klasse zusammengefasst werden. Viele Varianten sind denkbar.

Besonderes Interesse gilt den Klassen mit den Namen P, NP, und manchmal auch der Klasse $PSPACE$. Die Bedeutung dieser Kürzel zu klären ist die Aufgabe des Rests dieses Kapitels.

Die Klasse P ist die Menge derjenigen Entscheidungsprobleme, für die es einen Algorithmus gibt, der *worst case* polynomielle Laufzeit besitzt und der das Entscheidungsproblem löst. Man nennt solche Probleme auch „effizient lösbare“ Probleme. Formaler:

Definition 2.12. *Gegeben sei ein Kodierungsschema E und ein Rechnermodell M. Betrachte ein Entscheidungsproblem Π, wobei jede Instanz aus Π durch das Kodierungsschema E kodiert sei. Wenn es einen auf M implementierbaren Algorithmus zur Lösung aller Probleminstanzen aus Π gibt, dessen worst case Laufzeitfunktion auf M polynomiell ist, dann gehört Π (bezüglich E und M) zur Klasse P.*

Beispiel: Solitär

Um ein Gefühl für die Klassen P zu bekommen, betrachten wir zunächst Problemstellungen bei einem verallgemeinerten Solitär-Brettspiel. Man kann sich das Solitärbrett als aus 5 Komponenten mit je 3 Feldern zusammengesetzt vorstellen. Ausserdem liegen in der Startposition auf allen Feldern, bis auf das Feld genau in der Mitte, Steine. Eine einfache Verallgemeinerung geht so, dass man Solitärbretter aus 5 Komponenten mit je $n \times n$ Feldern zusammensetzt. Eine beliebige Teilmenge der Felder kann zu Beginn besetzt sein und somit eine Startposition definieren.

Problem Steine zählen im Solitär.

Eingabe Eine beliebige Startposition im $n \times n$-Solitär

Ausgabe „ja“, falls schon mehr als die Hälfte der Steine vom Brett ist. „nein“, falls noch mindestens die Hälfte übrig ist.

Dieses Problem wird intuitiv als leicht empfunden. Die Größe der Eingabe hängt hauptsächlich von der Anzahl der Felder, und damit von n ab. Man sieht sofort einen Polynomialzeitalgorithmus, der das Problem löst. Das Problem liegt in P.

Problem Züge planen im Solitär.

Eingabe Eine beliebige Startposition im $n \times n$-Solitär

Ausgabe „ja“, wenn es einen Weg gibt, Steine so zu schlagen, dass genau ein Stein in der Mitte des Spielbretts übrig bleibt. „nein“, falls es keinen solchen Weg gibt.

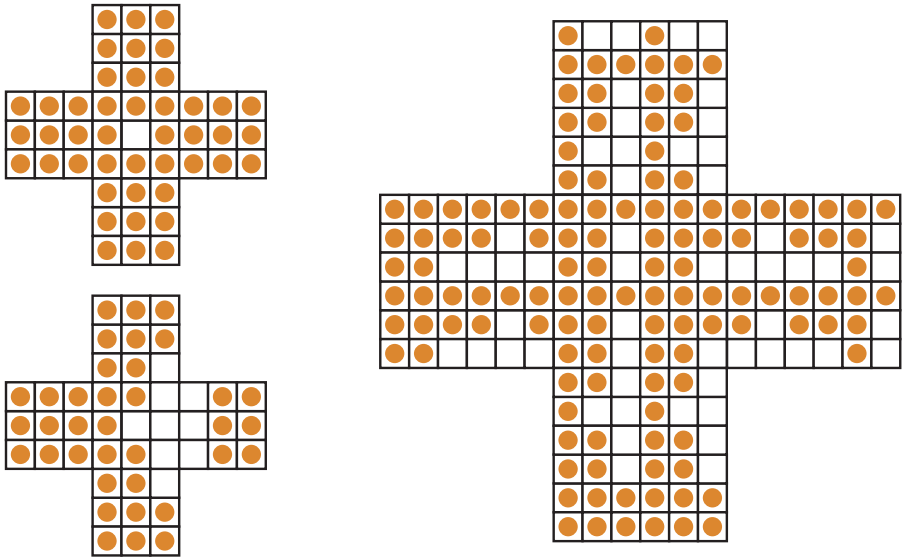


Abbildung 2.4: Grafik einiger Solitär-Bretter.
Die meisten Brettspiele bieten ein gut implementierbares Regelwerk und erlauben das Formulieren vieler Entscheidungsfragen. Wie leicht ein solches Problem zu lösen ist, misst man daran, ob ein Algorithmus existiert, der beliebig große Instanzen des Problems hinreichend schnell oder platzsparend lösen kann.

Dieses Problem hingegen erscheint intuitiv schwieriger zu sein. Immerhin, wenn es einen Weg gibt, und man diesen Weg mit zu der Eingabe hinzutut, läßt sich die Lösung zumindest in Polynomzeit nachvollziehen: Man braucht nur der vorgegebenen Lösung zu folgen. Das Problem hat also eine etwas andere Charakteristik als das reine Zählproblem zuvor. Die Klasse NP ist wie folgt definiert:

Definition 2.13. *Ein Entscheidungsproblem Π gehört zur Klasse NP, wenn gilt:*

- *Für jedes Problembeispiel $I \in \Pi$, für das die Antwort „ja“ lautet, gibt es (mindestens) ein Objekt Q, mit dessen Hilfe die Korrektheit der Antwort „ja“ überprüft werden kann.*
- *Es gibt einen Algorithmus, der Problembeispiele $I \in \Pi$ und Zusatzobjekte Q als Eingabe akzeptiert und dessen Laufzeit polynomiell in $\langle I \rangle + \langle Q \rangle$ ist, der überprüft, ob Q ein Objekt ist, aufgrund dessen eine „ja“-Antwort gegeben werden muss.*
- *Es wird keine Aussage darüber gemacht, wie Q berechnet wird. Q kann geraten werden.*
- *Die einzige Aussage, die über „nein“-Antworten gemacht wird, ist, dass es einen Algorithmus geben muss, der korrekt „ja“ oder „nein“ entscheiden kann.*

Anmerkung: Man kann die Klasse NP auch über eine so genannte nicht-deterministische RAM definieren. Eine solche Maschine ist eine RAM mit dem zusätzlichen Befehl „`goto L1 or goto L2;`“. Da dies die ursprüngliche Definition ist, sie aber ohne eingehende Diskussion über Nicht-determinismus schwer zu verstehen ist, wird sie lediglich der Vollständigkeit halber angegeben. Beide Definitionen sind gleichwertig, und für diese Veranstaltung genügt die erste.

Definition 2.14 (gleichwertig zu Definition 2.13)**.** *Ein Problem Π ist in NP, wenn es einen (möglicherweise nicht-deterministischen) Algorithmus A gibt, so dass es für jedes Problembeispiel $I \in \Pi$, für das die Antwort „ja“ lautet, einen Berechnungsweg von A gibt, der polynomielle Länge in $\langle I \rangle$ besitzt. Für Instanzen, für die die Antwort „nein“ ist, darf A auf keinem Berechnungsweg die Antwort „ja“ ausgeben. Für alle Instanzen muss A irgendwann halten.*

Das zweite Solitärproblem ist also in der Klasse NP. Ob es auch in der Klasse P ist, wissen wir an dieser Stelle nicht. Nur weil man nicht sofort sieht, wie ein Polynomialzeitalgorithmus arbeiten würde, heißt das ja nicht, dass es keinen gibt. In den bisherigen Beispielen waren ja auch die schnellsten Algorithmen nicht die offensichtlichsten.

Das SAT Problem

Seien $x_1, ..., x_n$ boolsche Variablen, die die Werte true (1) oder false (0) annehmen können. Es sollen folgende Gesetzmäßigkeiten für boolsche Ausdrücke gelten: $x \vee y = false$ genau dann, wenn $x = false$ und $y = false$. $x \wedge y = true$ genau dann, wenn $x = true$ und $y = true$. Darüber hinaus soll genau dann $x = true$ gelten, wenn $x \neq false$.

Definition 2.15. *Eine boolesche Funktion $f(x_1, x_2, \ldots, x_n)$, für die es eine Wertebelegung für $x_1, x_2, \ldots, x_n$ gibt, so dass $f(x_1, x_2, \ldots, x_n) = 1$ ist, heißt erfüllbar.*

Zum Beispiel ist die Funktion

$$f(x, y, z) = (x \vee y) \wedge (z \vee \neg x \vee \neg y) \wedge (x \vee \neg z)$$

erfüllbar, denn die Belegung $x = 1, y = 0, z = 0$ liefert

$$f(1, 0, 0) = (1 \vee 0) \wedge (0 \vee \neg 1 \vee \neg 0) \wedge (1 \vee \neg 0) = 1$$

Damit wird das SAT-Problem definiert:

Problem Erfüllbarkeit einer booleschen Funktion.

Eingabe Eine boolesche Funktion $\Phi(x_1, x_2, \ldots, x_n)$.

Ausgabe „ja“, falls es eine Belegung für $x_1, x_2, \ldots, x_n$ gibt, so dass $\Phi(x_1, x_2, \ldots, x_n) = 1$ ist. „nein“, falls keine solche Belegung existiert.

Im Allgemeinen ist ein SAT-Problem natürlich mit exponentieller Laufzeit (in der Anzahl der Variablen) entscheidbar, indem alle Wertebelegungen durchprobiert werden. Es ist nicht bekannt, ob auch polynomielle Algorithmen existieren, jedoch ist die Vermutung, dass es für dieses Problem keinen Polynomzeitalgorithmus gibt. SAT ist so zu sagen die Mutter aller Probleme, von denen man vermutet, dass sie in NP, aber nicht in P liegen.

2.4.2 Abhängigkeiten der Klassen

Die Klassen P und NP decken einen Großteil der in der Literatur vorzufindenden Problemstellungen ab. Weitere interessante Klassen sind $PSPACE$ und $EXPTIME$:

Klasse P Klasse aller Probleme, die von einer deterministischen RAM in Polynomzeit gelöst werden können.

Klasse NP Klasse aller Probleme, die von einer nichtdeterministischen RAM in Polynomzeit gelöst werden können.

Klasse PSPACE Klasse aller Probleme, die von einer deterministischen RAM mit polynomiell viel Platz gelöst werden können.

Klasse EXPTIME Klasse aller Probleme mit einfach exponentieller Laufzeit:
EXPTIME $= \bigcup_k \text{TIME}(2^{n^k})$.

Vieles ist noch ungeklärt über den Zusammenhang zwischen diesen Klassen. Sicher weiss man nur, dass P $\neq$ EXPTIME und

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

Man vermutet darüber hinaus, dass alle diese Inklusionen echt sind, d.h.

$$\text{P} \subset \text{NP} \subset \text{PSPACE} \subset \text{EXPTIME}$$

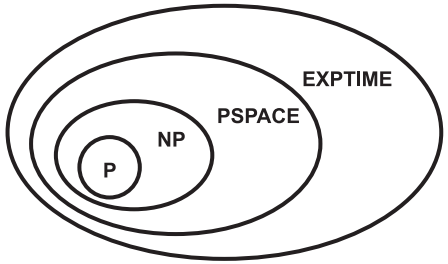


Abbildung 2.5: Schematische Darstellung zur Inklusion der Komplexitätsklassen.

2.4.3 Einordnung von Problemen

Nicht immer lässt sich leicht beurteilen, in welcher der kennengelernten Klassen ein konkretes Problem liegt. Immerhin muss zur Einordnung ein Algorithmus mit hinreichend guter Laufzeit gefunden oder die Existenz eines solchen widerlegt werden. Besonders letzteres ist oft sehr schwer zu zeigen. So hat man es zu noch keinem Problem in $\mathcal{NP}$ hinbekommen, einen Beweis dafür zu finden, dass das Problem nicht in polynomialer Zeit lösbar ist, denn sonst wüsste man ja, dass $\mathcal{P} \neq \mathcal{NP}$.

Hilfreich ist bei der Einordnung von Problemen oft die Technik der *Reduktion*. Die Idee ist es dabei, ein unbekanntes Problem zu einem bereits klassifizierten Problem in Beziehung zu setzen. Wenn man es nämlich schafft, Instanzen eines Problems P so in Instanzen eines anderen Problems Q umzubauen, dass die Antwort zu einer Instanz in P genau dann „ja“ ist, wenn die Antwort in der umgebauten Instanz des Problems Q ebenfalls ja ist, hat man diese beiden Probleme aneinandergekoppelt. Wenn sich der Umbau der Instanzen pro Instanz auch noch in polynomieller Zeit realisieren lässt, sind die beiden Probleme in gewissem Sinne sehr eng aneinander gekoppelt worden. Gibt es nämlich für Q einen effizienten Algorithmus (also einen polynomiellen), dann gibt es auch für P einen effizienten Algorithmus. Andersherum, wenn es für P keinen effizienten Algorithmus gibt, gib es auch keinen für Q.

Der Reduktionsmechanismus sieht im Detail wie folgt aus.

Definition 2.16. *Seien P und Q Probleme und seien L_P und L_Q die Mengen der Instanzen der Probleme P und Q, für die die Antwort „ja“ ist. Sei zudem Σ ein Alphabet zur Problemkodierung und Σ^* die Menge aller Symbolketten, die aus dem Alphabet Σ erzeugt werden können. Wenn es eine von einem deterministischen Algorithmus in Polynomzeit berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ gibt, so dass*

$$x \in L_P \iff f(x) \in L_Q$$

gilt, dann heißt P polynomiell reduzierbar auf Q. Man schreibt $P \leq_p Q$.

Man sagt, P ist nicht schwerer als Q bzw. Q ist mindestens so schwer wie P.

NP-schwierig und NP-vollständig

Im Studium der ADM und der Optimierungsvorlesungen Optimierung I-III werden fast ausschließlich Probleme, die zu den Klassen P und NP gehören, von Relevanz sein. Im Folgenden wird deswegen ein Schwerpunkt auf die so genannte NP-Schwere und die NP-Vollständigkeit gelegt. Schwere und Vollständigkeit können aber auf sämtliche andere Klassen übertragen werden.

Definition 2.17. *Eine Sprache S heißt NP-schwer (oder NP-schwierig, NP-hard), wenn jede Sprache aus NP mit einer Polynomzeit-Reduktion auf S reduziert werden kann, d.h. wenn $\forall\, L \in NP$: $L \leq_p S$.*

NP-schwierige Probleme sind also mindestens so schwer wie andere NP-Probleme.

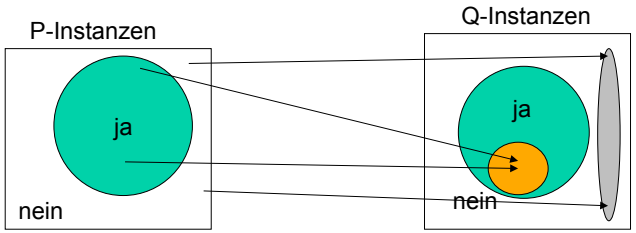


Abbildung 2.6: Reduktion eines Problems.
Ein Problem ist definiert als eine Relation zwischen der Menge seiner Instanzen und ihrer Lösungen (vgl. Abschnitt 2.2). Wenn die Mengen der Instanzen zweier Probleme P und Q in kodierter Form vorliegen, können sie durch Algorithmen ineinander überführt werden. Falls es gelingt einen Algorithmus zu finden, der polynomielle Laufzeit hat und der alle „ja"-Instanzen des Problems P auf „ja"-Instanzen des Problems Q abbildet (und analog „nein"-Instanzen von P auf „nein"-Instanzen von Q), dann kann man das Problem P offenbar lösen, indem man es nach Q reduziert und Q löst. (Frage: was ist mit Symbolketten, die gar keine Kodierungen des Problems P bzw. Q sind?) Da die Reduktion in polynomieller Zeit berechnet wird, wird sich die Komplexität von P im groben Raster der Klassen **P**, **NP**, **PSPACE**, **EXPTIME** nicht nennenswert von der Komplexität von Q unterscheiden.

Satz 2.4. *Falls eine NP-schwierige Sprache in P ist, gilt P=NP.*

Beweis. Wenn $S \in$ P, folgt aus $L \leq_p S$, dass auch $L \in$ P. □

Definition 2.18. *Eine Sprache $S \in$ NP heißt NP-vollständig (NP-complete), wenn S NP-schwierig ist.*

NP-vollständige Probleme sind also genauso schwierig wie andere NP-vollständige Probleme.

Satz 2.5. *Falls eine NP-vollständige Sprache in P ist, dann gilt P=NP.*

Beweis. Eine NP-vollständige Sprache ist per Definition NP-schwierig. Damit folgt die Behauptung aus Satz 2.4. □

Die Probleme 3-SAT und CLIQUE

Nach ihrem abstrakten Einsatz soll die Reduktionstechnik an einem Beispiel vorgeführt werden. Das so genannte 3-SAT-Problem ist ein Spezialfall des zuvor untersuchten Erfüllbarkeitsproblems der Aussagenlogik, bei dem die Form der booleschen Funktion eingeschränkt wird. Das CLIQUE-Problem auf einem Graphen beschäftigt sich mit der Suche nach vollständigen Teilgraphen, d.h. es werden Teilmengen der Knotenmenge gesucht, deren Knoten paarweise durch Kanten verbunden sind. Die Probleme lassen sich wie folgt formalisieren:

Definition 2.19. *Eine Formel der Aussagenlogik ist in konjunktiver Normalform (CNF, conjunctive normal form), wenn sie eine Konjunktion von Disjunktionstermen ist. Eine Formel ist in k-CNF, wenn sie in konjunktiver Normalform ist und jeder Disjunktionsterm maximal k Variablen enthält.*

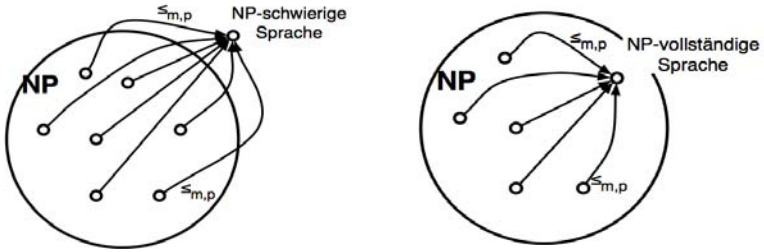


Abbildung 2.7: Schematische Darstellung zu NP-schwierig und NP-vollständig.
Der Kreis stellt jeweils die Klasse NP dar, die Punkte sind Probleme. Ein NP-schwieriges Problem ist mindestens so schwierig wie Probleme in NP, da alle NP-Probleme in polynomielleler Zeit darauf reduziert werden können. Von NP-vollständigen Problemen wird zusätzlich gefordert, dass sie selbst in NP liegen. Sie sind somit die schwierigsten aller in NP liegenden Probleme.

Problem Erfüllbarkeit einer booleschen Funktion in 3-CNF.

Eingabe Eine boolesche Funktion in 3-CNF.

Ausgabe „ja", falls es eine Belegung der Variablen gibt, so dass die Funktion erfüllt wird. „nein", falls keine solche Belegung existiert.

Definition 2.20. *Sei $G = (V, E)$ ein Graph mit Knoten V und Kanten E. Eine Clique ist eine Teilmenge $V_C \subseteq V$, sodass jeder Knoten aus V_C mit jedem anderen Knoten aus V_C durch eine Kante aus E verbunden ist. Eine k-Clique ist eine Clique mit genau k Knoten.*

Problem Existenz einer k-Clique in einem Graphen.

Eingabe Ein ungerichteter Graph $G = (V, E)$, eine natürliche Zahl n

Ausgabe „ja", falls der Graph eine Clique der Größe k enthält. „nein", falls keine Teilmenge von V eine k-Clique von G ist.

Das 3-SAT-Problem ist nicht schwerer als das CLIQUE-Problem, wie folgender Satz zeigt:

Satz 2.6. *Es gilt: 3-SAT $\leq_{m,p}$ CLIQUE.*

Beweis. Zu zeigen: Es existiert eine Reduktionsfunktion im Sinne von Definition 2.16.

Sei $f(\Phi) = \langle G, k \rangle$ eine Funktion, die eine Formel der Aussagenlogik in 3-CNF in einen Graphen G überführt. Für jede Disjunktion C in Φ werden drei Knoten angelegt, die mit den Literalen (d.h. den Variablen oder negierten Variablen) der Disjunktion bezeichnet werden. Zwischen zwei Knoten wird genau dann eine Kante eingefügt, wenn die beiden Knoten nicht zur selben Disjunktion gehören und die beiden Knoten nicht einer Variablen und ihrer negierten Variablen entsprechen.

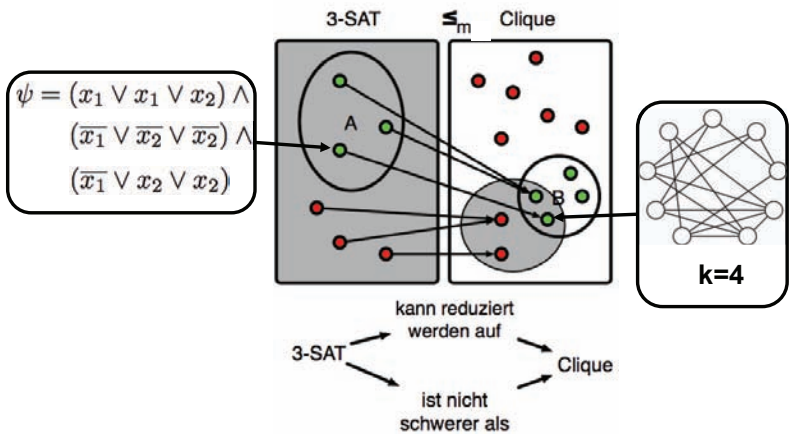


Abbildung 2.8: Skizze der Problemreduktion von 3-SAT nach CLIQUE.
Der linke Kasten stellt symbolisch eine boolesche Funktion in 3-CNF dar, der rechte Kasten einen Graphen mit einer 4-Clique. In der Mitte sind Instanzen der Probleme 3-SAT und CLIQUE durch Punkte visualisiert. Die Pfeile deuten eine polynomielle Abbildung an, die 3-SAT auf CLIQUE reduziert. Die Abbildung verdeutlicht, wie die gesuchte Reduktionsfunktion, die zeigt, dass CLIQUE mindestens so schwer ist wie 3-SAT, beschaffen sein soll.

Behauptung: Eine erfüllende Belegung in Φ existiert genau dann, wenn eine k-Clique in G existiert, d.h. die $f(\Phi)$ ist eine korrekte Reduktionsfunktion.

Die Äquivalenz wird durch zwei Implikationen gezeigt:

Angenommen, es existiert eine erfüllende Belegung in Φ. Dann liefert diese Belegung in jeder Disjunktion mindestens ein Literal mit Wert 1, denn sonst wäre sie nicht erfüllend. Wählt man aus jeder Knotenmenge einer Disjunktion einen beliebigen Knoten, der zu solch einem 1-Literal gehört, dann besteht die Auswahl aus k Knoten. Außerdem sind diese Knoten per Konstruktion paarweise durch Kanten verbunden, denn eine Variable und ihre Negation können nicht gleichzeitig 1 sein. Es existiert also eine k-Clique in G.

Umgekehrt angenommen, es existiert eine k-Clique in G. Dann gehört jeder Knoten dieser Clique zu einer anderen Klausel. Aus der Forderung, dass die zugehörigen Literale den Wert 1 annehmen müssen, ergibt sich die gesuchte Variablenbelegung. Das Verfahren führt zu keinem Widerspruch in der Belegung, da keine Kanten zwischen dem Knoten eines Literals und dem Knoten seines negierten Literals existieren.

Bleibt zu zeigen: Die Reduktionsfunktion hat polynomielle Laufzeit.

Die Reduktion besteht aus der Konstruktion der Knoten und der Kanten. Beides benötigt höchstens quadratische Zeit in der Anzahl der Variablen. Also ist die Reduktionsfunktion polynomiell. □

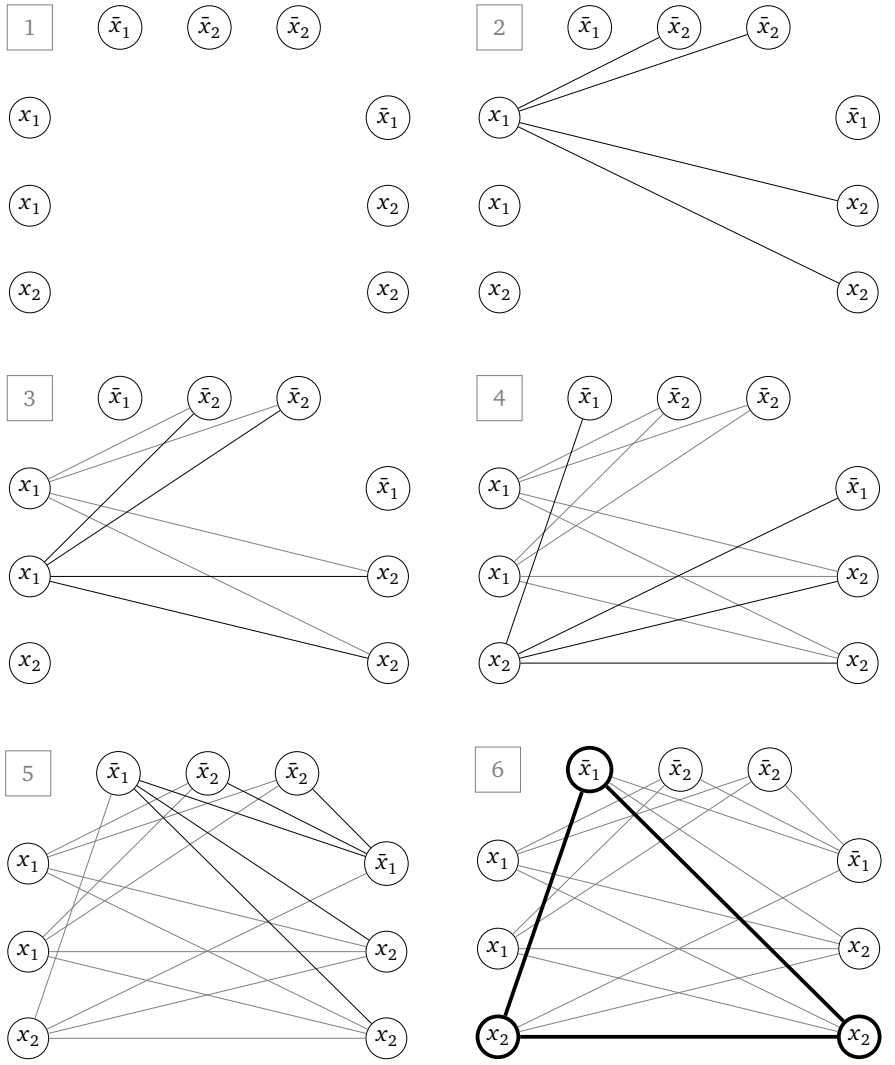


Abbildung 2.9: Schematischer Ablauf der Reduktion von 3-SAT zu CLIQUE.
Die Reduktionsfunktion des 3-SAT-Problems zum CLIQUE-Problem erstellt aus einer booleschen Funktion einen Graphen. Dargestellt ist die schrittweise Konstruktion entsprechend Satz 2.6 zur Formel $\psi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$, wobei das erste Bild den Entwurf der Knoten und das fünfte Bild den fertigen Graphen darstellt. Bild sechs zeigt am Beispiel der Wertebelegung $x_1 = 0, x_2 = 1$, dass der Graph tatsächlich eine 3-Clique enthält.

Kapitel 3

Algorithmen auf Graphen

Viele Alltagsprobleme lassen sich sehr einfach und elegant mit Hilfe von Graphen modelieren. Ein sehr altes, bekanntes Beispiel der Graphentheorie ist das *Königsberger Brückenproblem*, bei dem untersucht wird, ob es einen Rundgang durch eine Stadt gibt, bei dem jede der sieben Brücken genau einmal benutzt wird. Der eigentliche Problemgehalt lässt sich als Ansammlung von Knoten und Kanten darstellen, die die Topologie des Stadtplans abstrakt wiedergeben. Auf ähnliche Weise lassen sich viele Problem, die Wege, Netzwerke, Nachbarschaften oder Flüsse behandeln, auf ähnliche Graphen zurückführen und mit einer kleinen Zahl vielseitig einsetzbarer Graphenalgorithmen lösen.

Dieses Kapitel gibt eine kurze Einführung in die Graphentheorie und führt die nötigen Begriffe ein. In den darauffolgenden Abschnitten werden typische algorithmische Fragestellungen und Lösungsalgorithmen vorgestellt.

3.1 Grundlagen zu Graphen

Nachdem in der Einführung schon informell über Graphen gesprochen wurde, folgt schließlich eine ordentliche Definition. Man unterscheidet zwei Typen von Graphen:

Definition 3.1. *Ein ungerichteter Graph ist ein Paar $G = (V, E)$, wobei $E \subseteq \{\, \{u, v\} \mid u, v \in V \,\}$. Die Elemente von E sind (ungeordnete) Mengen von Elementen aus V. Elemente aus V heißen Knoten, Elemente aus E heißen Kanten.*

Definition 3.2. *Ein gerichteter Graph ist ein Paar $G = (V, E)$, wobei $E \subseteq \{\, (u, v) \mid u, v \in V \,\}$. Die Elemente von E sind (geordnete) Paare von Elementen aus V. Elemente aus V heißen Knoten, Elemente aus E heißen gerichtete Kanten oder Bögen.*

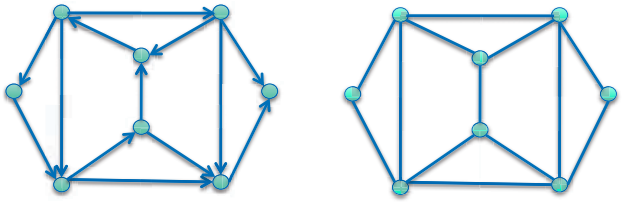


Abbildung 3.1: Schema eines gerichteten und eines ungerichteten Graphen.
Graphen sind abstrakte Knotenmengen, die paarweise durch Kanten verbunden sein können. Sind die Kanten ungerichtet (dargestellt durch eine Linie), repräsentieren sie z.B. einen Weg oder eine Beziehung. Eine gerichtete Kante (dargestellt durch einen Pfeil) kann z.B. ein hierarchisches Verhältnis oder eine Flussrichtung wiedergeben.

Nachbarschaftsbeziehungen

Um verschiedenste Eigenschaften von Graphen besser untersuchen zu können, ist es sinnvoll, Begrifflichkeiten für die Nachbarschaftsbeziehungen von Knoten und Kanten einzuführen:

Definition 3.3. *Ein Knoten v heißt mit einer Kante e inzident, wenn $v \in e$. Zwei Knoten x und y heißen adjazent in G, wenn $\{x, y\} \in E$.*

Definition 3.4. *Der Grad eines Knotens v ist die Anzahl der mit v inzidenten Kanten. Sind alle Knoten von G paarweise adjazent, heißt G vollständig oder wird Clique genannt.*

Abbildung 3.2: Schematische Darstellung von Inzidenz und Adjazenz.
Die Begriffe inzident und adjazent beschreiben, ob ein Knoten mit einer Kante verbunden ist oder ob zwei Knoten durch eine Kante miteinander verbunden sind. Mit diesen kleinsten Beziehungen werden größere Zusammenhänge beschrieben.

Definition 3.5. *Ein Graph heißt bipartit, wenn sich V in zwei diskunkte Teile V_1 und V_2 teilen lässt, so dass jede Kante in E einen Endknoten in V_1 und einen Endknoten in V_2 besitzt.*

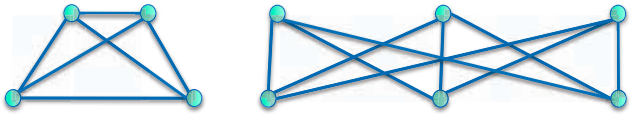


Abbildung 3.3: Darstellung einer Clique und eines bipartiten Graphen.
Eine Clique (ein vollständiger Graph) ist ein Graph, in dem jede mögliche Kante existiert. Sie wird häufig als Teilgraph eines größeren Graphen untersucht. Ein bipartiter Graph lässt sich so in zwei Teile zerlegen, dass innerhalb jedes der zwei Teile kein Knoten mit einem anderen verbunden ist, und alle Kanten reichen von einem Teil in den anderen. Solche Strukturen treten bei Zuordnungsproblemen auf.

Satz 3.1. *Ein Graph ist genau dann bipartit, wenn er nur Kreise gerader Länge enthält.*

Beweis. Übung. □

3.1.1 Kodierung von Graphen

Graphen müssen ebenso wie andere Datenstrukturen im Computer kodiert gespeichert werden. Es gibt mehrere Ansätze, die den Speicherbedarf, aber möglicherweise auch die Laufzeit von Algorithmen beeinflussen.

Definition 3.6. *Ist $G = (V, E)$ ein Graph mit $n = |V|$ Knoten und $m = |E|$ Kanten, so sieht eine Kantenliste wie folgt aus:*

$$n, m, \{a_1, e_1\}, \{a_2, e_2\}, \ldots, \{a_m, e_m\} \quad \textit{wobei } \{a_i, e_i\} \textit{ Endknoten der Kante } i \textit{ sind}$$

Die Reihenfolge der Kanten innerhalb der Liste ist beliebig. Bei Schleifen an einem Knoten v wird $\{v, v\}$ eingefügt. Bei gerichteten Graphen ist zu beachten, dass immer zunächst der Anfangsknoten und dann der Endknoten repräsentiert wird.

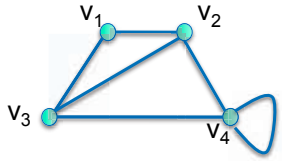


Abbildung 3.4: Beispiel zur Konstruktion einer Kantenliste.
Gegeben sei der dargestellte Graph mit den Knoten v_1, v_2, v_3, v_4. Er hat vier Knoten und sechs Kanten. Daher lautet eine Kantenliste: $4, 6, \{v_1, v_2\}, \{v_2, v_4\}, \{v_4, v_4\}, \{v_3, v_4\}, \{v_2, v_3\}, \{v_3, v_1\}$. Kantenlisten sind nicht eindeutig, da die Reihenfolge der Kanten beliebig ist. Schleifen an Knoten sind selten relevant und sind hier nur der Vollständigkeit halber aufgeführt.

Kanten können auch ein Gewicht oder Kosten haben. Dann gibt es eine Kostenfunktion $c : E \to \mathbb{R}$ und das Gewicht wird schematisch neben die zugehörige Kante geschrieben. Die Gewichte werden kodiert, indem man für jede Kante das Gewicht dazu notiert. Eine Kantenliste erfordert $2m + 2$ Speicherstellen zur Speicherung, eine Kantenliste mit Gewichten erfordert $3m + 2$ Zellen.

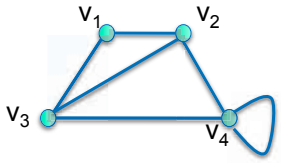
Definition 3.7. *Sei $G = (V, E)$ ein einfacher Graph mit den Knoten $1, \ldots, n$. Dann heißt die Matrix $A \in \mathbb{R}^{n \times n}$ mit*

$$A_{ij} = \begin{cases} 1 & \textit{falls } i \textit{ und } j \textit{ durch eine Kanten verbunden sind,} \\ 0 & \textit{sonst} \end{cases}$$

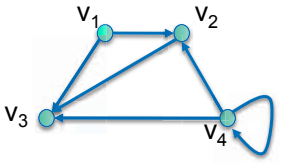
die Adjazenzmatrix von G. Falls G ein ungerichteter Graph ist, ist A symmetrisch und es reicht, die obere Dreiecksmatrix zu speichern. Hat G Kantengewichte, so setzt man

$$A_{ij} = \begin{cases} c((i, j)) & \textit{falls } i, j \in E \\ 0, +\infty, \infty & \textit{sonst, je nach Bedarf} \end{cases}$$

Der Speicherbedarf einer Adjazenzmatrix beträgt $O(n^2)$ Zellen.



$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

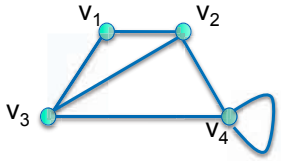


$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

Abbildung 3.5: Beispiel zur Konstruktion einer Adjazenzmatrix.
Adjazenzmatrizen enthalten prinzipiell nicht mehr Informationen als Kantenlisten. Die Anzahl der Zeilen und Spalten entspricht der Anzahl der Knoten und die 1-Einträge der Matrix beschreiben die Kanten des Graphen. (Im gerichteten Graphen entspricht die Anzahl der Einsen in der Matrix der Anzahl der Kanten, im ungerichteten Graphen ist die Adjazenzmatrix symmetrisch.) Ein Vorteil der Adjazenzmatrix ist, dass man schnell alle Kanten eines Knoten finden kann, wohingegen man in der Kantenliste alle Kanten überprüfen muss. Nachteil ist der erhöhte Speicherbedarf.

Ein weiteres Problem ist die Berechnung der Kantenanzahl jedes Knotens. Auch diese Information lässt sich vorab mit in die Datenstruktur aufnehmen:

Definition 3.8. *Speichert man für einen Graphen $G = (V, E)$ die Anzahl von Knoten, und für jeden Knoten seinen Grad und seine Nachbarknoten, so nennt man solch eine Datenstruktur eine Adjazenzliste von G. Der Speicherbedarf ist $O(n + m)$.*



Knoten	Grad	Nachbarn
1	2	2, 3
2	3	1, 3, 4
3	3	1, 2, 4
4	3	2, 3, 4

Abbildung 3.6: Beispiel zur Konstruktion einer Adjazenzliste.
Zu jedem Knoten wird zum einen der Grad des Knotens und zum anderen eine Liste der benachbarten Knoten gespeichert. Die Liste der Nachbarn enthält implizit die Kanteninformation und eignet sich um den Graphen zu durchlaufen.

3.1.2 Spezielle Graphen

Viele Probleme beschäftigen sich mit dem Traversieren von Graphen:

Definition 3.9. *Ein Weg (oder Pfad) in einem (gerichteten oder ungerichteten) Graphen $G = (V, E)$ ist eine Folge von Knoten $(v_0, v_1, \ldots, v_k)$ aus V, so dass für alle $i \in \{1, \ldots, k\}$ gilt, dass $(v_{i-1}, v_i) \in E$ (bzw. $\{v_{i-1}, v_i\} \in E$) eine Kante ist. Ein Weg heißt einfach, falls in der Folge $(v_i, \ldots, v_k)$ kein Knoten mehrfach auftritt. Ein einfacher Weg heißt Kreis, falls $(v_0 = v_k)$. Ein Graph heißt azyklisch (oder kreisfrei), falls es in ihm keinen Kreis gibt.*

Vorsicht: Diese Definitionen sind so, wie sie meistens in der Literatur auftreten. Sie unterscheiden sich damit ein klein wenig von den ADM-Vorlesungen der Vorjahre, wo es eine Unterscheidung zwischen Wegen und Pfaden gab.

Definition 3.10. *Ein Graph heißt zusammenhängend, falls es zu jedem Knotenpaar $v, w \in V$ einen Weg von v nach w gibt. Die zusammenhängenden Teile von G heißen Zusammenhangskomponenten.*

Definition 3.11. *Ein Baum ist ein zusammenhängender Graph, der keine Kreise enthält. Ein Wald ist ein Graph, der keine Kreise enthält, also eine Ansammlung von Bäumen.*

Definition 3.12. *Der zu $G = (V, E)$ komplementäre Graph G' ist der Graph $G' = (V, E')$ mit $(i, j) \in E' \Leftrightarrow (i, j) \notin E$.*

Satz 3.2. *Mindestens einer der Graphen G oder G' ist zusammenhängend.*

Beweis. Übung. □

Definition 3.13. *$G' = (V', E')$ heißt Untergraph (oder Teilgraph) von $G = (V, E)$, falls $V' \subseteq V$ und $E' \subseteq E$ ist. $G' \subseteq G$ heißt aufspannender Untergraph von G, falls zusätzlich $V' = V$ gilt.*

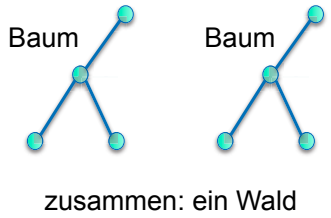


Abbildung 3.7: Darstellung eines Baumes und eines Waldes.
Ein Baum ist ein zusammenhängender Graph, der keine Kreise enthält. Man sieht leicht ein, dass es dann zwischen zwei beliebigen Punkten immer genau einen einfachen Weg gibt. Zur Behandlung von Probleme auf Bäumen gibt es spezielle Algorithmen. Einen Graphen, der aus mehreren Bäumen besteht, nennt man einen Wald.

Abbildung 3.8: Darstellung eines komplementären Graphen.
Der zu einem Graphen komplementäre Graph ist konstruierbar, indem man alle Kanten des Ursprungsgraphen entfernt und zwischen allen Knoten eine Kante legt, wo ursprünglich keine Kante war.

Abbildung 3.9: Darstellung eines Untergraphen und eines aufspannenden Baumes.
Ein Untergraph ist ein Graph, der aus einer Teilmenge der ursprünglichen Knotenmenge und einer Teilmenge der ursprünglichen Kantenmenge besteht. Wenn der Untergraph alle Originalknoten enthält und nur Kanten fehlen, spricht man von einem aufspannenden Untergraphen. Ist dieser aufspannende Untergraph zudem noch ein Baum, handelt es sich um einen aufspannenden Baum.

3.1.3 Beispiele für Problemstellungen

Wie schon erläutert, können mit Hilfe von Graphen viele Probleme modelliert werden. Zum Abschluss wollen wir einige Beispielprobleme betrachten.

Sei $G = (V, E)$ ein Graph (gerichtet oder ungerichtet).

Problem 1 Eingabe Graph G in Adjazenzlistenform.

Ausgabe „ja“, wenn G einen Kreis enthält. „nein“, sonst.

Problem 2 Eingabe Graph G in Adjazenzlistenform.

Ausgabe Anzahl der Zusammenhangskomponenten von G.

Problem 3 Eingabe Graph G in Adjazenzlistenform.

Ausgabe Für jede Zusamenhangskomponente ein aufspannender Baum.

3.2 Graphensuche

Wir betrachten zwei Standardalgorithmen zum Traversieren bzw. Durchsuchen eines Graphen. Ziel dieser Algorithmen ist in der Regel, einen Knoten mit bestimmten Eigenschaften zu finden. Die Wahl des Suchalgorithmus ist im Prinzip egal, kann aber die Laufzeit erheblich beeinflussen.

3.2.1 Depth-First Search (DFS) Algorithmus

Die Tiefensuche (*Depth-First Search*) ist ein Algorithmus zum Traversieren eines Graphen. Dabei wird jeder Weg zunächst so weit wie möglich abgesucht, indem vom zuletzt besuchten Knoten weitergegangen wird. Sobald nicht weiter „in die Tiefe“ gesucht werden kann, werden andere Wege verfolgt. Die genaue Reihenfolge hängt von der Repräsentation des Graphen ab.

Problem Tiefensuche (Traversierung eines Graphen in die Tiefe).

Eingabe Ein Graph $G = (V, E)$ in Form von Adjazenzlisten, d.h. für jeden Knoten v ist eine Adjazenzliste `adj[v]` gegeben.

Ausgabe Eine Partition der Kantenmenge E in einen aufspannenden Wald T und die übrigen Kanten B, d.h. $B \cup T = E$ und $B \cap T = \emptyset$.

Der Graph wird in Form von Adjazenzlisten eingelesen, da der Algorithmus auf den Knoten des Graphen operiert und zu jedem Zeitpunkt nur die lokale Topologie kennen muss, also Anzahl und Ziel der abgehenden Kanten.

Der aufspannende Wald beschreibt den Weg, den der Algorithmus beim Durchlaufen des Graphen genommen hat. (Die bei Verzweigungen gewählte Reihenfolge geht bei dieser Darstellung verloren. Sie spielt aber auch keine Rolle, da sie von der nicht eindeutigen Darstellung der Adjazenzlisten abhing.)

Beschreibung des Algorithmus

Der DFS-Algorithmus beruht auf der Idee, dass er sich besuchte Knoten merken kann und mit Hilfe dieser Information über den weiteren Weg entscheidet. Um sein „Gedächtnis“ visuell zu veranschaulichen, ordnet man den Knoten eine Farbe zu.

Zu Beginn sind alle Knoten *unentdeckt* und werden weiß gefärbt. Dann werden bei einem Startknoten beginnend Schritt für Schritt die Knoten abgelaufen. Wird ein Knoten *entdeckt* (also ein weißer Knoten besucht), so wird dieser grau gefärbt und wieder ein Nachfolger dieses Knotens gesucht.

Rekursiv wird dies weitergeführt, bis ein Knoten keinen weißen Nachfolger mehr hat. Er wird dann als *abgearbeitet* betrachtet und schwarz gefärbt. Anschließend wird bei seinem letzten noch nicht abgearbeiteten (grauen) Vorgänger fortgefahren, indem dort nach weiteren noch unentdeckten (weißen) Nachfolgern gesucht wird. Dies läuft solange, bis alle Knoten abgearbeitet wurden, also schwarz gefärbt sind.

Zusätzlich merkt der Algorithmus sich nebenbei noch zwei Zeitstempel d und f, wobei für jeden Knoten der Entdeckungszeitpunkt (Knoten wird grau) in d und der Beendigungszeitpunkt (Knoten wird schwarz) in f gespeichert wird. (Der Begriff *Zeit* wird hier diskret im Sinne von *Arbeitsschritten* verwendet.) Außerdem existiert eine Liste `pred`, in welcher der (eindeutige) Vorgänger jedes Knotens gespeichert wird.

Der Pseudocode des Algorithmus lässt sich in eine Hauptfunktion `DFS` (zur Initialisierung) und eine Hilfsfunktion `DFS-Visit` (zum Abarbeiten eines Knotens) gliedern:

Listing 3.1: DFS-Algorithmus (*depth-first search*), rekursive Version

```
1   DFS (graph G):
2       for each node u in V(G) do
3           color[u] := white
4           pred[u] := nil
5       time := 0
6       for each node u in V(G) do
7           if color[u] == white
8               DFS-Visit (u)
9
10  DFS-Visit (node u):
11      color[u] := grey
12      time := time + 1
13      d[u] := time
14      for each node v in adj[u] do
15          if color[v] == white
16              pred[v] := u
17              DFS-Visit (v)
18      color[u] := black
19      time := time + 1
20      f[u] := time
```

Die Funktion `DFS` färbt zu Beginn alle Knoten weiß, setzt die Zeit auf 0 und legt fest, dass zu jedem Knoten noch kein Vorgänger bekannt ist (`nil`). Desweiteren ruft sie für alle Knoten die Hilfsfunktion `DFS-Visit` auf, um sicherzustellen, dass jeder Knoten mindestens einmal besucht wird. (Andernfalls würden nur solche Knoten besucht werden, die über Vorgängerknoten mit einem vorzugebendem Startknoten der gleichen Zusammenhangskomponente verbunden sind. Im Allgemeinen wäre das unzureichend.)

Die Funktion `DFS-Visit` führt zum eigentlichen Durchlaufen des Graphen, da sie die Knoten färbt und sich rekursiv für die Nachfolger aufruft. `DFS-Visit(u)` färbt also zunächst den Knoten u grau, erhöht die Zeit und speichert den Entdeckungszeitpunkt von u in $d[u]$. Dann speichert sie den Knoten u für alle weißen Nachbarn als Vorgänger in `pred` und ruft sich selbst für diese Nachbarn auf. Zuletzt färbt `DFS-Visit` den Knoten u noch schwarz, erhöht die Zeit erneut und speichert den Beendigungszeitpunkt von u in $f[u]$.

Der schematische Ablauf des DFS-Algorithmus ist in den Abbildungen 3.10 und 3.11 dargestellt.

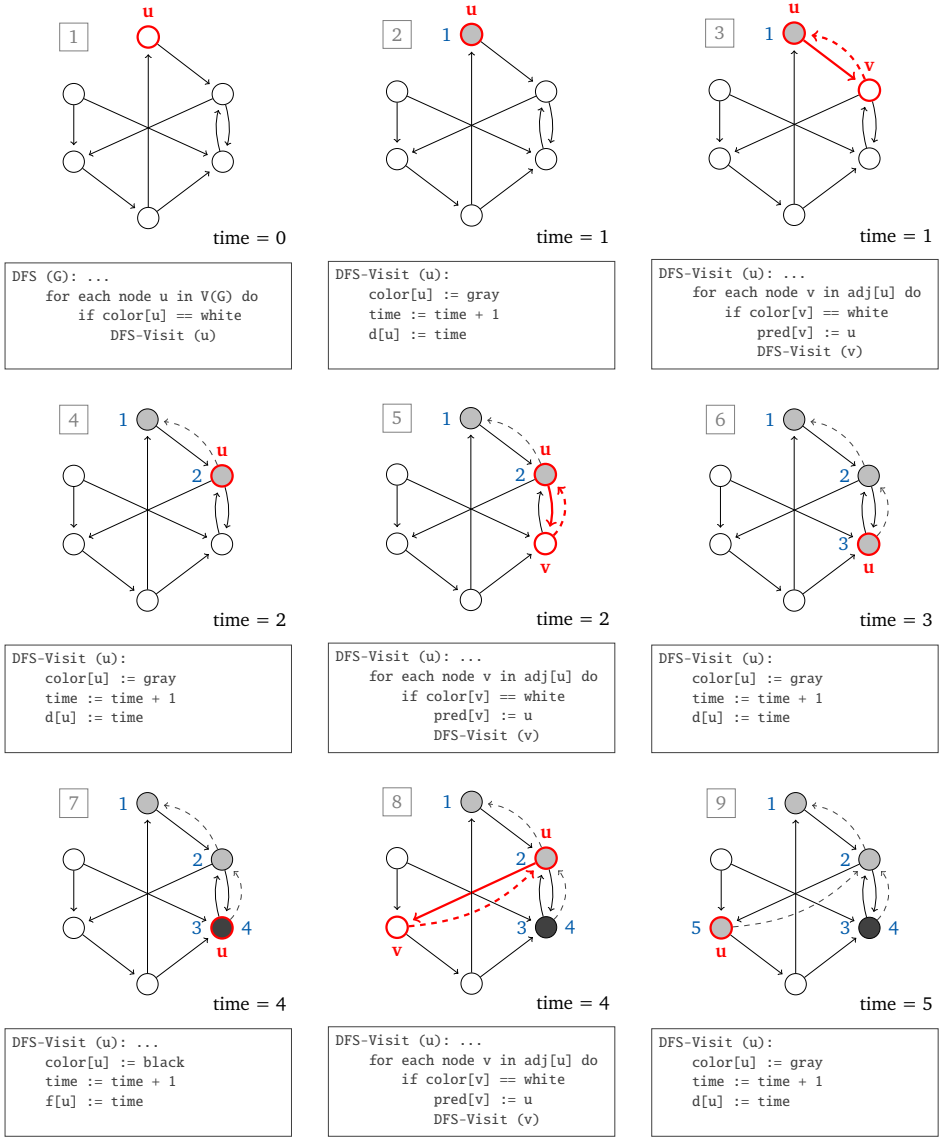


Abbildung 3.10: Schematischer Ablauf des DFS-Algorithmus (Teil 1).
Diese zweiteilige Abbildung zeigt den vollständig Ablauf des DFS-Algorithmus auf einem Beispielgraphen. Unter jedem Teilschritt befindet sich ein Auszug des Pseudocodes 3.1, der die Änderungen seit dem letzten Schritt erläutern soll. Oben links vom Graphen wird die Schrittnummer angegeben, unten rechts vom Graphen ist die Zeitvariable dargestellt.

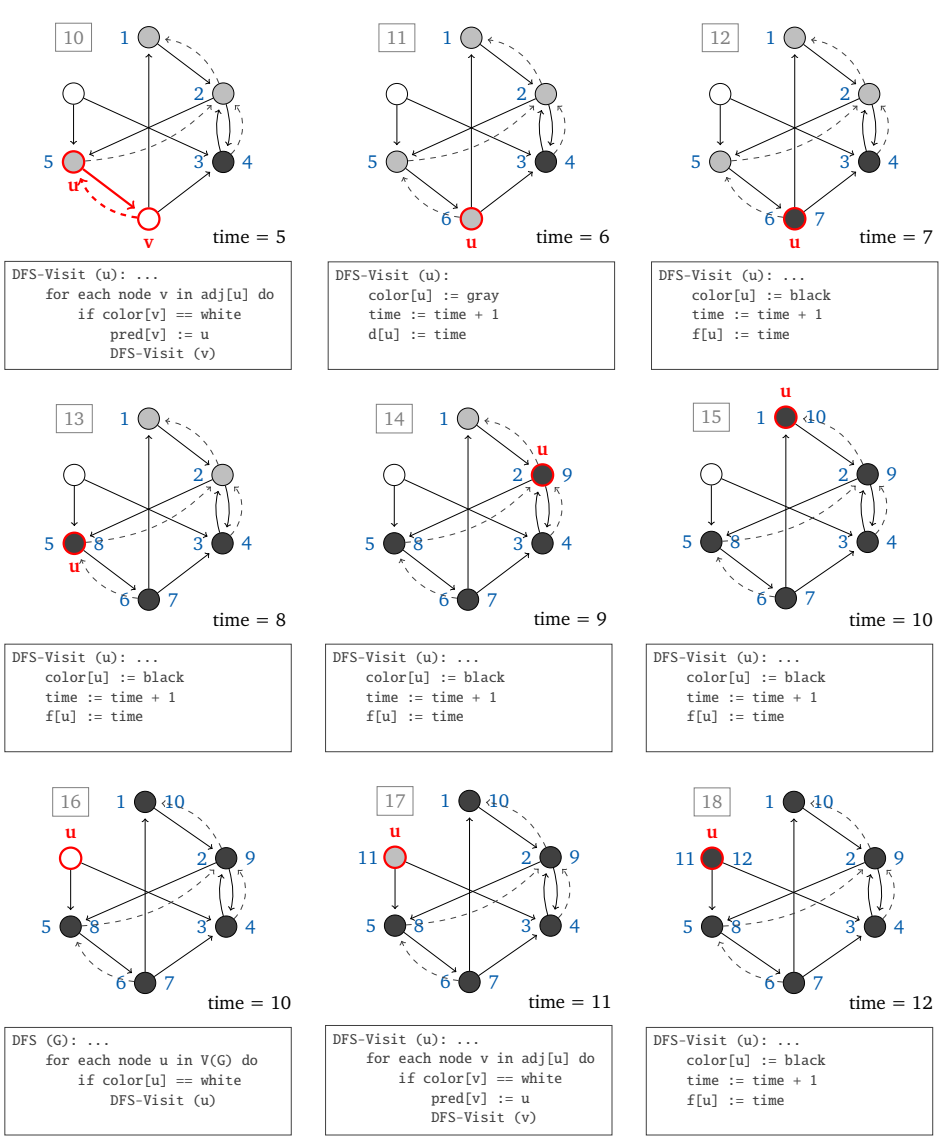


Abbildung 3.11: Schematischer Ablauf des DFS-Algorithmus (Teil 2).
(Fortsetzung der vorangehenden Seite.) Aktive Knoten und Kanten (das sind solche, die gerade ausgewählt sind oder bearbeitet wurden) sind rot hervorgehoben. Links neben jedem Knoten steht sein Entdeckungszeitpunkt, rechts neben jedem Knoten steht sein Beendigungszeitpunkt. Die gestrichelten Kanten repräsentieren den Vorgänger-Speicher.

Analyse des Algorithmus

Als *Laufzeit* des DFS-Algorithmus ergibt sich $T_{\text{DFS}} = O(|V| + |E|)$, da jeder Knoten genau einmal besucht wird und pro Knoten nur die wegführenden Kanten gespeichert werden und somit jede Kante genau einmal gespeichert wird. Mit $|E| \leq |V|^2$ folgt außerdem $T_{\text{DFS}} = O(|V|^2)$.

Der Speicherbedarf wird durch die Länge der diversen Listen und aufgrund der geschachtelten Funktionsaufrufe durch den längsten zurückgelegten Weg bestimmt. Damit ist $S_{\text{DFS}} = O(|V|)$.

Ist der Algorithmus beendet, so lassen sich die Kanten anhand des aufspannenden Waldes (implizit durch `pred` gegeben) in vier Teilmengen zerlegen:

Kantenklassifizierung

Baumkanten Die Kanten des aufspannenden Waldes.

Rückwärtskanten Die Kanten (u, v), die Knoten u mit Vorgängerknoten v verbinden.

Vorwärtskanten Die Kanten (u, v), die Knoten u mit Nachfolgeknoten v verbinden, aber selbst keine Kanten des aufspannenden Waldes sind.

Kreuzkanten Alle weiteren Kanten.

Manchmal werden Baumkanten nicht als eigene Klasse betrachtet, sondern mit zu den Vorwärtskanten gezählt. Die Beschreibung wird dadurch weniger explizit, aber wirkt symmetrischer.

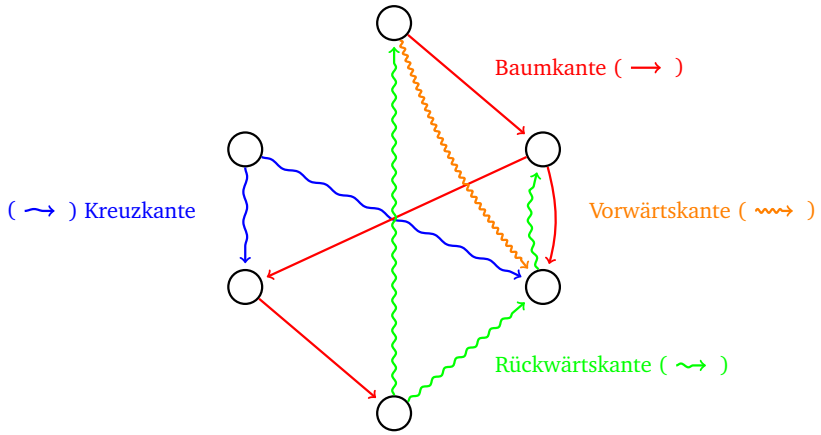


Abbildung 3.12: Schema zur Kantenbezeichnung im DFS-Algorithmus.
Der DFS-Algorithmus findet zu einem gegebenen Graphen einen aufspannenden Wald. Dieser kann herangezogen werden, um die Kanten des Graphen zu klassifizieren. Baumkanten sind Teilkanten des aufspannenden Waldes. Vorwärtskanten sind Nicht-Baumkanten, die auf Nachfolgeknoten zeigen. Rückwärtskanten zeigen auf Vorgängerknoten. Alle Kanten, die sich nicht in dieses Schema einordnen lassen, bezeichnet man als Kreuzkanten.

Satz 3.3 (Klammerungstheorem)**.** *Seien u und v Knoten eines gerichteten oder ungerichteten Graphen G. Nach einer DFS-Suche auf G gilt genau eine der drei folgenden Aussagen:*

1. *Die Intervalle $[d[u], f[u]]$ und $[d[v], f[v]]$ sind disjunkt, d.h. u und v sind keine Vorgänger bzw. Nachfolger voneinander.*
2. *Das Intervall $[d[u], f[u]]$ ist vollständig in dem Intervall $[d[v], f[v]]$ enthalten, und der Knoten u ist ein Nachfolger von v.*
3. *Das Intervall $[d[v], f[v]]$ ist vollständig in dem Intervall $[d[u], f[u]]$ enthalten, und der Knoten v ist ein Nachfolger von u.*

Beweis. Einer der Knoten u und v wird zuerst entdeckt. ⇒ Fallunterscheidung:

- Sei zunächst $d[u] < d[v]$. Dann gibt es die folgenden beiden Fälle:
  1. $d[v] < f[u]$: Dann wurde v entdeckt, als u noch nicht abgearbeitet (grau) war. Damit muss v ein Nachfolger von u sein. Denn v wurde später als u entdeckt, daher wurden alle von v ausgehende Kanten untersucht, bevor die Suche zu u zurückkehrte. Somit wurde erst v beendet, bevor die Suche zu u zurückkehrte und u beendete. Es git folglich $f[v] < f[u]$ und insgesamt $d[u] < d[v] < f[v] < f[u]$, also $[d[v], f[v]] \subseteq [d[u], f[u]]$
  2. $f[u] < d[v]$, dann wurde v erst entdeckt, als u schon beendet war. Da ein Knoten natürlich kann erst beendet werden, nachdem er entdeckt wurde, gilt auch $d[u] < f[u]$ und $d[v] < f[v]$, somit folgt $d[u] < f[u] < d[v] < f[v]$, d.h. die Intervalle sind disjunkt.
- Der Fall $d[v] < d[u]$ folgt analog zu $d[u] < d[v]$.

□

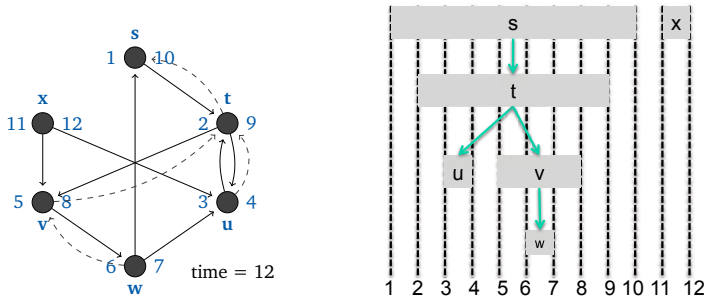


Abbildung 3.13: Schematische Zeitleiste zum Klammerungstheorem.
Die gestrichelten Linien kennzeichnen die Entdeckungs- und Beendigungszeitpunkte. Die grauen Balken sind Bearbeitungsintervalle der Knoten s, t, u, v, w und x. Aus dem Klammerungstheorem folgt dann: s hat den Nachfolger t, t hat die Nachfolger u und v, v hat den Nachfolger w. Alle diese Knoten sind keine Nachfolger oder Vorgänger von x.

Satz 3.4 (Satz vom weißen Weg)**.** *In einem gerichteten DFS-Wald eines gerichteten oder ungerichteten Graphen $G = (V, E)$ ist ein Knoten v genau dann ein Nachfolger eines Knotens u, wenn zur Zeit $d[u]$ der Knoten v von u aus allein über weißen Knoten erreichbar ist.*

Beweis. Die Äquivalenz wird durch zwei Implikationen gezeigt:

$\Rightarrow$: Sei v ein Nachkomme von u. Sei also $d[v] > d[u]$ und sei u noch nicht beendet. Dann war der Weg zwischen u und v zur Zeit $d[u]$ offenbar weiß, denn DFS hat sich von u zu v mit Hilfe der Nachfolger-suchenden Schleife von DFS-Visit bewegt (Zeilen 15 bis 17).

$\Leftarrow$: Es gebe nun einen weißen Weg von u nach v zur Zeit $d[u]$. O.b.d.A. werde jeder andere Knoten auf dem Weg ein Nachkomme von u. (Falls das nicht so ist, machen wir folgende Überlegung einfach zuerst für den ersten Knoten auf dem Weg, der nicht mehr Nachkomme von u wird.) Sei w der letzte Knoten auf dem weißen Weg, der noch Nachkomme von u wird. Offenbar gilt dann: $f[w] < f[u]$. Da $d[v] > d[u]$ und $d[v] < f[w]$ (v ist direkter Nachfolger von w) gilt: $d[u] < d[v] < f[w] < f[u]$. Wegen des Klammerungstheorems muss $f[v] < f[w]$ gelten, da v ein Nachfolger von w ist. Also ist $d[u] < d[v] < f[v] < f[w] < f[u]$ und somit gilt $[d[v], f[v]] \subset [d[u], f[u]]$. Nach dem Klammerungstheorem folgt, dass v ein Nachkomme von u ist. Vergleiche auch Abbildungen 3.14 und 3.15. □

Satz 3.5. *Sei $G = (V, E)$ ein ungerichteter Graph. Dann gilt:*

- *Im DFS-Wald gibt es nur Baum- und Rückwärtskanten.*
- *Die Menge aller Baumkanten bildet einen Wald, in dem jede Zusammenhangskomponente von G einen aufspannenden Baum erzeugt.*
- *G ist genau dann kreisfrei, wenn es keine Rückwärtskanten gibt.*

Beweis. Übung. □

Abbildung 3.14: Schema eines weißen Weges.
Zum Entdeckungszeitpunkt des Knotens u ist der Knoten v von u aus allein über weiße Knoten erreichbar. Nach dem Satz vom weißen Weg ist dieser Zusammenhang äquivalent dazu, dass der Knoten v ein Nachfolger des Knotens u ist.

Abbildung 3.15: Schema zum Beweis des Satzes vom weißen Weg.
Es ist schwieriger die Implikation zu zeigen, dass aus der Existenz eines weißen Weges zwischen u und v zum Entdeckungszeitpunkt von u schon folgt, dass v ein Nachkomme von u ist. Dazu wird der direkte Vorgängerknoten w von v betrachtet und das Klammerungstheorem angewendet.

Die Tiefensuche lässt sich auch ohne Rekursion realisieren. Dafür lässt sichder abstrakte Datentyp Stacks verwenden:

Definition 3.14. *Ein Stack ist ein abstrakter Datentyp, in dem sich Daten dynamisch speichern lassen. Es können beliebig Elemente hinzugefügt werden, jedoch kann immer nur das zuletzt hinzugefügte Element wieder weggenommen oder angeschaut werden. Dies entspricht der last-in-first-out (LIFO) Strategie.*

Für Stacks sind die folgenden Methoden mit Laufzeit $O(1)$ definiert:

`empty (S)`	Gibt aus, ob der Stack S leer ist.
`push (S,x)`	Fügt das Element x zum Stack S hinzu.
`pop (S)`	Entfernt das zuletzt zu S hinzugefügte Element wieder.
`x := top (S)`	Ließt den Inhalt des zuletzt zum Stack S hinzugefügten Elements.

Mit Hilfe eines Stacks lässt sich nun der DFS-Algorithmus so umformulieren, dass er ohne Rekursion auskommt. Dabei werden nun alle (!) Nachbarn des zuletzt entdeckten Knoten auf dem Stack gespeichert. Dann wird dieser Knoten aus dem Stack gelöscht. Als nächstes wird der letzte zum Stack hinzugefügte Knoten betrachtet.

Da immer der letzte zum Stack hinzugefügte Knoten betrachtet wird, findet auch hier eine Tiefensuche statt, denn dieser Knoten ist ein Nachfolger des zuletzt betrachteten Knotens, soweit dieser noch einen Nachfolger hatte, sonst wird zu dem zuletzt betrachteten Vorgänger gesprungen.

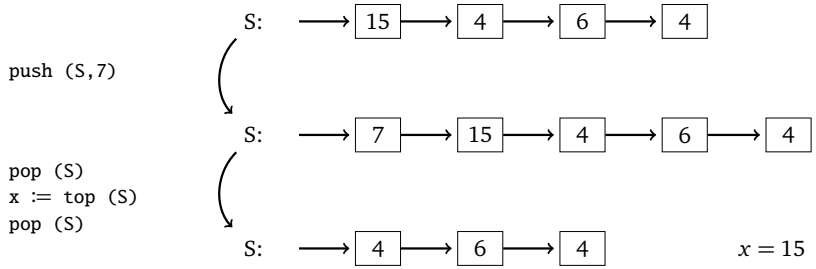


Abbildung 3.16: Beispiel zu Stack-Operationen.
Gegeben ist ein Stack (Kellerstapel) $S = (15, 4, 6, 4)$. Im ersten Schritt wird das Element 7 zum Stack hinzugefügt („auf den Stack gelegt"). Im zweiten Schritt wird das oberste Element (aktuell die 7) vom Stack entfernt, dann wird das oberste Element (aktuell die 15) ausgelesen und in der Variablen x gespeichert. Abschließend wird erneut das oberste Element (ebendiese 15) entfernt. Effektiv wurde in diesem Schritt also das zweit-oberste Element ausgelesen und der Stack um zwei Elemente verkleinert.

Der Pseudocode des iterativen DFS-Algorithmus lautet:

Listing 3.2: DFS-Algorithmus (*depth-first search*), iterative Version

```
1   DFS (graph G):
2       for each node u in V do
3           color[u] := white
4       time := 0
5       S := empty stack
6       for each node u in V do
7           if color[u] == white
8               DFS-Visit (u);
9
10  DFS-Visit (node u):
11      push (S,u)
12      while not empty (S) do
13          u := top (S)
14          if color[u] == white
15              color[u] := gray
16              time := time+1
17              d[u] := time
18              for each node v in adj[u] do
19                  if color[v] == white
20                      push (S,v)
21          else if color[u] == gray
22              color[u] := black
23              time := time+1
24              f[u] := time
25              pop (S)
26          else if color[u] == black
27              pop (S)
```

In der stackbasierten `DFS` Funktion wird ein leerer Stack S angelegt. Ansonsten unterscheidet sich die stackbasierte `DFS` Funktion nur wenig von der rekursiven Version.

`DFS-Visit(u)` fügt zuerst durch `push(S,u)` (in Zeile 11) den Knoten u in den Stack S ein. Solange der Stack nicht leer ist, wird das zuletzt in den Stack eingefügte Element (`top(S)`) als u gesetzt und betrachtet. Dazu werden die beiden folgenden Fälle unterschieden. Wenn der Knoten u noch weiß ist, wird er grau gefärbt, die Zeit erhöht und der Entdeckungszeitpunkt gespeichert. Dann wird jeder noch weiße Nachbar v von u durch `push(S,v)` (in Zeile 20) in den Stack aufgenommen. Ist andernfalls u zu dem Betrachtungszeitpunkt bereits grau, so wird er nun schwarz gefärbt, die Zeit erhöht, sein Beendigungszeitpunkt gespeichert und er wird durch `pop(S)` aus dem Stack S entfernt, da er nicht mehr weiter betrachtet werden muss.

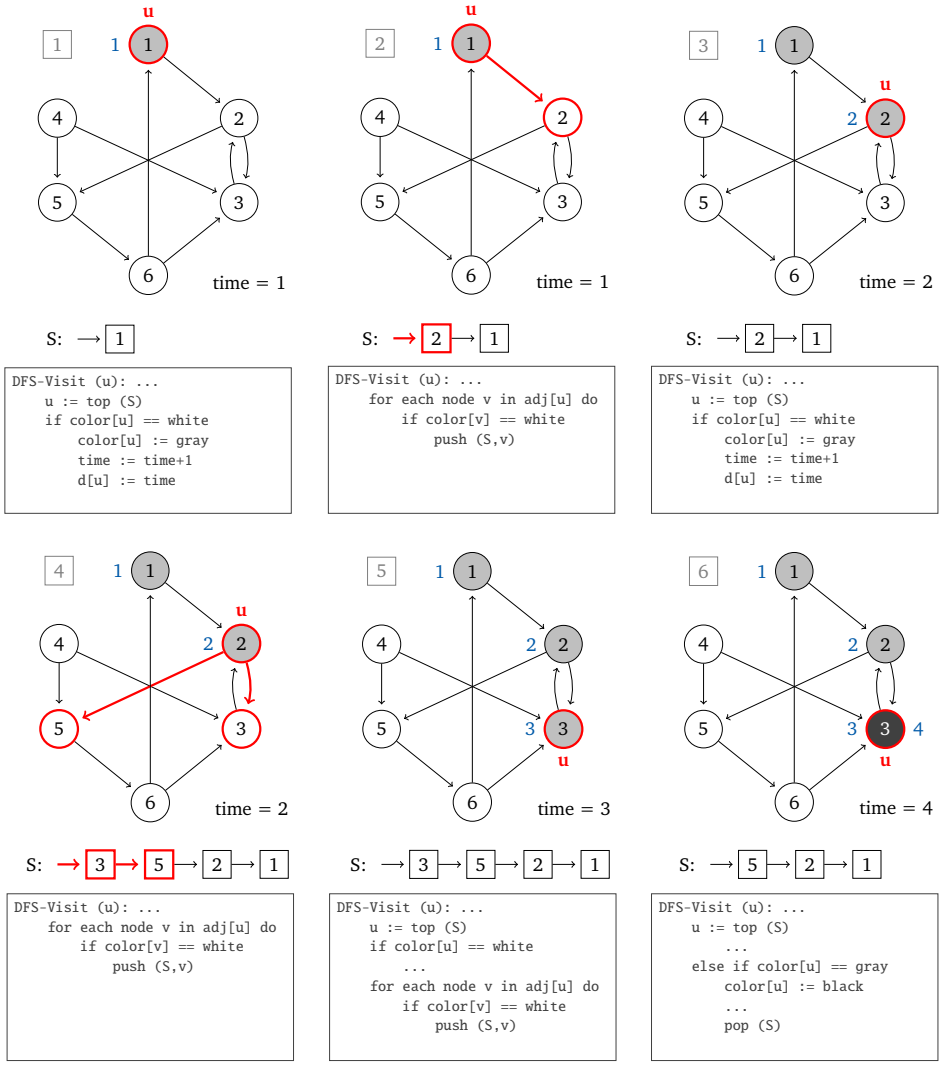


Abbildung 3.17: Schematischer Ablauf des Stack-basierten DFS-Algorithmus.
In der iterativen Version des DFS-Algorithmus wird die Ablaufkontrolle von einem Stack übernommen. (Zur Darstellung wurden einige repräsentative Schritte ausgewählt.) Die Analogie lässt sich wie folgt einsehen: Im rekursiven Algorithmus ruft die Funktion `DFS-Visit` sich selbst mit ihren Nachfolgeknoten auf. Die begonnenen Instanzen verweilen im Speicher und setzen den Programmfluss fort, sobald der aktuell laufenden rekursive Aufruf abgearbeitet wurden. Im iterativen Algorithmus wird die Reihenfolge der Knoten stattdessen dynamisch auf einem Stack abgelegt. Es wird stets der neuste Knoten zuerst abgearbeitet.

3.2.2 Breadth-First Search (BFS) Algorithmus

Die Breitensuche (*Breadth-First Search*) bildet die Grundlage vieler Algorithmen. Sie ist wie die Tiefensuche ein Algorithmus zum Traversieren eines Graphen. Dabei werden von jedem Knoten zunächst alle Nachbarknoten abgesucht. Sobald nicht weiter „in die Breite“ gesucht werden kann, werden die Wege ab den Nachbarknoten weiter verfolgt. Die genaue Reihenfolge hängt von der Repräsentation des Graphen ab.

Ziel ist es häufig, in einem gegebenen Graph $G = (V, E)$ von einer Quelle $s \in V$ alle Knoten $v \in V$ zu finden, die von s aus erreichbar sind. Dabei heißt ein Knoten v von s aus erreichbar, wenn es einen Weg von s zu v gibt. Die BFS-Algorithmus berechnet für alle Knoten v den Abstand $\delta(s, v)$ von s zu v. Dabei ist der Abstand die minimale Anzahl von Kanten auf einem Weg. Der Begriff Breitensuche kommt daher, dass der Algorithmus erst alle Knoten mit $\delta(s, v) < k$ vor den Knoten mit $\delta(s, v) = k$ bestimmt.

Analog zur Tiefensuche kann die Breitensuche auf einem abstrakten Datentyp operieren.

Definition 3.15. *Ein Queue ist ein abstrakter Datentyp, in dem sich Daten dynamisch speichern lassen. Es können beliebig Elemente hinzugefügt werden, jedoch kann immer nur das als erstes hinzugefügte Element wieder weggenommen oder angeschaut werden. Dies entspricht der first-in-first-out(FIFO) Strategie.*

Für Queues sind die folgenden Methoden mit Laufzeit $O(1)$ definiert:

`empty (Q)`	Gibt aus, ob die Queue leer ist.
`enqueue (Q,x)`	Hängt das Element x an die Queue Q an.
`dequeue (Q)`	Entfernt das erste Element der Queue Q.
`x := head (Q)`	Ließt den Inhalt des ersten Elements der Queue Q.

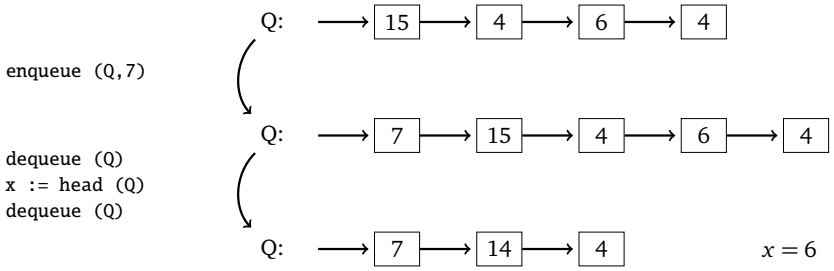


Abbildung 3.18: Beispiel zu Queue-Operationen.
Gegeben ist eine Queue (Warteschlange) $Q = (15, 4, 6, 4)$. Im ersten Schritt wird das Element 7 zur Queue hinzugefügt („an die Queue angehängt“). Im zweiten Schritt wird das älteste Element (aktuell die 4) aus der Queue entfernt, dann wird das älteste Element (aktuell die 6) ausgelesen und in der Variablen x gespeichert. Abschließend wird erneut das älteste Element (ebendiese 6) entfernt. Effektiv wurde in diesem Schritt also das zweit-älteste Element ausgelesen und die Queue um zwei Elemente verkürzt.

Nun lässt sich der folgende Algorithmus implementieren:

Listing 3.3: BFS-Algorithmus (*breadth-first search*), iterative Version

```
1   BFS (graph G):
2       for each node u in V do
3           color[u] := white
4       time := 0
5       Q := empty queue
6       for an arbitrary node u in V do
7           BFS-Visit (u);
8
9   BFS-Visit (node u):
10      enqueue (Q,u)
11      while not empty (Q) do
12          u := head (Q)
13          if color[u] != black
14              color[u] := black
15              time := time + 1
16              d[u] := time
17              for each node v in adj[u] do
18                  if color[v] == white
19                      enqueue (Q,v)
20                      color[v] := gray
21          else
22              time := time + 1
23              f[u] := time
24              dequeue (Q)
```

Der BFS-Algorithmus besteht aus den beiden Funktionen `BFS` und `BFS-Visit`.

Zu Beginn werden alle Knoten weiß gefärbt, um sie als unbetrachtet zu kennzeichnen. Die Zeit wird auf 0 gesetzt und eine leere Queue angelegt, die als Speicher für die noch abzuarbeitenden Knoten dienen soll. Die Zeit wird benötigt, um für jeden Knoten den Entdeckungszeitpunkt in dem Zeitstempel d und den Beendigungszeitpunkt in dem Zeitstempel f zu speichern. Durch `BFS` wird `BFS-Visit` für einen beliebigen Startknoten aufgerufen.

BFS-Visit speichert den aktuellen Knoten in der Queue und arbeitet solange die Queue nicht leer ist den ältesten Knoten der Queue ab. Wurde dieser Knoten noch nicht abgearbeitet (nicht schwarz), werden alle seine unentdeckten (weiß) Nachfolger in die Queue aufgenommen und grau markiert; der Knoten selbst wird schwarz gefärbt und sein Entdeckungszeitpunkt gespeichert. Ist der ausgewählte Knoten allerdings schon schwarz, wird er aus der Queue entfernt und sein Beendigungszeitpunkt gespeichert.

Da immer der älteste Knoten der Queue bearbeitet wird, wird erst in der Breite gesucht, bevor eine Stufe tiefer gegangen wird.

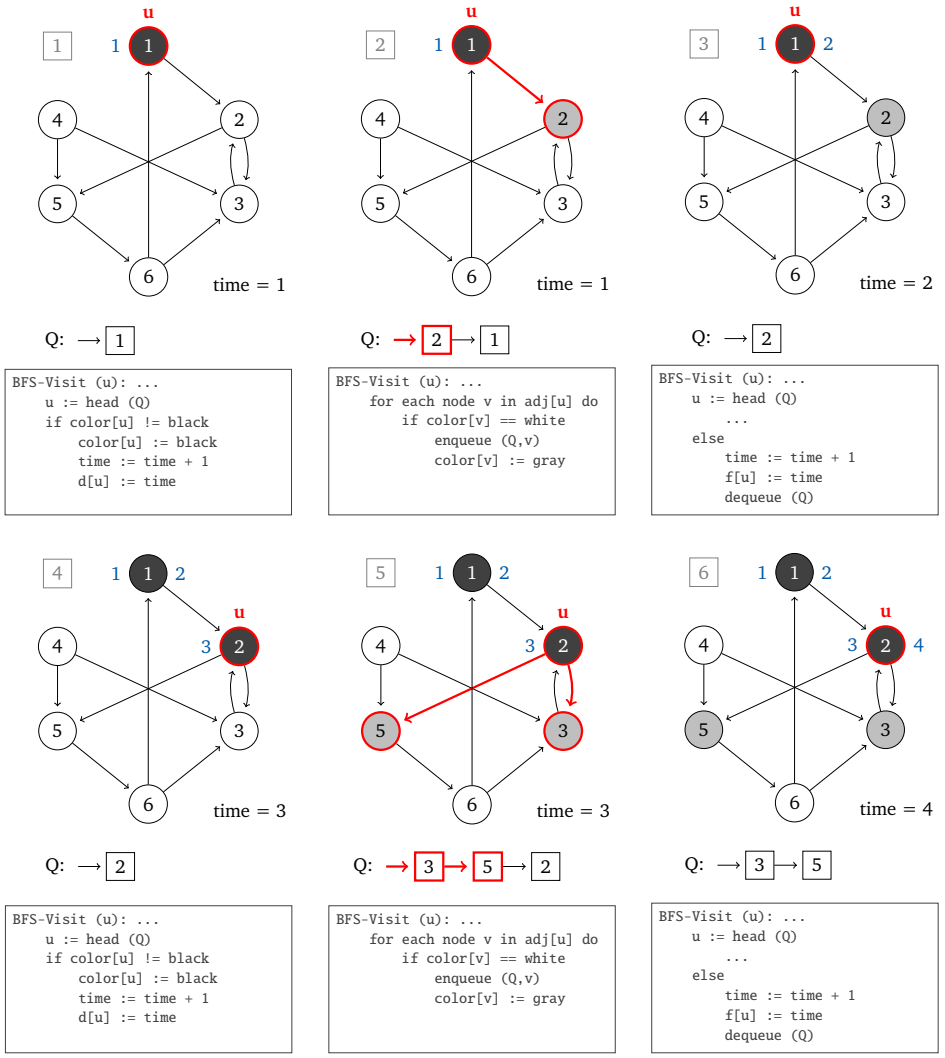


Abbildung 3.19: Schematischer Ablauf des Queue-basierten BFS-Algorithmus (Teil 1).
Diese zweiteilige Abbildung zeigt den vollständig Ablauf des iterativen DFS-Algorithmus auf einem Beispielgraphen. Unter jedem Teilschritt befindet sich ein Auszug des Pseudocodes 3.3, der die Änderungen seit dem letzten Schritt erläutern soll. Oben links vom Graphen wird die Schrittnummer angegeben, unten rechts vom Graphen ist die Zeitvariable dargestellt. Unter jedem Graphen befindet sich die aktuelle Queue.

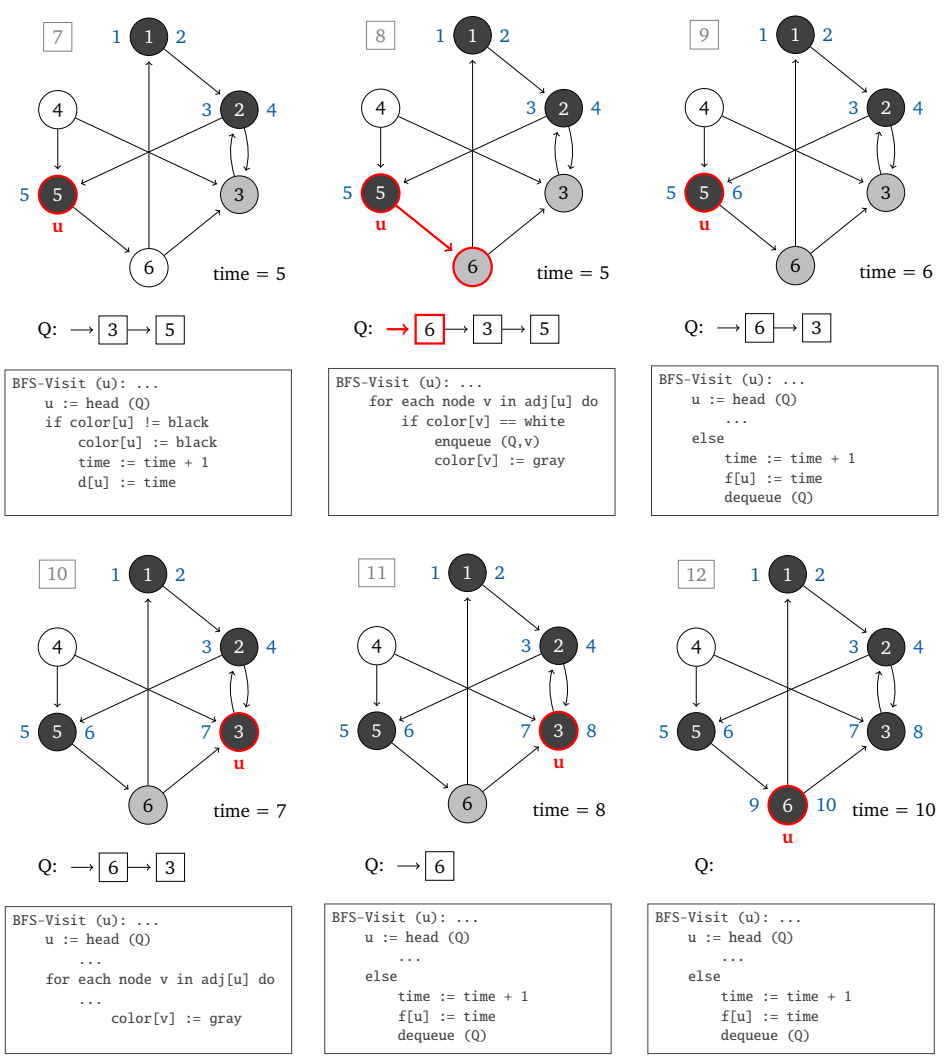


Abbildung 3.20: Schematischer Ablauf des Queue-basierten BFS-Algorithmus (Teil 2).
(Fortsetzung der vorangehenden Seite.) Aktive Knoten und Kanten (das sind solche, die gerade ausgewählt sind oder bearbeitet wurden) sind rot hervorgehoben. Links neben jedem Knoten steht sein Entdeckungszeitpunkt, rechts neben jedem Knoten steht sein Beendigungszeitpunkt. Aus der Queue geht hervor, in welcher Reihenfolge die Knoten ausgewählt werden. Knoten 4 wird (im Gegensatz zu DFS) niemals erreicht, was auch daran liegt, dass Zeile 6 modifiziert wurde.

Der oben beschriebene Algorithmus, lässt sich noch ein wenig umwandeln, sodass der Code etwas kompakter dargestellt wird:

Listing 3.4: BFS-Algorithmus (*breadth-first search*), schnellere (iterative) Version

```
1    BFS (graph G):
2        for each node u in V do
3            color[u] := white
4            dist[u] := infinity
5        Q := empty queue
6        for an arbitrary node u in V do
7            BFS-Visit (u);
8
9    BFS-Visit (node u):
10       enqueue (Q,u)
11       dist[u] := 0
12       while not empty(Q) do
13           u := head (Q)
14           for each node v in adj[u] do
15               if color[v] == white
16                   color[v] := gray
17                   enqueue (Q,v)
18                   dist[v] := dist[u] + 1
19                   pred[v] := u
20           color[u] := black
21           dequeue (Q)
```

Hier wird die while-Schleife so umgeformt, dass ein Knoten sofort als abgearbeitet (schwarz) betrachtet und aus der Queue gelöscht wird, sobald alle seinen Nachbarn in der Queue gespeichert wurden. Somit wird die Queue in weniger Schleifendurchläufen geleert. Dabei werden andere Entdeckungs- und Beendigungszeitpunkte erreicht, was aber keine weiteren Folgen hat.

Satz 3.6 (Satz über kürzeste Wege der Breitensuche)**.** *Sei $\delta(s,v)$ die kürzeste Entfernung von s nach v, also die minimale Anzahl von Kanten, über die man laufen muss, um von s nach v zu kommen. Wenn es keinen Weg von s nach v gibt, sei $\delta(s,v)=\infty$. Der BFS-Algorithmus berechnet für alle Knoten $v \in V$ die kürzeste Entfernung von s nach v.*

Für den Beweis werden drei Lemmata benötigt:

Satz 3.7 (Erstes BFS-Lemma)**.** *Sei $G=(V,E)$ ein gerichteter oder ungerichteter Graph und sei $s \in V$ ein beliebiger Knoten. Dann gilt für jede Kante $(u,v) \in E$:* $\delta(s,v) \leq \delta(s,u)+1$

Beweis. Wenn u nicht von s erreichbar ist, gilt $\delta(s,u)=\infty$. Damit ist klar, das die Aussage erfüllt ist. Wenn u erreichbar ist, so ist es auch v. Der kürzeste Weg von s nach v kann aber nicht länger sein, als der kürzeste Weg von s nach u zuzüglich der Kante (u,v). □

Satz 3.8 (Zweites BFS-Lemma)**.** *Sei $G=(V,E)$ ein Graph. Sei der BFS-Algorithmus auf G mit Startknoten s ausgeführt worden. Dann gilt für jeden Knoten $v \in G$:* $\text{dist}[v] \geq \delta(s,v)$

Beweis. Der Beweis erfolgt durch Induktion über die Anzahl der `enqueue`-Aufrufe in `BFS-Visit`:

Induktionsstart Sei s soeben in Q platziert worden. Dann wird $\text{dist}(s)=0$ gesetzt und auch nicht wieder verändert. Alle anderen Werte sind auf ∞ gesetzt. Außerdem gilt $\delta(s,s)=0$.

Induktionsvoraussetzung Für alle $v \in V$, die bereits entdeckt wurden, gilt: $\text{dist}[v] \geq \delta(s,v)$.

Induktionsschluss Sei v von u aus entdeckt worden, dann gilt: $\text{dist}[u] \geq \delta(s,u)$. Wegen $\text{dist}[v] = \text{dist}[u]+1$ in `BFS-Visit`, gilt $\text{dist}[v] = \text{dist}[u]+1 \geq \delta(s,u)+1 \geq \delta(s,v)$ nach dem Ersten BFS-Lemma. Danach wird $\text{dist}[v]$ nicht mehr verändert.

□

Satz 3.9 (Drittes BFS-Lemma)**.** *Sei $G=(V,E)$ ein Graph. Sei während der BFS-Algorithmus arbeitet: $Q=\langle v_1, v_2, \ldots, v_r\rangle$ und sei v_1 gegeben durch $head(Q)$. Dann gilt für $i=1,2,\ldots,r-1$:* $\text{dist}(v_r) \leq \text{dist}(v_1)+1$ *und* $\text{dist}(v_i) \leq \text{dist}(v_{i+1})$

Beweis. Der Beweis erfolgt durch Induktion über die Durchläufe der `for`-Schleife:

Induktionsanfang Nach dem ersten Schleifendurchlauf ist $v_1 = u = s$ und $v_r = v$. Damit $\text{dist}(v_1)=0$ und $\text{dist}(v_r)=1$.

Induktionsvoraussetzung Für die ersten n Schleifendurchläufe gilt: $\text{dist}(v_r^{(n)}) \leq \text{dist}(v_1^{(n)})+1$ und $\text{dist}(v_i^{(n)}) \leq \text{dist}(v_{i+1}^{(n)})$ nach dem jeweiligen Schleifendurchlauf.

Induktionsschluss Wenn $v_r^{(n+1)}$ eingefügt wird, ist $v_r^{(n+1)}$ Nachfolger von v_1. Also: $\text{dist}(v_r^{(n+1)}) = \text{dist}(v_1^{(n)})+1 \leq \text{dist}(v_1^{(n+1)})+1$ durch `BFS-Visit`. Es ist aber auch $\text{dist}(v_r^{(n)}) \leq \text{dist}(v_1^{(n)})+1 = \text{dist}(v_1^{(n+1)})+1 (= d(u)+1 = d(v)) = \text{dist}(v_r^{(n+1)})$

□

Beweis des Satzes über kürzeste Wege der Breitensuche.

1. Falls v nicht erreichbar ist, wird v nicht entdeckt und $\text{dist}[v]=\infty$.
2. Definiere $V_k := \{\, v \in V \mid \delta(s,v)=k \,\}$. Der Beweis erfolgt durch Induktion über k.

   Induktionsvoraussetzung für alle Knoten mit $\delta(s,v)<k$ gilt die Behauptung.

   Induktionsanfang $k=0 : V_0=\{s\}$ und $\text{dist}(s)=0$.

   Induktionsschluss $k-1 \to k$: Sei $v \in V_k$. Wegen dem Dritten BFS-Lemma gilt $\text{dist}(v_i) \leq \text{dist}(v_{i+1})$, wenn v_i vor v_{i+1} in Q eingefügt wurde. Wegen dem Zweiten BFS-Lemma gilt $\text{dist}[v] \geq k$. Deshalb muss v in Q eingefügt werden, nachdem alle $u \in V_{k-1}$ in Q eingefügt wurden. Da $\delta(s,v)=k$ gilt, gibt es einen Weg der Länge k von s nach v und somit einen Knoten $u \in V_{k-1}$, so dass $(u,v) \in E$.

   Wird jetzt nochmal die Schleife aus `BFS-Visit` betrachtet folgt die Behauptung.

□

3.3 Kürzeste Wege

Ein sehr große Rolle in der Anwendung spielen *Kürzeste Wege Algorithmen*. Diese sollen für einen gegebenen Graph den kürzeste Weg zwischen bestimmten Knoten finden. Dabei können für die Kanten des Graphen auch Gewichte vorgegeben werden, die angeben sollen wie „teuer“ es ist, diese Kante zu benutzen.

Definition 3.16. *Ein gewichteter Graph G ist ein Tupel (V,E) mit einer Gewichtsfunktion $f : E \to \mathbb{Q}$ mit $E \subseteq V \times V$, d.h. f ordnet allen Kanten des Graphen G ein Wert aus $\mathbb{Q}$ zu, dieser Wert heißt Gewicht der Kante.*

Definition 3.17. *Sei $G = (V,E)$ ein gewichteter Graph und $u, v \in V$. Dann ist der kürzeste Weg von u nach v der Weg mit kleinstmöglichem Gewicht. Das Gewicht eines Weges ist dabei die Summe aller Kantengewichte auf diesem Weg.*

Satz 3.10. *Sei $G = (V,E)$ ein gerichteter Graph mit nicht-negativen Kantengewichten, d.h. mit Gewichtsfunktion $f : E \to \mathbb{Q}^+$, und sei $s, v, t \in V$. Wenn ein Weg w von s nach t ein kürzester Weg ist und v ein Knoten auf diesem Weg, dann sind auch die Teilwege von s nach v und v nach t kürzeste Wege.*

Beweis. Sei $\widetilde{w}$ ein kürzerer Weg von s nach v mit $\widetilde{w} \notin w$. Dann ist der Weg von s nach t der als ersten Teilweg $\widetilde{w}$ und als zweiten Teilweg den Teilweg von v nach t aus w enthält kürzer als w. Widerspruch! $\Rightarrow$ Es gibt keinen kürzeren Weg $\widetilde{w}$.
Analog für den Weg von v nach t. □

3.3.1 Dijkstra Algorithmus

Der *Dijkstras-Algorithmus* kann für einen gerichteten Graphen mit nicht-negativen Kantengewichten für alle Knoten den kürzesten Weg vom Startknoten und dessen Gewicht berechnen.

Das Grundprinzip ist es, zu überprüfen, ob die Distanz eines Knotens zum Startknoten kleiner wird, wenn man statt des alten Weges, den Weg über einen anderen Nachbarn nimmt, dessen kürzester Weg bereits bekannt ist. Dazu werden zuerst alle Knoten in zwei Mengen aufgeteilt: Eine Menge S, die alle Knoten enthält, deren entgültige Entfernung zum Startknoten bereits bekannt ist. Und eine Menge A, die alle Knoten enthält, deren entgültige Entfernung noch nicht bekannt ist. Zu Beginn ist somit die Menge S leer und M entspricht der Menge aller Knoten (V).

Die Distanz des Startknoten s wird auf 0 gesetzt und die Distanz der anderen Knoten auf ∞, da sie noch keine berechnete Distanz zum Startknoten haben. Dann wird solange A nicht leer ist der Knoten u aus A genommen, der die kleinste Distanz zum Startknoten s hat, und zu S hinzugefügt. Für alle seine Nachbarn v wird geprüft, ob ihre bisherige Distanz zu s größer ist als die Summe der Distanz von u und des Kantengewichts der Kante von u nach v. Ist dies der Fall wird die Distanz von v gleich der Summe gesetzt und u als Vorgänger von v gespeichert. Andernfalls wird nichts verändert und beim nächsten Nachbarn fortgefahren.

Für die *Laufzeit des Dijkstras Algorithmus* ergibt sich $O(E \cdot O(\text{dist}[v] := \text{dist}[u] + f(u,v)) + |V| \cdot O(u := \arg\min\{\text{dist}[a] | a \in A\}))$. Einfacher ergibt sich $O(|E| + |V|^2)$

Listing 3.5: Dijkstra-Algorithmus

```
1   DijkstraInit (G,s):
2       for each node v in V(G) do
3           pred[v] := nil
4           if v == s
5               dist[v] := 0
6           else
7               dist[v] := infinity
8
9   Dijkstra (G,s):
10      DijkstraInit (G,s)
11      S := empty set
12      A := V(G)
13      while not empty (A) do
14          u := argmin { dist[a] | a in A }
15          S := S union {u};
16          for each node v in adj[u] do
17              if dist[v] > dist[u] + f (u,v)
18                  dist[v] := dist[u] + f (u,v)
19                  pred[v] := u
```

Satz 3.11 (Satz über die Korrektheit des Dijkstra-Algorithmus)**.** *Verwendet man den Dijkstra-Algorithmus auf einem gewichteten, gerichteten Graphen mit nicht-negativer Gewichtsfunktion w und Startknoten s, so gilt $\forall u \in S$:* $\text{dist}[u] = \delta(s,u)$.

Für den Beweis werden drei Lemmata benötigt:

Satz 3.12 (Erstes Dijkstra-Lemma)**.** $\forall v \in V$ *gilt* $\text{dist}[v] \geq \delta(s,v)$.

Beweis. Der Beweis erfolgt durch Induktion über die Anzahl der Aufrufe in der `for`-Schleife:

Induktionsstart Zu Beginn ist $\text{dist}[s] = 0$ und $\text{dist}[u] = \infty$ für alle anderen $u \in S$. $\Rightarrow \text{dist}[u] \geq \delta(s,v)$ und $0 = \text{dist}[s] = \delta(s,s)$.

Induktionsvoraussetzung Die Behauptung gilt bis zum k-ten Aufruf der `for`-Schleife.

Induktionsschluss Die Aussage wird nun durch einen Widerspruch gezeigt:
Annahme: Sei v der erste Knoten, für den die Behauptung nicht gilt und sei u der Vorgänger von v. v wurde also durch den $(k+1)$-ten Aufruf der `for`-Schleife erzeugt. Dann gilt nach diesem $(k+1)$-ten Aufruf: $\text{dist}[u] + f(u,v) = \text{dist}[v]$ und nach Vorraussetzung $\text{dist}[v] < \delta(s,v)$.
Wegen der Kürzeste-Wege-Eigenschaft gilt: $\delta(s,v) \leq \delta(s,u) + f(u,v)$. Da sonst der Weg von s über u nach v kürzer wäre als der kürzeste Weg von s nach v. Insgesamt folgt somit $\text{dist}[u] + f(u,v) < \delta(s,u) + f(u,v)$. Also $\text{dist}[u] < \delta(s,u)$. Da aber $\text{dist}[u]$ im $(k+1)$-ten Aufruf gar nicht verändert wurde, muss nach der Induktionsvorraussetzung $\text{dist}[u] \geq \delta(s,u)$ gelten. $\Rightarrow$ Widerspruch!
Die Annahmme $\text{dist}[v] < \delta(s,v)$ war falsch. Somit muss die Behauptung auch für v gelten. □

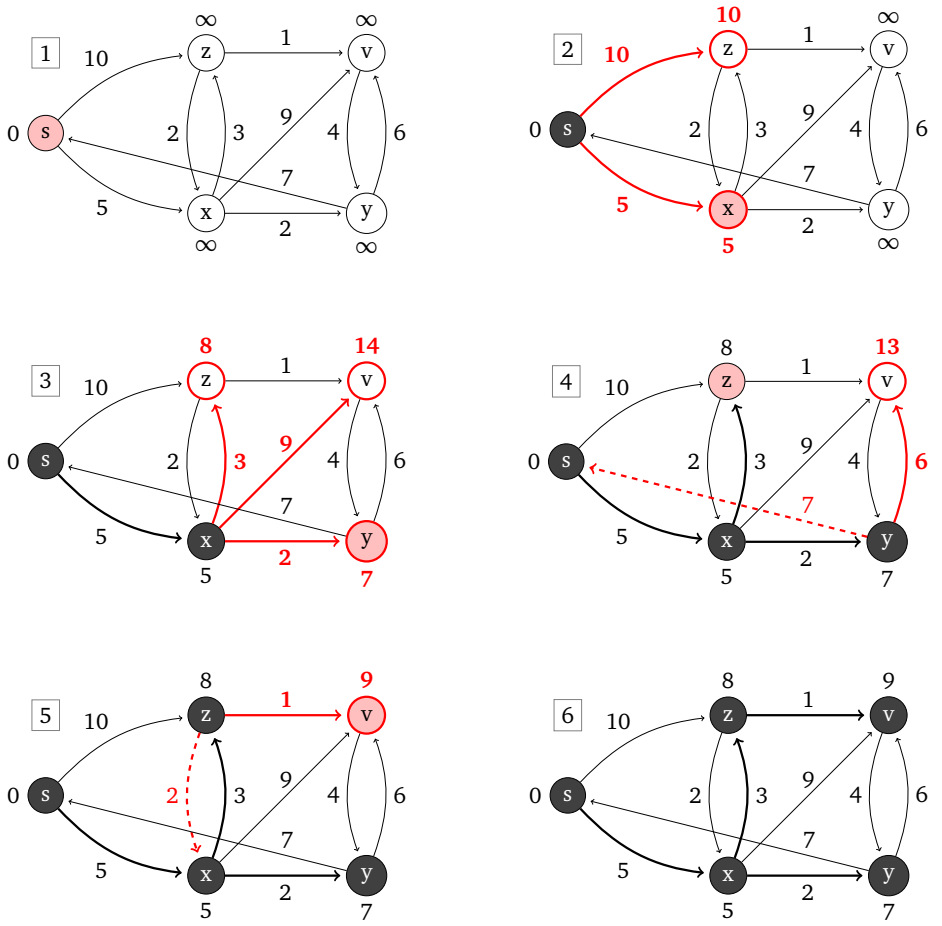


Abbildung 3.21: Schematischer Ablauf des Dijkstra-Algorithmus.
Der Dijkstra-Algorithmus unterteilt die Knoten in zwei Mengen: Knoten, deren kürzeste Entfernung schon feststeht (schwarz), und solche, deren kürzeste Entfernung noch zu bestimmen ist (weiß). Pro Iterationsschritt wird ein Knoten der weißen Menge in die schwarze Menge überführt, und zwar derjenige mit dem aktuell kleinsten Distanzwert (rot gefüllt). Von diesem aus werden die Entfernungen zu seinen Nachbarknoten berechnet. Falls die berechnete Distanz kürzer als die dort hinterlegte (zuletzt beste) Distanz ist, wird sie aktualisiert (rot umrahmt) - sonst geschieht nichts (rot gestrichelt). Es werden so viele Iterationen durchlaufen, bis die kürzeste Entfernung zu allen Knoten bekannt ist. Der kürzeste Weg zwischen dem Startknoten s und einem beliebigen Endknoten (z.B. v) wird dann durch die Vorgängerbeziehung (`pred` im Pseudocode, dicke Kanten in der Abbildung) eindeutig beschrieben.

Um nun die Korrektheit zu zeigen, also dass $u \in S$: $\text{dist}[u] = \delta(s,u)\ \forall u \in V$, soll ein Widerspruchsbeweis geführt werden. Es wird $\tilde{u}$ definiert als erster Knoten für den $\text{dist}[\tilde{u}] \neq \delta(s,\tilde{u})$ gilt. Die Korrektheit ist folglich bewiesen, wenn $\text{dist}[\tilde{u}] = \delta(s,\tilde{u})$ gezeigt werden kann.
Dazu werden folgende Vorüberlegungen getroffen:

- $\tilde{u} \neq s$ denn $\text{dist}[s]$ wird korrekt eingeführt und nicht mehr verändert. (Da $\text{dist}[s] = \delta(s,s) = 0$)
- Vor dem Einfügen von $\tilde{u}$ in S gilt $S \neq \emptyset$, da $s \in S$, und $A \neq \emptyset$, da der Algorithmus sonst nicht mehr laufen würde.
- Es gibt einen Weg von s nach $\tilde{u}$, da sonst $\text{dist}[\tilde{u}] = \delta(s,\tilde{u}) = \infty$. $\Rightarrow$ Es gibt einen kürzesten Pfad p von s nach $\tilde{u}$.

Betrachtet man nun einen Zeitpunkt, bevor $\tilde{u}$ in S eingeführt wird, also wenn $\tilde{u} \in A (= V \setminus S)$, so folgt mit den Vorüberlegungen: Sei y der erste Knoten entlang p so dass $y \in V \setminus S$ und sei $x \in S$ der Vorgänger von y. Dann lässt sich p aufteilen in p_1 und p_2, so dass p_1 komplett in S liegt.

Satz 3.13 (Zweites Dijkstra-Lemma). *Es gilt* $\text{dist}[y] \leq \delta(s,y)$.

Beweis. Da $x \in S$ und y im kürzesten Weg vor u liegt, wurde y mit Hilfe der `for`-Schleife auf den endgültigen Wert gesetzt, bevor dies für u passierte. Damit gilt:

$$\begin{aligned}
\text{dist}[y] &\leq \text{dist}[x] + f(x,y) && \text{(folgt aus der for-Schleife)}\\
&= \delta(s,x) + f(x,y) && \text{(Wegen } x \in S \text{ und der Annahme, dass für alle Knoten}\\
& && \text{vor } \tilde{u} \text{ die Korrektheit gilt)}\\
&= \delta(s,y) && \text{(weil } s \to x \to y \text{ der kürzeste Weg ist)}
\end{aligned}$$

□

Satz 3.14 (Drittes Dijkstra-Lemma). *Wenn u in S eingefügt wird, gilt* $\text{dist}[y] = \delta(s,y)$.

Beweis. Der Beweis folgt direkt aus dem Zusammenschluss des 1. & 2. Dijkstras Lemma. □

Beweis des Satzes über die Korrektheit des Dijkstra-Algorithmus.

$\text{dist}[y] = \delta(s,y) \leq \delta(s,u) \leq \text{dist}[u]$, dabei folgt die Gleichheit aus dem Dritten Dijkstra Lemma, die erste Abschätzung gilt, da y auf dem kürzesten Weg von s nach $\tilde{u}$ liegt, und die zweite Abschätzung folgt aus dem Ersten Dijkstra Lemma.

$\text{dist}[\tilde{u}] \leq \text{dist}[y]$, da $y, \tilde{u} \in A$ und in dem Moment, indem $\tilde{u}$ in S eingefügt wird, gilt nach dem Algorithmus $\tilde{u} = \arg\min\{\text{dist}[a] | a \in A\}$.

Insgesamt folgt $\text{dist}[\tilde{u}] \leq \text{dist}[y] \leq \delta(s,\tilde{u}) \leq \text{dist}[\tilde{u}]$, also $\text{dist}[\tilde{u}] = \delta(s,\tilde{u})$ □

Die Laufzeit des Dijkstras-Algorithmus lässt sich jedoch noch verkürzen. Dafür wird ein neuer abstrakter Datentyp eingeführt, der *Heap*. Die *Laufzeit* kann dann auf $O(|E| \cdot \log|V| + |V| \cdot \log|V|)$ oder sogar $O(|E| + |V| \cdot \log|V|)$ gebracht werden.

Heaps

Definition 3.18. *Ein Heap ist ein abstrakter Datentyp, der aus einem Binärbaum besteht, dessen Knoten je einem Element entsprechen. Der Baum wird dabei ebenenweise aufgefüllt und meist in einem Array gespeichert. Die Anzahl der Elemente in einem Heap wird häufig mit* `size` *bezeichnet. Es gilt die Heap-Eigenschaft: Die Werte in den Nachfolgern v_1, v_2 eines Knotens v sind größer als das Element in dem Knoten v selbst.*

Für Heaps sind die folgenden Methoden definiert:

`BuildHeap`	erstellt aus einer Menge von Elementen einen Heap.
`Insert`	fügt ein zusätzliches Element in den Heap ein.
`ExtractMin`	nimmt das kleinste Element aus dem Heap heraus.
`Heapify`	stellt die Heapeigenschaft her.
`DecreaseKey(A,i,newkey)`	verkleinert das Element i des Heaps A auf newkey und stellt die Heapeigenschaft her.

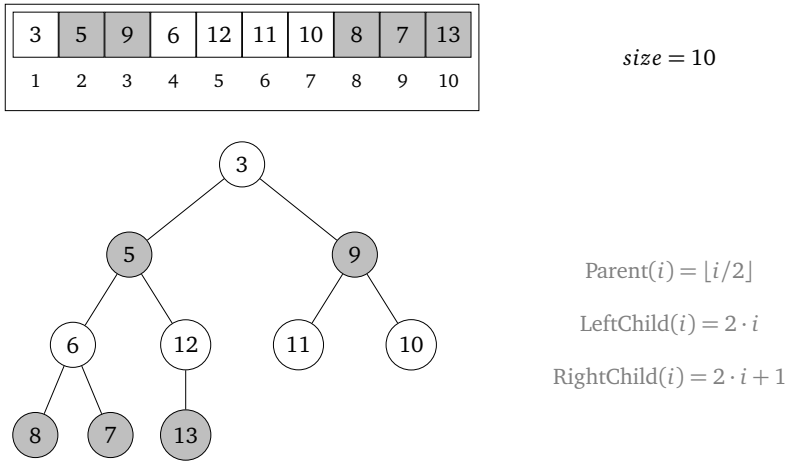


Abbildung 3.22: Schematische Darstellung eines Heaps.
Ein Heap ist ein abstrakter Datentyp, der eine partielle Ordnung einem Array definiert. Man kann ihn als einen Binärbaum darstellen, in dem die beiden Nachfolger eines Knotens stets größere Werte besitzen als der Knoten selbst (*Heap-Eigenschaft*). Heaps spielen eine wichtige Rolle für verschiedene Algorithmen, da man ihr kleinstes Element sehr schnell auslesen kann. (Es ist immer der Wurzelknoten! Der Rechenaufwand besteht „nur" darin, die Heap-Eigenschaft zu erhalten. Dies ist aber deutlich effizienter, als eine vollständig sortierte Liste zu speichern.) Heaps müssen nicht als Graphenstruktur im Rechner modelliert werden. Da jeder Knoten (bis auf die Blätter) genau zwei Nachfolger hat, können alle Operationen auf einem eindimensionalen Array modelliert werden.

Die Implementierungen dieser fünf Heap-Methoden werden nun zum besseren Verständnis erläutert:

Heapify(A,i) stellt unter der Annahme, dass die Heap-Eigenschaft außer in der Wurzel i überall erfüllt ist, diese für den ganzen Heap her. Dafür wird für den linken und rechten Nachfolger der Wurzel (wenn sie existieren) geprüft, ob der enthaltene Wert kleiner ist als der Wert der Wurzel. Ist dies der Fall, so ist die Heap-Eigenschaft nicht gegeben, also wird dieser kleinere Wert mit dem Wert der Wurzel vertauscht. Nun wird Heapify für den Teilbaum des getauschten Wertes erneut aufgerufen, um die dort durch den Tausch möglicherweise zerstörte Heap-Eigenschaft wieder herzustellen.
Dass Heapify wirklich die Heap-Eigenschaft herstellt lässt sie folgendermaßen zeigen:
Heapify macht aus den Bäumen der letzten 2 Ebenen sicher Heaps. Werden nun 2 Heaps zusammengefasst, indem ein Vorgängerknoten für die 2 vorhandenen Subheaps geschaffen wird, können 2 Fälle auftreten:

1. Der Wert im neuen Knoten ist kleiner als die Nachfolgerwerte. Dann ist die Heapeigenschaft erfüllt. Oder
2. Der Wert im neuen Knoten ist größer als einer der Nachfolgerwerte. Dann ist die Heapeigenschaft an v verletzt, sonst nirgends. Der Heap wird dann durch heapify repariert.

ExtractMin(A) nimmt das Wurzelement heraus, da es nach der Heap-Eigenschaft das kleinste Element im Heap A ist. Dann wird das size-te Element in die Wurzel gepackt und size um eins verringert. Zuletzt wird Heapify für den gesamten Heap aufgerufen, um die Heap-Eigenschaft wieder herzustellen.

Insert(A,x) erhöht size um eins und fügt das Element x an der Stelle ein, wo es die Heap-Eigenschaft nicht zerstört. Dafür wird bei dem letzten Element im Heap A angefangen zu prüfen, ob der Vorfahre des Elements größer ist als x, solange dies der Fall ist, wird der Vorfahre auf den Platz des geprüften Elements geschoben, dann wird x an der Stelle eingefügt, wo der Vorfahre kleiner ist als x.
Die Heap-Eigenschaft wird beibehalten, da wenn ein Element x eines Knotens v in einen Nachfolger kopiert wird, dann deshalb, weil das neu einzufügende Element kleiner als x ist. x wird also die Heap-Eigenschaft am Ende nicht zerstören. Die heruntergezogenen Elemente aber auch nicht. Denn die Heap-Eigenschaft war schon vor dem Herunterziehen erfüllt, dann wird sie auch durch ein kleineres Element nicht zerstört werden.

BuildHeap(A) erstellt aus einem Array (Baum) in dem alle n Elemente unsortiert vorliegen einen Heap. Dafür wird size auf n gesetzt und dann werden alle Teilbäume von oben nach unten durch Heapify auf Heap-Eigenschaft gebracht.

DecreaseKey(A, i, newkey) verkleinert das Element i des Heaps A auf newkey und stellt dann die Heap-Eigenschaft wieder her, indem für alle Vorgänger von i überprüft wird, ob sie noch der Heap-Eigenschaft entsprechen, falls nicht, werden sie mit ihrem Nachfolger getauscht. Da i hier nur verkleinert werden kann, müssen auch nur die Vorgänger von i überprüft werden, da alle Nachfolger bereits vor der Verkleinerung größer waren.
Dass die Heap-Eigenschaft beibehalten wird folgt wie für Insert.

Listing 3.6: Heapify (Knotenvertauschung eines binären Baums)

```
1   Heapify (A,i):
2       if Left (i) <= size and A[Left (i)] < A[i]
3           smallest := Left (i)
4       else
5           smallest := i
6       if Right (i) <= size and A[Right (i)] < A[smallest]
7           smallest := Right (i)
8       if smallest != i
9           exchange (A[i], A[smallest])
10          Heapify (A, smallest)
```

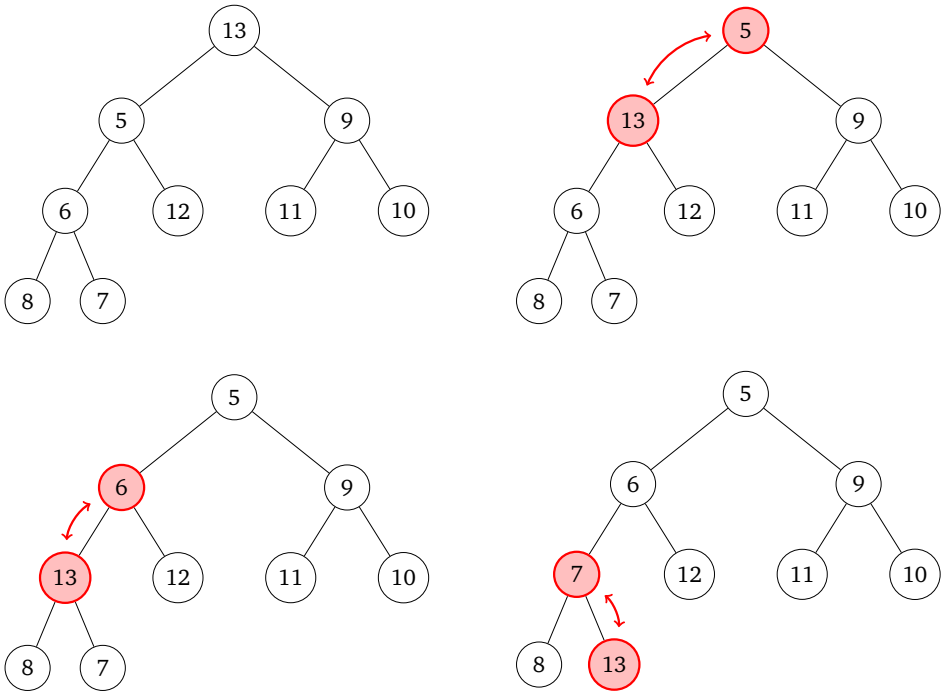


Abbildung 3.23: Schematischer Ablauf der Heapify-Funktion.
Aufgabe der Heapify-Funktion ist es, in einem gegebenen binären Baum die Heap-Eigenschaft („kleinere Werte stehen oben“) herzustellen. Zunächst wird der Wurzelknoten ausgewählt und mit seinen Unterknoten verglichen. Sollte sein Wert zu groß sein, wird er mit dem kleineren der beiden Knoten vertauscht. Anschließend fährt die Funktion mit den Unterknoten fort, bis alle Knoten, die die Heap-Eigenschaft verletzt haben, nach unten „durchgetauscht“ worden sind.

Für `Heapify` lässt sich die einfache Laufzeitschranke $O(n \log n)$ bestimmen. Genauer sogar $O(n)$.

In dem Dijkstras Algorithmus lässt sich nun die Menge A durch einen Heap ersetzen. Werden dann noch $\arg\min\{\text{dist}[a] \mid a \in A\}$ durch `ExtractMin(A)` ersetzt und geänderte dist-Werte durch `DecreaseKey` auch im Heap vermerkt, so ergibt sich diese schnellere Dijkstras Variante, die vom Prinzip her gleich arbeitet wie die vorherige und nur den nächsten zu bearbeitenden Knoten mit Hilfe des Heap leichter findet.

Weitere Wege-Algorithmen

Neben den bereits kennengelernten gibt es noch weitere Wege-Probleme:

- Kürzeste Wege in gerichteten Graphen mit allgemeinen, also positiven und negativen, Gewichten
  → *Bellman-Ford Algorithmus* gibt kürzeste Wege ausgehend von einem Startknoten s, oder „es gibt negativen Kreis, der von Startknoten s aus erreichbar ist“ aus.
  → ganz allgemein sind Kürzeste-Wege Probleme mit allgemeinen Gewichten NP-vollständig
- Kürzeste Wege in gerichteten Graphen ohne Kreise
  → Es gibt einen schnellen Algorithmus, auch bei allgemeinen Kantengewichten.
- *Längste Wege*
  - gibt es einen einfachen Pfad von zwei ausgezeichneten Knoten s nach t im Graphen G, so dass jeder Knoten genau einmal besucht wird? Dieses Problem heißt *Hamiltonpfadproblem* und ist NP-vollständig.
  - Gibt es einen *Hamiltonkreis*, also einen Hamiltonpfad von s nach s im Graphen G? Auch dieses Problem ist NP-vollständig

3.3.2 Bellman-Ford Algorithmus

Der eben bereits erwähnte Bellman-Ford Algorithmus kann auch bei negativen und positiven Kantengewichten einen kürzesten Weg finden. Dies funktioniert nur, wenn der Graph keinen negativen Kreis enthält, ansonsten wird ausgegeben, dass es einen negativen Kreis gibt. Dabei läuft der Algorithmus $(|V|-1)$-mal und überprüft jedes mal für alle Kanten, ob der Weg bis zum Kantenende durch diese Kante verkürzt werden kann.

Listing 3.7: Bellman-Ford Algorithmus

```
1    BellmanFordInit (G,s):
2        for each node v in V(G) do
3            pred[v] := nil
4            if v == s
5                dist[v] := 0
6            else
7                dist[v] := infinity
8
9    BellmanFord (G,s):
10       BellmanFordInit (G,s)
11       for i := 1 to |V|-1 do
12           for each edjge (u,v) in E do
13               if dist[v] > dist[u] + f(u,v)
14                   dist[v] := dist[u] + f(u,v)
15                   pred[v] := u
16       for each edge (u,v) in E do
17           if dist[v] > dist[u] + f(u,v)
18               return false
19       return true
```

Der Bellman-Ford Algorithmus läuft $(|V|-1)$ -mal die erste `for`-Schleife durch und prüft dabei jedesmal für jede Kante, ob der dist-Wert des Kantenendes größer ist als die Summe des dist-Wert des Kantenanfangs und des Kantengewichts. Ist dies der Fall, so wird der dist-Wert des Kantenendes der Summe des dist-Wert des Kantenanfangs und des Kantengewichts gleichgesetzt. Dabei kommt es vor, dass für einen Durchlauf nur sehr wenige Werte geändert werden. Bei dem k-ten Durchlauf wird also der optimale k-lange Weg gefunden. Da pro Durchlauf der „optimale Weg“ somit um mindestens eine Kante wächst und ein zyklenfreier Weg maximal $|V|-1$ Kanten enthält ist der kürzeste Weg nach $|V|-1$ Durchläufen gefunden. (Sofern er existiert und keinen negativen Kreis enthält.)
Nach diesen Durchläufen wird mit der zweiten `for`-Schleife überprüft, ob der Graph auch keine negativen Kreise enthält und das Ergebnis somit korrekt ist. Dafür wird erneut für jede Kante geprüft, ob der dist-Wert des Kantenendes größer ist als die Summe des dist-Wert des Kantenanfangs und des Kantengewichts. Tritt dieser Fall ein, so gibt es einen negativen Kreis.

Um die Korrektheit des Bellman-Ford Algorithmus zu zeigen, nutzen wir folgendes Lemma:

Satz 3.15 (Erstes Bellman-Ford Lemma)**.** *Sei $G=(V,E)$ ein gewichteter gerichteter Graph mit Startknoten s und Gewichtsfunktion $f: E \to \mathbb{R}$, und habe G keine negativen Kreise, d.h. G habe keine Kreise, deren Summe der Kantengewichte ihrer Kanten negativ ist, die von s aus erreicht werden können. Dann gilt nach Beendigung des Bellman-Ford Algorithmus:* $\text{dist}[v]=\delta(s,v)$ *für alle Knoten v, die von s aus erreichbar sind.*

Beweis. Sei v ein Knoten, der von s aus erreichbar ist, und sei $p=\langle s=v_0, v_1, ..., v=v_k\rangle$ eine kürzester Weg von s nach v. p ist ein einfacher Weg und deshalb gilt $k \leq |V|-1$. Durch Induktion soll nun gezeigt werden, dass nach dem i-tem Durchlauf der zweiten `for`-Schleife $\text{dist}[v_i]=\delta(s,v_i)$ gilt.

Induktionsanfang Zu Beginn gilt: $\text{dist}[s]=0=\delta(s,s)$. Da es nach Voraussetzung keinen negativen Kreis gibt, wird $\text{dist}[s]$ nicht mehr verändert, denn 0 ist bei positven Gewichten bereits minimal.

Induktionsvorraussetzung $\text{dist}[v_{i-1}]=\delta(s,v_{i-1})$ nach dem $(i-1)$-tem Durchlauf der zweiten `for`-Schleife.

Induktionsschluss Der Schluss von $i-1$ auf i folgt analog zum Beweis des 2. Dijkstras Lemma.

□

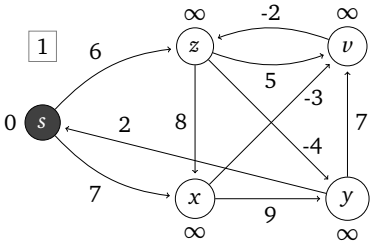
Satz 3.16 (Satz über die Korrektheit des Bellman-Ford Algorithmus)**.** *Werde der Bellman-Ford Algorithmus auf einem gerichteten, gewichteten Graphen $G=(V,E)$ mit Startknoten s ausgeführt. Falls G einen negativen Kreis enthält, der von s aus erreichbar ist, gibt der Algorithmus* `false` *zurück. Ansonsten gibt er* `true` *zurück und es gilt für alle Knoten* $\text{dist}[v]=\delta(s,v)$.

Beweis.

1. Fall Wenn es keine von s aus erreichbaren negativen Kreise gibt, gilt für alle erreichbaren Knoten v: $\text{dist}[v]=\delta(s,v)$ wegen des 1. Bellman-Ford Lemma. Ist v nicht erreichbar von s, so bleibt $\text{dist}[v]$ offenbar ∞. Da nach Beendigung für alle Knoten gilt: $\text{dist}[v]=\delta(s,v) \leq \delta(s,u)+f(u,v)=\text{dist}[u]+f(u,v)$, wobei die Ungleichung immer für kürzeste Wege gilt, wird niemals `false` zurückgegeben.

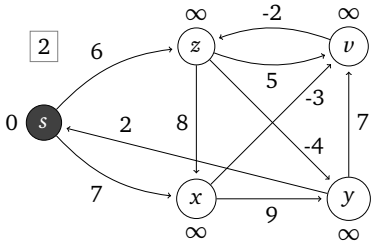
2. Fall Enthalte G nun einen negativen Kreis $c=\langle v_0, v_1, ..., v_k\rangle$ der von s aus erreichbar ist, und für den gilt: $v_0=v_k$. Es gilt dann
$\sum_{i=1}^{k} f(v_{i-1},v_i)<0$.
Nehmen wir an, Bellman-Ford gibt `true` zurück, d.h., es gilt für alle $i=1,2,...,k$: $\text{dist}[v_i] \leq \text{dist}[v_{i-1}]+f(v_{i-1},v_i)$. Dann gilt auch für die Summen:
$\sum_{i=1}^{k}\text{dist}[v_i] \leq \sum_{i=1}^{k}\text{dist}[v_{i-1}]+\sum_{i=1}^{k}f(v_{i-1},v_i)$
Da c ein Kreis ist, kommt jeder Summand in den beiden ersten Summen vor. Damit folgt:
$\sum_{i=1}^{k}\text{dist}[v_i]=\sum_{i=1}^{k}\text{dist}[v_{i-1}]$ und damit $0 \leq \sum_{i=1}^{k} f(v_{i-1},v_i)$
Dies ist aber ein Widerspruch zu $\sum_{i=1}^{k} f(v_{i-1},v_i)<0 \Rightarrow$ Bellman-Ford gibt `false` zurück

In beiden Fällen liefert der Algorithmus also das gewünschte Ergebnis. □



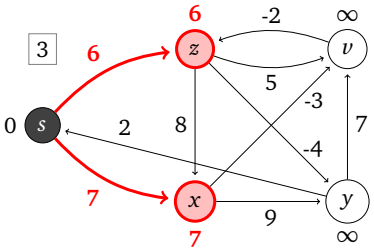
Kantenreihenfolge:

$(v,z), (x,v), (x,y), (y,v), (y,s),$
$(z,v), (z,x), (z,y), (s,x), (s,z)$



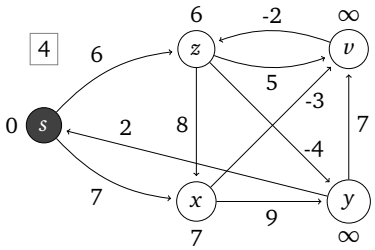
Kantenreihenfolge:

(v, z), (x, v), (x, y), (y, v), (y, s),
(z, v), (z, x), (z, y), $(s,x), (s,z)$



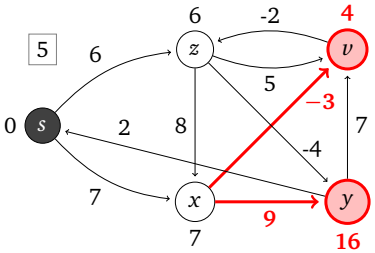
Kantenreihenfolge:

$(v,z), (x,v), (x,y), (y,v), (y,s),$
$(z,v), (z,x), (z,y),$ **(s, x), (s, z)**



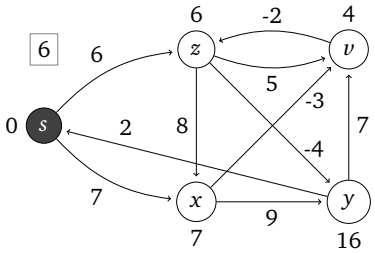
Kantenreihenfolge:

(v, z), $(x,v), (x,y), (y,v), (y,s),$
$(z,v), (z,x), (z,y), (s,x), (s,z)$



Kantenreihenfolge:

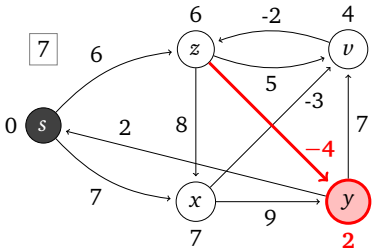
$(v,z),$ **(x, v), (x, y),** $(y,v), (y,s),$
$(z,v), (z,x), (z,y), (s,x), (s,z)$



Kantenreihenfolge:

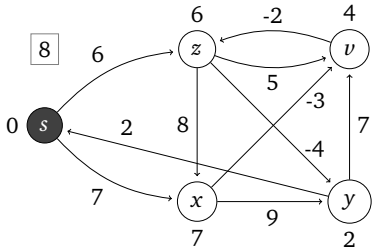
$(v,z), (x,v), (x,y), (y,v), (y,s),$
$(z,v), (z,x),$ **(y, v), (y, s),**
(z, v), (z, x), $(z,y), (s,x), (s,z)$

Abbildung 3.24: Schematischer Ablauf des Bellman-Ford-Algorithmus (Teil 1).
Im Gegensatz zum Dijkstra-Algorithmus besitzt der Bellman-Ford-Algorithmus keine Auswahlstrategie. Er durchläuft zyklisch alle Kanten und prüft, ob der Distanzwert am Zielknoten geupdatet werden muss. Nach $|V|-1$ Runden stehen die entgültigen Distanzen fest und der Algorithmus hat entweder einen kürzesten Weg oder einen negativen Kreis gefunden.



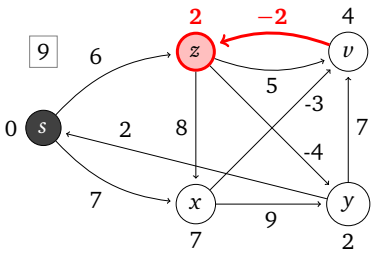
Kantenreihenfolge:

$(v,z), (x,v), (x,y), (y,v), (y,s),$
$(z,v), (z,x),$ **(z, y),** $(s,x), (s,z)$



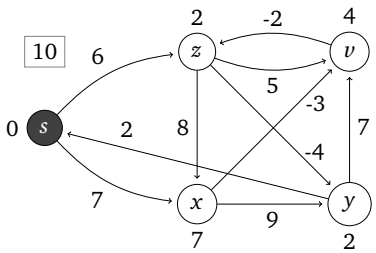
Kantenreihenfolge:

$(v,z), (x,v), (x,y), (y,v), (y,s),$
$(z,v), (z,x), (z,y),$ **(s, x), (s, z)**



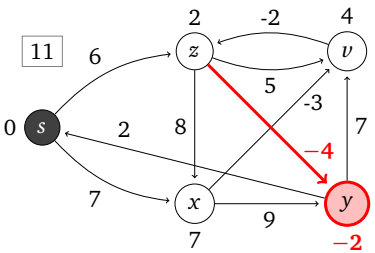
Kantenreihenfolge:

(v, z), $(x,v), (x,y), (y,v), (y,s),$
$(z,v), (z,x), (z,y), (s,x), (s,z)$



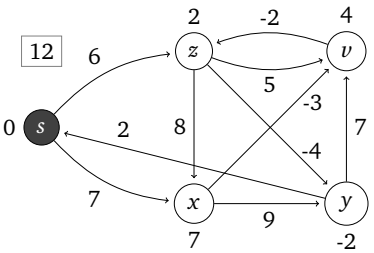
Kantenreihenfolge:

$(v,z),$ **(x, v), (x, y), (y, v), (y, s),**
(z, v), (z, x), $(z,y), (s,x), (s,z)$



Kantenreihenfolge:

$(v,z), (x,v), (x,y), (y,v), (y,s),$
$(z,v), (z,x),$ **(z, y),** $(s,x), (s,z)$



Kantenreihenfolge:

$(v,z), (x,v), (x,y), (y,v), (y,s),$
$(z,v), (z,x), (z,y),$ **(s, x), (s, z)**

Abbildung 3.25: Schematischer Ablauf des Bellman-Ford-Algorithmus (Teil 2).
(Fortsetzung der vorangehenden Seite.) Die Schritte des Algorithmus nicht einzeln dargestellt. Die Kanten wurden zu Gruppen zusammengefasst, in denen Distanzwerte geupdatet werden und in denen nichts passiert. Folglich stellen die beiden linken Spalten dieses Schemas für sich alleine betrachtet auch schon den Bellman-Ford-Algorithmus dar.

3.4 Spannbäume

In diesem Kapitel sollen nun Spannbäume behandelt werden. Dies sind spezielle Bäume, die alle Knoten des zugehörigen Graphen enthalten, aber nur so viele Kanten des Graphen, wie benötigt werden, damit der Baum zusammenhängend ist. Sie spielen beispielsweise bei der Planung von Bewässerungssystemen zur Vermeidung redundanter Rohre eine Rolle.

Definition 3.19. *Ein Gewichteter ungerichteter Graph (G, f) ist ein ungerichteter Graph $G = (V, E)$ mit Gewichtsfunktion $f : E \to \mathbb{R}$.*

Definition 3.20. *Ist $H = (U, F)$, $U \subseteq V$, $F \subseteq E$, ein Teilgraph von G, so ist das Gewicht $f(H)$ von H definiert als $w(H) = \sum_{e \in F} w(e)$.*

Definition 3.21. *Ein Teilgraph H eines ungerichteten Graphen G heißt Spannbaum von G, wenn H ein Baum ist und alle Knoten von G enthält.*

Definition 3.22. *Ein Spannbaum S eines gewichteten ungerichteten Graphen G heißt minimaler Spannbaum von G, wenn S minimales Gewicht unter allen Spannbäumen von G besitzt.*

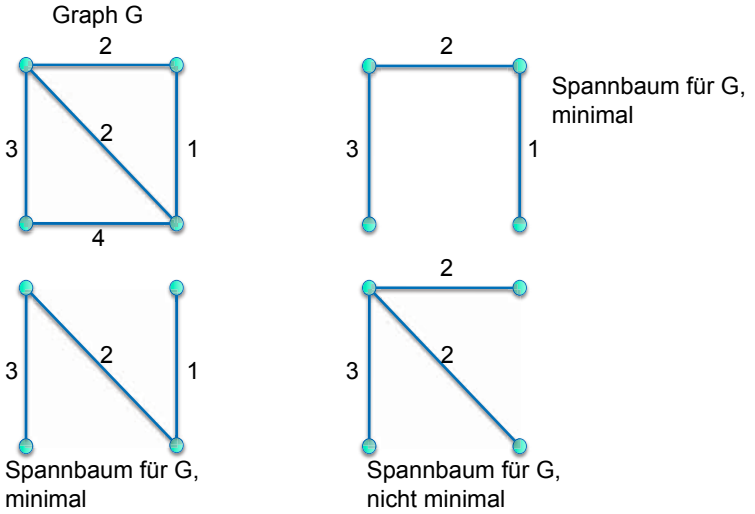


Abbildung 3.26: Schema einiger Spannbäume.
Dargestellt ist ein Graph und drei Spannbäume dieses Graphen. Zwei der Spannbäume sind minimal, da sie unter allen Spannbäumen minimales Gewicht (nämlich 6) besitzen. Es gibt zu diesem Graphen weitere Spannbäume, aber keine weiteren minimalen Spannbäume.

3.4.1 Minimal Spanning Tree (MST) Algorithmus

Es soll nun ein Algorithmus gefunden werden, der bei einem gegeben gewichteten ungerichteten Graph (G, f) mit $G = (V, E)$ effizient einen minimalen Spannbaum von (G, f) findet.
Dabei soll so vorgegangen werden, dass iterativ eine Kantenmenge $A \subseteq E$ zu einem minimalen Spannbaum erweitert wird.

Definition 3.23. *Eine Kante (u, v) heißt A-sicher, wenn $A \cup (u, v)$ zu einem minimalen Spannbaum erweitert werden kann.*

Der Grundaufbau des Algorithmus ist dann wie folgt:

- Zu Beginn: $A = \emptyset$
- Ersetze in jedem Schritt A durch $A \cup (u, v)$, wobei (u, v) eine A-sichere Kante ist.
- Wiederhole dies so lange, bis $|A| = |V| - 1$

Als generischer „Minimal Spanning Tree"-Algorithmus (kurz MST) ergibt sich:

Listing 3.8: Generischer MST-Algorithmus

```
1  Generic-MST (G,f):
2      A := empty set
3      while A is no spanning tree
4          for an arbitrary A-safe edge (u,v)
5              A := A union {(u,v)}
6      return A
```

Wobei in dem Algorithmus das Ersetzen solange wiederholt wird, bis A ein Spannbaum ist. Dies entspricht aber $|A| = |V| - 1$, da nur A-sichere Kanten eingefügt wurden und die $(|V| - 1)$-te Kante somit den Spannbaum fertigstellt.

Definition 3.24. *Ein Schnitt $(C, V \setminus C)$ in einem Graphen $G = (V, E)$ ist eine Partition der Knotenmenge V des Graphen G in die beiden Mengen C und $V \setminus C$.*

Definition 3.25. *Eine Kante von G kreuzt einen Schnitt $(C, V \setminus C)$, wenn ein Knoten der Kante in C, der andere in $V \setminus C$ liegt.*

Definition 3.26. *Für einen Schnitt $(C, V \setminus C)$ ist eine Teilmenge $A \subseteq E$ verträglich, wenn kein Element von A den Schnitt kreuzt.*

Definition 3.27. *Eine den Schnitt $(C, V \setminus C)$ kreuzende Kante heißt leicht, wenn sie eine Kante minimalen Gewichts unter den $(C, V \setminus C)$ kreuzenden Kanten ist.*

Satz 3.17 (Erster Satz über Minimale Spannbäume). *Sei (G, f) ein gewichteter, ungerichteter Graph. Es gebe einen minimalen Spannbaum T in G, der die Kantenmenge $A \subseteq E$ enthalte. Sei $(S, V \setminus S)$ ein mit A verträglicher Schnitt*

[...verträglich: kein Element von A kreuzt den Schnitt]

und (u, v) sei eine leichte $(S, V \setminus S)$ kreuzende Kante.

[... leicht: Kante minimalen Gewichts unter den $(C, V \setminus C)$ kreuzenden Kanten]

Dann ist (u, v) eine A-sichere Kante.

Satz 3.18 (Zweiter Satz über Minimale Spannbäume). *Sei (G, f) ein gewichteter, ungerichteter Graph. Es gebe einen minimalen Spannbaum in G, der die Kantenmenge $A \subseteq E$ enthalte. Ist (u, v) eine leichte Kante minimalen Gewichts, die einen Baum B des Waldes $G_A = (V, A)$ mit einem anderen Baum von G_A verbindet, so ist (u, v) A-sicher.*

Beweis.

- Der Schnitt $(B, V \setminus B)$ ist A-verträglich. Denn:
  - B ist eine Zusammenhangskomponente von $G_A = (V, A)$
  - also ist B ein Baum, der keine Kanten in G_A zu $V \setminus B$ aufweist
  - also A-verträglich (folgt direkt aus der Definition)
- (u, v) ist deshalb eine leichte Kante für den Schnitt. Denn:
  - (u, v) verbindet zwei Bäume, die Teil eines minimalen Spannbaums sind.
  - Billiger können die beiden Teilbäume nicht verbunden werden.
- Also folgt (u, v) A-sicher mit dem 1. Satz über Minimale Spannbäume.

□

3.4.2 Algorithmus von Prim

Ein weitere Algorithmus der sich mit Minimalen Spannbäumen beschäftigt ist der Algorithmus von Prim. Die Idee dabei ist:

- Zu jedem Zeitpunkt des Algorithmus besteht der Graph $G_A = (V, A)$ aus einem Baum T_A und einer Menge von isolierten Knoten I_A
- Eine Kante minimalen Gewichts, die einen Knoten aus I_A mit T_A verbindet, wird zu A hinzugefügt und somit der Spannbaum minimal weiter aufgebaut.
- Die Knoten in I_A sind in einem Min-Heap organisiert. Dabei ist der Schlüssel `key[v]` eines Knotens $v \in I_A$ gegeben durch das minimale Gewicht einer Kante, die v mit T_A verbindet.

Listing 3.9: Algorithmus von Prim

```
1   Prim-MST (G,f,w):
2       for all v in V do
3           key[v] := infinity
4           pred[v] := nil
5       key[w] := nil
6       Q := Build-Heap (V)
7       while Q != empty set do
8           u := Extract-Min (Q)
9           for all v in adj(u) do
10              if v in Q and f(u,v) < key[v]
11                  pred[v] := u
12                  key[v] := f(u,v)
13                  Decrease-Key (Q,v,key[v])
```

Der Algorithmus bekommt als Eingabe den zu bearbeitenden Graphen G, die passende Gewichtsfunktion f und den Startknoten W. Dann bekommt der Startknoten den `key` 0 zugewiesen alle anderen Knoten den `key` ∞, außerdem wird für alle Knoten festgelegt, dass sie noch keinen Vorgänger π haben. Aus diesen Knoten wird nun ein Heap erzeugt.
In der `while`-Schleife wird nun solange der Heap nicht leer ist, der Knoten u mit minimalem Schlüssel aus dem Heap entfernt. Für alle seine Nachfolger, die sich noch im Heap befinden und deren `key` größer ist als das Kantengewicht der Kante zu u, wird der `key` auf dieses Kantengewicht gesetzt und u als Vorgänger gespeichert. Die Veränderung des `key` wird natürlich auch im Heap gespeichert.

Für die *Laufzeit des Algorithmus von Prim* ergibt sich insgesamt $O(|E| \cdot \log |V|)$, dies ergibt sich aus den einzelnen Teilen:

- $|V|$-viele `while`-Durchläufe
- Zeile 7: $O(\log |V|)$
- Zeilen 9-12: $O(2 \cdot |E|)$ Durchläufe
- Zeile 12: $O(\log |V|)$

Verwendet man spezielle Fibonacci-Heaps ist eine Gesamtlaufzeit in $O(|V| \log |V| + |E|)$ möglich.

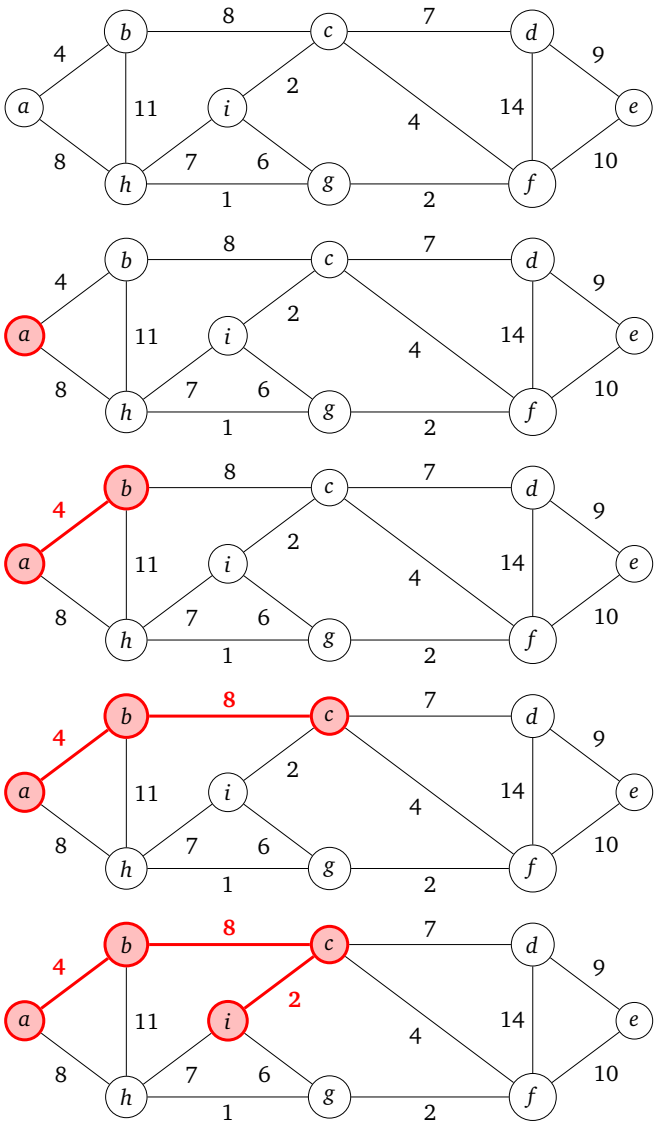


Abbildung 3.27: Schematischer Ablauf des Algorithmus von Prim (Teil 1).
Gegeben sei ein ungerichteter Graph. Die Knoten und Kanten, die zum minimalen Spannbaum gehören, werden rot markiert. Als nächste Kante für den minimalen Spannbaum wird in jedem Schritt diejenige Kante mit dem kleinsten Gewicht gewählt, die zu keinem Knoten des Spannbaumes führt.

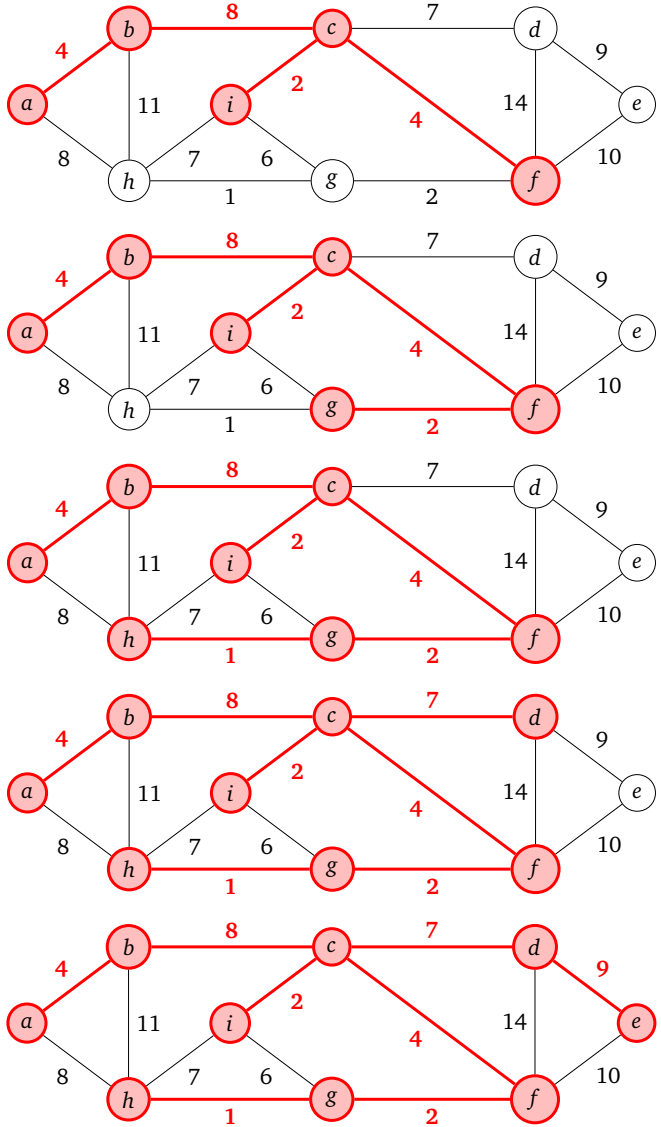


Abbildung 3.28: Schematischer Ablauf des Algorithmus von Prim (Teil 2).
(Fortsetzung der vorangehenden Seite.) Wie zu sehen ist, können dabei auch mehrere Kanten des gleichen Ausgangsknotens gewählt werden, wenn die Kanten entsprechende Gewichte besitzen. Dass der Algorithmus mit einem Heap implementiert ist, geht in die graphische Darstellung nur mittelbar über die Kantenauswahlregel ein.

3.5 Maximale Flüsse

In diesem Abschnitt werden so genannte Flussprobleme behandelt. Dabei ist es meist Ziel eines Flussproblems für einen gewichteten Graphen einen maximalen Fluss vom Start- zum Endknoten, bzw. von Quelle zu Senke zu finden. Flussprobleme finden häufig Anwendung bei der Modellierung von Verteilungsproblemen, Transport- und Umladeproblemen. Zu transportierende Ware sind z.B. Wasser, Strom, Gas, Fahrzeuge.

Zur Einführung soll folgendes Transportproblem als Beispiel dienen:

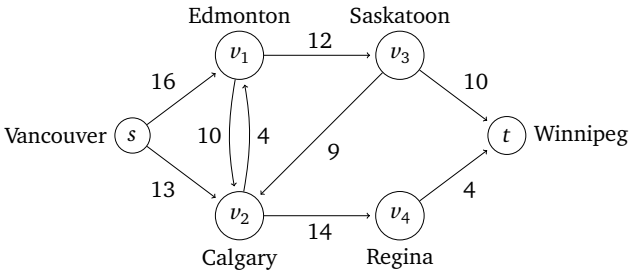


Abbildung 3.29: Beispiel eines Flussproblems:
Die Fabrik in Vancouver ist die Quelle s, der Grossmarkt in Winnipeg die Senke t. Die Güter müssen über Straßen zum Zielort gebracht werden. Leider können nur $c_{u,v}$ viele Gütereinheiten pro Woche über (u, v) transportiert werden. Wie kriegt man also jede Woche so viele Einheiten des zu verkaufenden Guts über alle möglichen Wege von Vancouver nach Winnipeg?

Definition 3.28. *Eine Flussnetzwerk hat folgenden Aufbau:*

- $G = (V, E)$ *ist ein gerichteter Graph,*
- *jedes* $(u, v) \in E$ *besitzt eine nicht-negative Kapazitätsbeschränkungen* $c(u, v) \geq 0$
- *falls* $(u, v) \notin E$, *gilt* $c(u, v) = 0$
- *es gibt zwei ausgezeichnete Knoten: Quelle* s *und Senke* t
- *es gibt für jeden Knoten* v *einen Pfad von* s *nach* v *und einen von* v *nach* t

Definition 3.29. *Sei* $G = (V, E)$ *ein Flussnetzwerk, sei* s *Quelle und* t *Senke. Ein Fluss in* G *ist eine Funktion* $f : V \times V \to \mathbb{R}$ *mit:*

- *Kapazitätsbeschränkung:* $f(u, v) \leq c(u, v)$ *für alle* $u, v \in V$
- *Symmetrie:* $f(u, v) = -f(v, u)$ *für alle* $u, v \in V$
- *Flusskonservierung:* $\sum_{v \in V} f(u, v) = 0$ *für alle* $u \in V$

Definition 3.30. *Der Wert eines Flusses ist* $|f| = \sum_{v \in V} f(s, v)$*, der Gesamtfluss aus* s *heraus*

3.5.1 Ford Fulkerson Algorithmus

Der Ford-Fulkerson Algorithmus kann für ein Flussproblem einen maximalen Fluss finden. Dafür wird mit einem Fluss angefangen und immer geschaut, ob es für diesen Fluss einen verbessernden Pfad gibt. Solange dies zutrifft wird der Fluss entlang dieses Pfades verbessert.

Definition 3.31. *Gegeben sei ein Flussnetzwerk und ein zulässiger Fluss* x *von* s *nach* t*. Ein „augmenting path" (oder „verbessernder Pfad") ist ein Pfad* P *von* s *nach* t *bei dem die Kantenrichtungen ignoriert werden mit folgenden Eigenschaften:*

- *Für jede Kante* (a, b)*, die von* P *in Vorwärtsrichtung durchschritten (Vorwärtskante) wird gilt:* $f(a, b) < c(a, b)$*. D.h., Vorwärtskanten haben freie Kapazitäten.*
- *Für jede Kante* (b, a)*, die von* P *in Rückwärtsrichtung durchschritten wird (Rückwärtskante) gilt:* $f(a, b) > 0$*.*

Für die maximale Änderung entlang P *ergibt sich dabei:*

$$min_{Kanten\ von\ P} \begin{cases} c(a,b) - f(a,b) & \text{entlang Vorwärtskanten} \\ f(a,b) & \text{entlang Rückwärtskanten} \end{cases}$$

Listing 3.10: Ford-Fulkerson Algorithmus

```
1    Ford-Fulkerson (G,s,t):
2        Initialize flow with 0
3        while exists(augmenting path p)
4            Improve flow f along p
5        return f
```

Definition 3.32. *Sei* f *ein Fluss in* G*.* $c_f(u, v) = c(u, v) - f(u, v)$ *nennt man residuale Kapazität.*

Definition 3.33. *Sei* $G = (V, E)$ *ein Flussnetzwerk und* f *ein Fluss. Das residuale Netzwerk ist dann* $G_f = (V, E_f)$ *mit* $E_f = \{(u, v) \in V \times V | c_f(u, v) > 0\}$*. Es können im residualen Graphen mehr Kanten sein, als im Originalen. Denn im residualen Netzwerk sind an allen Kanten des Orginals zwei Kanten (in beide Richtungen), außer es gilt* $f(u, v) = 0$ *oder* $f(u, v) = c(u, v)$*.*

Definition 3.34. *Ein Schnitt eines Flußnetzwerkes ist eine Aufteilung der Knoten* V *in die Mengen* S *und* $T = V \setminus S$*, so dass* $s \in S$ *und* $t \in T$*.*

Definition 3.35. *Wenn* f *ein Fluss ist, dann ist ein Netzfluss über einen Schnitt* (S, T) *definiert als*
$$\sum_{\substack{(a,b) \in E \\ a \in S,\, b \in T}} f(a, b)$$

Definition 3.36. *Die Kapazität eines Schnittes* (S, T) *ist:* $C(S, T) := \sum_{\substack{(a,b) \in E \\ a \in S,\, b \in T}} c(a, b)$

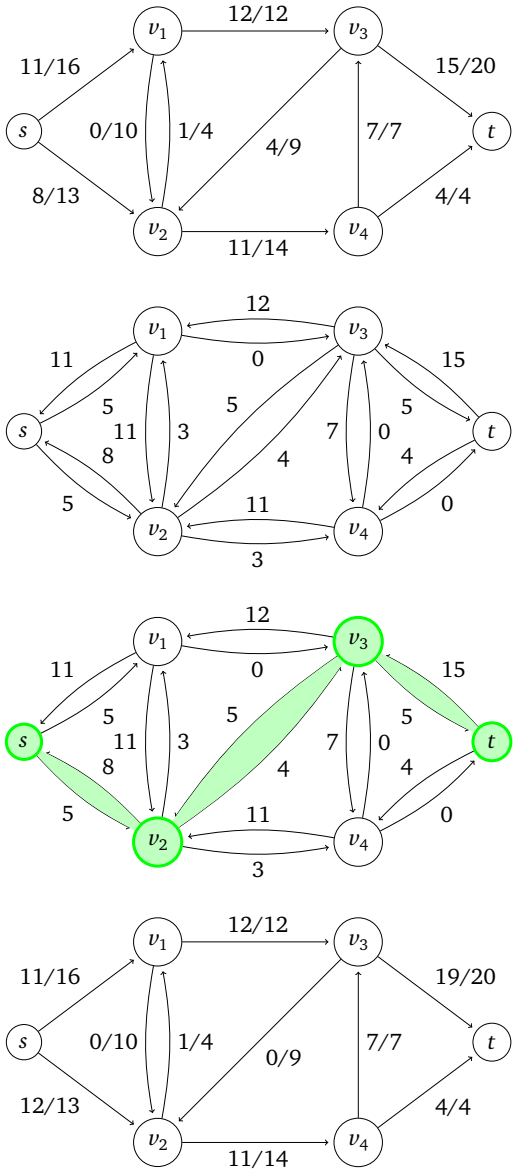


Abbildung 3.30: Beispiel eines residualen Netzwerks.
Im ersten Bild sind jeweils Fluss-Wert und Kapazität der Kanten des Graphen angegeben. Im zweiten Bild ist das Residuale Netzwerk abgebildet. Das dritte Bild zeigt einen verbessernden Pfad und im vierten sind Fluss-Wert des verbesserten Flusses und Kapazität der Kanten angegeben.

Satz 3.19 (Erster Satz über residuale Netzwerke)**.** *Sei $G = (V, E)$ ein Flussnetzwerk und sei f ein Fluss. Sei G' das residuale Netzwerk von G, und sei f' ein Fluss in G' entlang eines verbessernden Pfades. Dann gilt für die Summe der Flüsse $f + f'$: $|f + f'| = |f| + |f'|$. Die Erhöhung in G entspricht also dem Pfad aus G'.*

Beweis. Die Aussage folgt direkt aus der Konstruktion von G'. □

Satz 3.20 (Zweiter Satz über residuale Netzwerke)**.** *Wenn (S, T) ein Schnitt ist, kann der Fluss von S nach T nicht größer sein, als die Kapazität des Schnitts.*

Beweis. Für jede einzelne Kante (u, v) von S nach T gilt, dass $f(u, v) \leq c(u, v)$. Also gilt auch für die Summe über alle Kanten von S nach T: $\sum_{u \in S, v \in T} f(u, v) \leq \sum_{u \in S, v \in T} c(u, v)$ □

Satz 3.21 (Max-Flow Min-Cut Theorem)**.** *Sei f ein Fluss in einem Flussnetzwerk $G = (V, E)$ mit Quelle s und Senke t. Dann sind folgende Aussagen äquivalent:*

1. *f ist ein maximaler Fluss.*
2. *Das residuale Netzwerk G_f enthält keinen verbessernden Pfad.*
3. *Es gibt einen Schnitt (S, T) so, dass $|f| = \sum_{u \in S, v \in T} c(u, v)$*

Beweis.

1 ⇒ 2: Annahme, f sei ein maximaler Fluss und G_f enthält einen verbessernden Pfad f'. Der verbessernde Pfad ist aber gerade so gewählt, dass er hilft, den Fluss f zu vergrößern. Damit wäre also $|f + f'| > |f|$. Dann wäre aber f nicht maximal gewesen. ⇒ Widerspruch! ⇒ Annahme war falsch ⇒ 2.

2 ⇒ 3: Annahme: Es gebe keinen verbessernden Pfad. Dann gibt es keinen Weg in G_f von s nach t, bei dem die Kantenkapazitäten > 0 sind. Sei nun $S = \{v \in V$ mit: es gibt einen Pfad von s zu v in $G_f\}$. Dann ist $(S, T = V \setminus S)$ eine Partition und für jede Kante (u, v) mit $u \in S$ und $v \in T$ gilt $f(u, v) = c(u, v)$, da sonst $(u, v) \in E_f$.

3 ⇒ 1: Es sei $|f| = \sum_{u \in S, v \in T} c(u, v)$, für S und T wie in Punkt 2. Aus dem zweiten Satz über residuale Netzwerke folgt, dass es keinen größeren Fluss gibt.

□

Verbessernde Pfade kann man mittels Breitensuche finden. Dabei muss dann nach einem Weg gesucht werden.

Satz 3.22 (Erster Satz des Ford-Fulkerson Algorithmus)**.** *Wenn der Ford-Fulkerson Algorithmus terminiert, terminiert er mit optimaler Lösung.*

Beweis. Bilde nach der Terminierung die Mengen S und T wie im Max-flow min-cut Theorem. Alle Vorwärtskanten sind dann saturiert (d.h. der Fluss über sie ist maximal), alle Rückwärtskanten leer. (Sonst hätte der Algorithmus nicht terminiert) Der (S, T)-Schnitt hat den gleichen Wert, wie der Fluss, den der Algorithmus liefert. □

Satz 3.23 (Zweiter Satz des Ford-Fulkerson Algorithmus)**.** *Der Ford-Fulkerson Algorithmus terminiert nach endlich vielen Schritten, sofern alle Inputparameter natürliche oder rationale Zahlen sind.*

Beweis. natürliche Zahlen: klar, da der Fluss immer um ganzzahlige Einheiten erhöht wird.
rationale Zahlen: klar, da man vor Beginn der Berechnungen mit gemeinsamen Nenner multiplizieren kann. □

Unter bestimmten Bednigungen kann man für den Ford-Fulkerson-Algorithmus eine polynomielle Laufzeit nachweisen.

Weitere Fluss-Probleme

Neben dem behandelten Maximalen-Fluss-Problems werden nun noch zwei verwandte Probleme erläutert.

Das erste Problem unterscheidet sich lediglich durch mehrere Quellen und/oder Senken von dem bereits bekanntem.

Das zweite Problem ist das Maximum Matching in bipartiten Graphen. Dabei handelt es sich um ein Paarungsproblem. Ziel ist es einen Teilgraphen zu finden, der möglichst viele Knoten bzw. Kanten beinhaltet, wobei von einem Knoten immer nur ein Kante abgehen oder ankommen darf. Ein Teilgraph mit dieser Eigenschaft heißt Matching, die Anzahl der beinhalteten Kanten definiert die Größe des Matching und ein Maximum-Matching ist dementsprechen das größte Matching eines Graphen.

Satz 3.24. *Sei $G = (V, E)$ ein bipartiter Graph mit Knotenpartitionierung $V = L \cup R$. Sei $G' = (V', E')$ das zugehörige Flussnetzwerk. Dann gilt: Wenn M ein Matching in G ist, dann gibt es einen ganzzahligen Fluss in G' mit $|f| = |M|$. Wenn andersherum f ein ganzzahliger Fluss in G' ist, dann gibt es ein Matching M in G mit $|f| = |M|$.*

Beweis. Übungsaufgabe □

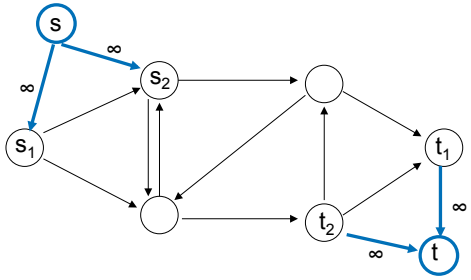


Abbildung 3.31: Beispiel eines Maximalen-Fluss-Problems mit mehreren Quellen und Senken.

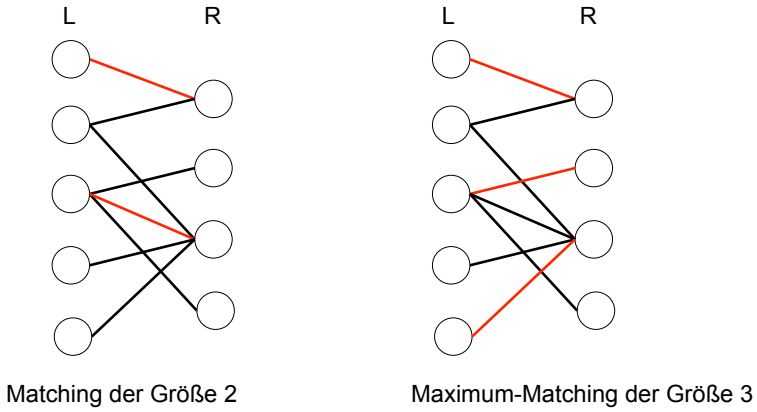


Abbildung 3.32: Darstellung eines Maximalen Matchings.
Abgebildet ist der gesamte Graph, das Matching besteht jeweils nur aus den roten Kanten.

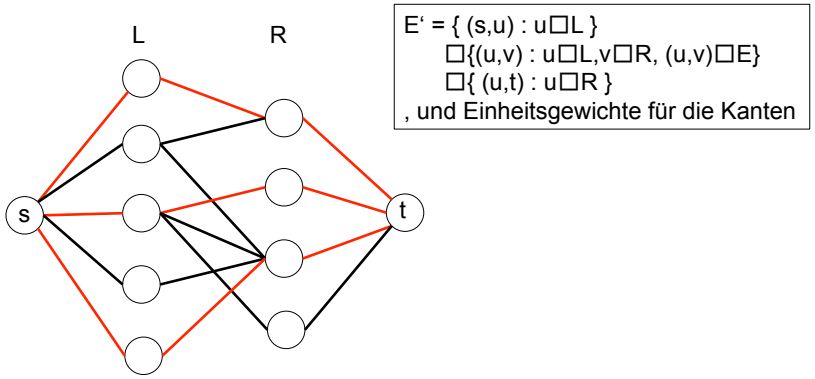


Abbildung 3.33: Schema eines Maximalen-Matching-Problems bipartiter Graphen.
An jedem der Knoten (außer s und t) ist entweder keine Kante im Matching oder es geht genau eine Kante (rot) ab und eine kommt an. Durch die Einführung der zusätzlichen Knoten s und t transformiert man das Matching-Problem in ein Flussproblem.

3.6 Traveling Salesman

Das Travelling Salesman Problem mit Dreiecksungleichung:
Gegeben ist ein vollständiger Graph $G = (V, E)$ mit Kostenfunktion $c : V \times V \to \mathbb{Z}, k \in \mathbb{Z}$.
Zusätzlich gilt die Dreiecksungleichung: $c(u, w) \leq c(u, v) + c(v, w)$
Die Frage ist, ob es eine Rundtour gibt, die jeden Knoten genau einmal besucht mit Kosten höchstens k.
Dieses Problem ist NP-vollständig.
Zur Lösung betrachte folgenden Algorithmus:

1. wähle einen Knoten r als Startknoten
2. erzeuge einen minimalen Spannbaum mit Hilfe des Algorithmus von Prim.
3. durchlaufe mittels Tiefensuche den minimalen Spannbaum

Beobachtung: Dieser Algorithmus läuft durch die Tiefensuche eine Tour, deren Kosten höchstens so groß sind, wie die das Doppelte der Kosten einer minimalen Rundreise. Denn es wird keine Kante des minimalen Spannbaums mehr als zweimal durchlaufen und die Kosten der minimalen Rundreise müssen größer sein als die des minimalne Spannbaums. Leider ist unsere bisherige Tour noch keine Rundreise. Das ändern wir wie folgt: Wenn ein Knoten auf unserer Tour bereits besucht wurde, besuche diesen nicht nochmal, sondern gehe zum nächsten auf der Tour liegenden Knoten direkt hin. Der direkte Weg ist eine Abkürzung, weil nach der Vorraussetzung die Dreiecksungleichung gilt. Die neue Tour ist eine Rundtour, und ihre Kosten sind nicht größer als die Kosten der Tour, die zweimal den minimalen Spannbaum abläuft.
Fazit: Wir haben einen Näherungsalgorithmus mit Güte 2 gefunden.

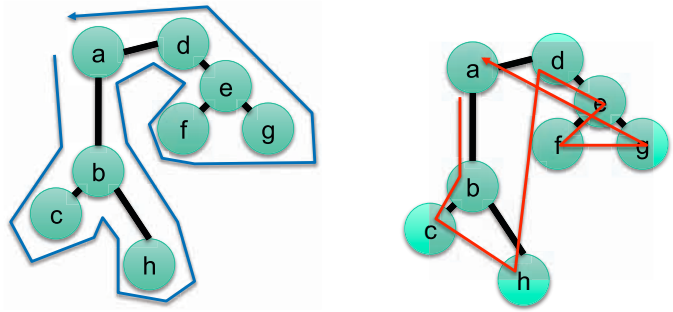


Abbildung 3.34: Beispiel zum Traveling-Salesman-Problem.
Der minimale Spannbaum ist schwarz eingezeichnet, im linken Bild ist zusätzlich der Weg der Tiefensuche (blau) dargestellt und rechts die Rundtour(rot).

Vermutlich gibt es keinen effizienten Algorithmus für das TSP - Problem mit Dreiecksungleichung. Aber, wir können verschiedene Möglichkeiten, Kanten zu Kreisen zu fügen ausprobieren und bewerten. Dies macht der Branch and Bound - Algorithmus, der nun an einem Beispiel gezeigt wird.

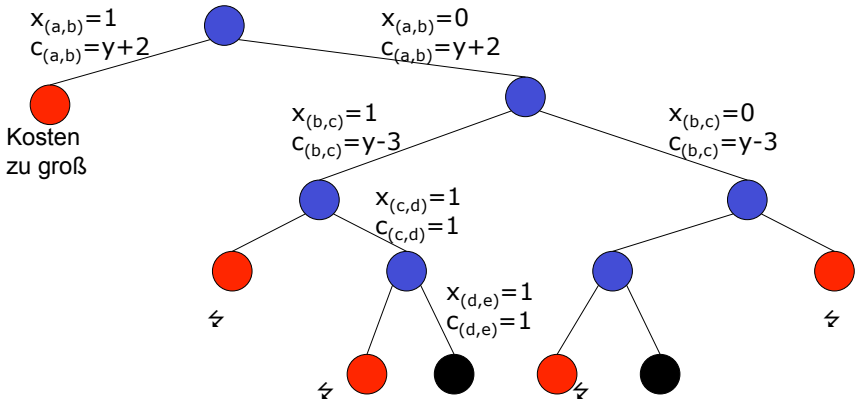


Abbildung 3.35: Schematischer Ablauf des Traveling-Salesman-Algorithmus.
Bekannt sei, dass es eine Tour mit Kosten y gibt. $x(a, b) = 1$ bedeute, Kante (a, b) wird hinzugenommen, $c(a, b)$ seien die Kosten der Kante (a, b). Mit Hilfe des Branch and Bound - Algorithmus lassen sich viele Möglichkeiten ausschließen, sodass fast nie wirklich alle möglichen Touren ausprobiert und bewertet werden müssen. Im worst-case ist die Laufzeit des Branch and Bound - Algorithmus dennoch exponentiell.

Kapitel 4

Sortieren in Arrays

Eine zentrale Aufgabe von Algorithmen ist das Sortieren von beliebigen Elementen, die eine Ordnungsrelation zueinander haben. Wir werden im Folgenden „nur“ Zahlenfolgen sortieren. Diese Einschränkung ist aber willkürlich und macht für die Algorithmik keinen Unterschied. Allgemeiner könnten wir davon ausgehen, dass eine Menge von Objekten vorliegt, für die eine Ordnungdrelation definiert ist. Das kann ebenfalls ein Zahlenschlüssel sein, kann aber auch z.B. eine lexikographische Ordnung wie beim Sortieren von Namen sein.

Eingabe Folge von Zahlen $\langle a_1, \ldots, a_n \rangle$

Gesucht Permutation $\langle a'_1, \ldots, a'_n \rangle$, so dass $a'_1 \leq \cdots \leq a'_n$

Typischerweise werden die Daten in Arrays gespeichert. Ein Array ist eine Datenstruktur, in der Daten an nummerierten Plätzen gespeichert werden können.

In diesem Kapitel werden mehrere Sortieralgorithmen vorgestellt.

4.1 Insertion Sort

Insertion-Sort beginnt zunächst die erste zwei Elemente zu sortieren und nimmt dann in jedem Durchlauf ein weiteres Element hinzu und ordnet dieses in den bereits sortierten Teil ein. Um das Einordnen im Array korrekt auszuführen, wird das einzusortierende Element zwischengespeichert. Dann werden alle Elemente des sortierten Teils, die größer sind als dieses Element um einen Platz verschoben und zum Schluss wird das zwischengespeicherte nach dem ersten kleineren Element im Array gespeichert.

Listing 4.1: Insertion-Sort Algorithmus

```
1    Insertion-Sort (A):
2        for j := 2 to length (A) do
3            key := A[j]
4            i := j - 1
5            while i > 0 and A[i] > key
6                A[i+1] := A[i]
7                i := i - 1
8            A[i+1] := key
```

Der Aufwand von Insertion-Sort liegt in $O(n^2)$, denn die `for`-Schleife wird n-mal durchlaufen, dabei wird jedes mal die `while`-Schleife bis zu n-mal ausgeführt.

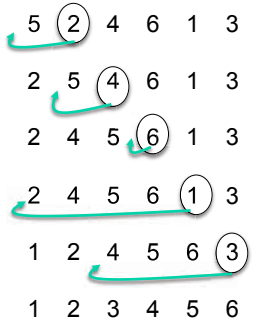


Abbildung 4.1: Schematischer Ablauf des Insertion-Sort-Algorithmus.
In jedem Schritt (bzw. in jeder Zeile) wird die eingekreiste Zahl mit dem vor ihr liegendem Teil des Array verglichen und an der richtigen (mit dem Pfeil gekennzeichneten) Stelle eingefügt. Im nächsten Schritt wird dann das nächste Element im Array betrachtet (und eingekreist).

4.2 Merge Sort

Seien A_1 und A_2 zwei sortierte Arrays der Längen m und n. Diese sollen zu einem neuen sortierten Array A der Länge $m+n$ verschmolzen werden. Dazu durchläuft die Merge-Operation A_1 und A_2 von „links“ nach „rechts“ mit zwei Indexzeigern i und j und führt eine Art „Reißverschlußverfahren“ aus. Dabei wird immer der kleinere der angezeigten Werte aus A_1 und A_2 in A hinzugefügt und der entsprechende Zeiger um eins nach rechts verschoben.

Listing 4.2: Merge-Algorithmus (Ansatz)

```
1    Merge(A1,A2):
2        i := 1
3        j := 1
4        k := 1
5        while i <= m and j <= n
6            if A1[i] < A2[j]
7                A[k] := A1[i]
8            else
9                A[k] := A2[j]
10           k := k + 1
11           i := i + 1
12       if i == m+1 and j <= n
13           Haenge die restlichen Elemente von A2 an A
14       if j == n+1 and i <= m
15           Haenge die restlichen Elemente von A1 an A
```

Der Aufwand der Merge-Operation liegt in $O(n+m)$.

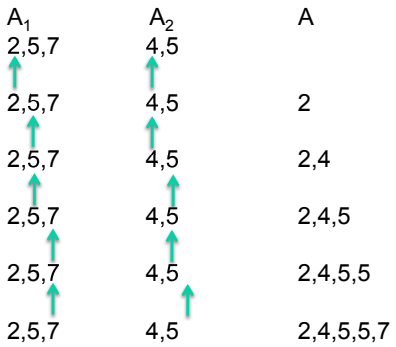


Abbildung 4.2: Idee der Merge-Operation.
A_1 und A_2 sind die beiden Arrays, die zu einem Array A zusammengefügt werden sollen. Die beiden Pfeile zeigen auf die zu vergleichenden Elemente, das kleinere von ihnen wird zu A hinzugefügt und der Pfeil in dem entsprechenden Array um eine Position nach rechts verschoben.

Der Merge-Sort Algorithmus (siehe Listing 4.3) ruft sich rekursiv immer weiter für die linke und rechte Hälfte des eingegebenen Array auf. Danach wird die Merge-Operation auf dem entstandenen Array aufgerufen. Da die Merge-Operation immer erst aufgerufen wird, wenn der Merge-Algorithmus schon für beide Hälften durchgelaufen ist, wird die Merge-Operation immer für zwei sortierte Arrays aufgerufen und fügt diese sortiert zusammen.

Listing 4.3: Merge-Sort Algorithmus

```
1    Merge-Sort (A,l,r):
2        if l < r
3            m := floor ( (l+r)/2 )
4            Merge-Sort (A,l,m)
5            Merge-Sort (A,m+1,r)
6            Merge (A,l,m,r)
```

Die Korrektheit kann durch die Induktion über die Anzahl der zu sortierenden Komponenten gezeigt werden.

Die Laufzeit $O(n \log n)$ des Merge-Sort Algorithmus bei einer Länge n der zu sortierenden Zahlenfolge ergibt sich mit Hilfe des Mastertheorems:

$$T(n) = \begin{cases} \Theta(1) & \text{falls n = 1} \\ 2T(\frac{n}{2}) & \text{falls n > 1} \end{cases}$$

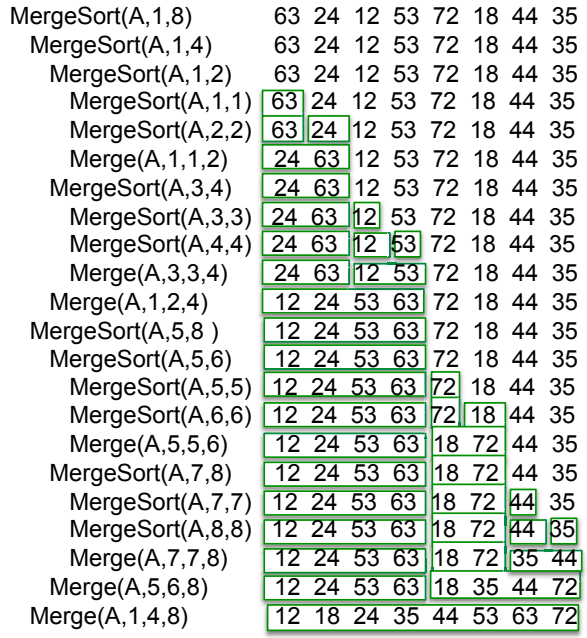


Abbildung 4.3: Schematische Ablauf des Merge-Sort-Algorithmus.
In der linken Spalte sind die einzelnen Aufrufe von MergeSort aufgeführt in der rechten das Array zu dem entsprechenden Zeitpunkt. Die grün markierten Teile des Arrays wurden bereits mit der Merge-Operation sortiert.

Für eine Eingabe von mehr als 600 Elementen eignet sich erfahrungsgemäß ein anderer Algorithmus besonders gut: Der Heapsort-Algoithmus beruht auf Vergleichen und Vertauschen von Elementenpaaren.

4.3 Heap Sort

Der Heap-Algorithmus baut aus den Elementen einen Heap. Dann tauscht es immer das erste Element mit dem letzten, verringert die Heapgröße um eins und stellt danach mit Heapify die Heapeigenschaft wieder her. Somit wird immer das kleinste Element des Heap hinten angehängt und auch nicht mehr verändert, da der Heap verkürzt wird.

Listing 4.4: Heap-Sort Algorithmus

```
1    Heap-Sort:
2        Build-Heap
3        for i := 1 to n do
4            Tausche erstes und letztes Element im Heaparray
5            size := size - 1
6            Heapify
```

Auch die Laufzeit von Heapsort liegt in $O(n \log n)$.

4.4 Quick Sort

Bei weniger Elementen gibt es einen weitere Algorithmus, der meist schneller als Heapsort ist. Es handelt sich um den auf einer variablen Aufteilung des Eingabearrays basierenden Quicksort-Algorithmus. Er wurde 1962 von C.A.R. Hoare entwickelt. Obwohl sich Quicksort bei einer geringen Eingabegröße bewährt hat, werden im Worst Case $\Omega(n^2)$ Vergleiche (Hauptoperation des Algorithmus) benötigt. Im Mittel ergeben sich aber nur $O(n \log n)$ Vergleiche.

Grob lässt sich der Quicksort-Algorithmus folgendermaßen darstellen:

1. Gegeben sei ein Array A mit n Komponenten
2. Wähle ein beliebiges Pivotelement $A[\texttt{pivot}]$
3. Zerlege A in zwei Teilbereiche $A[0],...A[k-1]$ und $A[k+1],...A[n]$, so dass
   - a) $A[i] < A[\texttt{pivot}]$ für alle i=0,...,k-1
   - b) $A[i] = A[\texttt{pivot}]$
   - c) $A[i] \geq A[\texttt{pivot}]$ für alle i=k+1,...,n
4. Sofern ein Teilbereich aus mehr als einer Komponente besteht, so wende Quicksort rekursiv auf ihn an. (also evtl. auch für beide Teilbereiche)

Die Korrektheit ergibt sich leicht: Durch die Zerlegung in den Zeilen 3a-3c befinden sich alle Elemente, die kleiner als $A[\texttt{pivot}]$ sind in einem Teilarray und alle die größer sind im anderen. Wenn beide Teilbereiche sortiert sind, ist also auch das Gesamtarray sortiert. Der Rest folgt induktiv.

Listing 4.5: Schritt 3: Zerlege A

```
1    pivot := arbitrary i in 1..n
2    l := 1
3    r := n
4    while l < r
5        Find smallest l' >= l mit A[l'] >= A[pivot]
6        Find largest r' <= r mit A[r'] < A[pivot] or r' ==
           pivot
7        if l' == r'
8            STOP
9        else
10           Swap A[l'] und A[r']
11       S := pivot;
12       if S == l'
13           pivot := r'
14           r := r'
15       else
16           r := r' - 1
17       if S == r'
18           pivot := l'
19           l := l'
20       else
21           l := l' + 1
```

Der Ablauf des Zerlegungsschritts wird am Beispiel klar:

Tabelle 4.1: Beispiel zum Quicksort-Algorithmus ($n = 13$, $\texttt{pivot} = 7$).

	Array												
Schritt	1	2	3	4	5	6	7	8	9	10	11	12	13
1	15	47	33	87	98	17	**53**	76	83	2	53	27	44
2	15	47	33	44	98	17	**53**	76	83	2	53	27	87
3	15	47	33	44	27	17	**53**	76	83	2	53	98	87
4	15	47	33	44	27	17	2	76	83	**53**	53	98	87
5	15	47	33	44	27	17	2	**53**	83	76	53	98	87

Als Pivotelement wurde $A[7]$ (also 53) gewählt. Von rechts und links wird nun A durchlaufen und die Elemente mit 53 verglichen. Wird ein Element gefunden, dass noch auf der falschen Seite ist, so wird es mit einem gefundenem falsch platzierten Element auf der anderen Seite getauscht. Im ersten Schritt werden so 87 und 44 getauscht.

Im Worst-Case wird das Array der Länge n in zwei Teilbereiche der Länge 1 und $n-1$ aufgeteilt. Die Anzahl der Rekursionsaufrufe ist dann gleich n. Da in jeder Rekursionsebene für einen Teilbereich der Länge r gerade $r-1$ Vergleiche statt finden, erhält man für die Anzahl der Vergleiche:

$$V(n) = (n-1)+(n-2)+\cdots+2+1 = \frac{n(n-1)}{2} = \Theta(n^2)$$

Dies ist sehr schlecht. Jedoch wird man bei der Analyse dem Quicksort-Algorithmus gerechter, wenn man sich die durchschnittliche Laufzeit bei Gleichverteilung aller $n!$ vielen möglichen Reihenfolgen der Elemente $A[1],\ldots,A[n]$ ansieht, anstatt die schlechtest mögliche Laufzeit. „Gleichverteilung“ bedeutet, dass jede mögliche Permutation mit dem gleichen Gewicht $1/(n!)$ eingehen soll.

Sei Π die Menge aller Permutationen von $1,\ldots,n$. Für $\pi \in \Pi$ sei $C(\pi)$ die Anzahl von Vergleichen, die Quicksort benötigt, um π zu sortieren. Dann ist die durchschnittliche Anzahl der Vergleiche:

$$V(n) = \frac{1}{n!}\sum_{\pi\in\Pi} C(\pi)$$

Im Folgenden schätzen wir $V(n)$ nach oben ab.

Teile die Menge Π aller Permutationen in die Mengen $\Pi_1,\ldots,\Pi_n$, wobei gilt:

$$\Pi_k := \{\pi \in \Pi \mid \text{das } k\text{-te Element wird festgehalten}\}$$

In so einer Menge Π_k ist also genau ein Element fixiert, alle anderen können in beliebiger Reihenfolge auftreten. Also ist $|\Pi_k| = (n-1)!$ für $k = 1,\ldots,n$.

Für alle $\pi \in \Pi_k$ ergibt sich die Zerlegung in Quicksort so, dass zwei Teilarrays π_1 (von $1,2,\ldots,k-1$) und π_2 (von $k+1,\ldots,n$) entstehen. Die Anzahl der Vergleiche, mit denen π in π_1 und π_2 zerlegt wird ist $\le n$. Dann gilt:

$$C(\pi) \le n + C(\pi_1) + C(\pi_2)$$

Summiert man über alle $\pi \in \Pi_k$, so ergibt sich wegen $|\Pi_k| = (n-1)!$ die Abschätzung:

$$\sum_{\pi\in\Pi_k} C(\pi) \le \sum_{\pi\in\Pi_k} n + \sum_{\pi\in\Pi_k} C(\pi_1) + \sum_{\pi\in\Pi_k} C(\pi_2) =: S_1 + S_2 + S_3$$

Hierbei ist $S_1 = \sum_{\pi\in\Pi_k} n = (n-1)!\cdot n = n!$. Wenn π alle Permutationen von Π_k durchläuft, entstehen bei π_1 alle Permutationen von $1,\ldots,k-1$, und zwar jede davon $((n-1)!/(k-1)!)$-mal, da Π_k ja insgesamt $(n-1)!$ Permutationen enthält. Also folgt:

$$S_2 = \frac{(n-1)!}{(k-1)!}\cdot\sum_{\pi_1} C(\pi_1) = (n-1)!\cdot V(k-1)$$

Analog gilt $S_3 = (n-1)!\cdot V(n-k)$. Durch Zusammensetzen aller Gleichungen bzw. Ungleichungen erhalten wir:

$$\begin{aligned}
V(n) &= \frac{1}{n!}\sum_{\pi\in\Pi} C(\pi) = \frac{1}{n!}\sum_{k=1}^{n}\sum_{\pi\in\Pi_k} C(\pi)\\
&\le \frac{1}{n!}\sum_{k=1}^{n}[n! + (n-1)!\cdot V(k-1) + (n-1)!\cdot V(n-k)]\\
&= \frac{n!}{n!}\cdot\sum_{k=1}^{n} 1 + \frac{(n-1)!}{n!}\cdot\sum_{k=1}^{n} V(k-1) + \frac{(n-1)!}{n!}\sum_{k=1}^{n} V(n-k)\\
&= n + \frac{1}{n}\sum_{k=1}^{n} V(k-1) + \frac{1}{n}\sum_{k=1}^{n} V(k-1)\\
&= n + \frac{2}{n}\sum_{k=1}^{n} V(k-1)
\end{aligned}$$

Beachtet man noch die Anfangswerte der Rekursionsgleichung, so gilt:

$$V(0) = V(1) = 0, \quad V(2) = 1, \quad V(n) \le n + \frac{2}{n}\cdot\sum_{k=2}^{n-1} V(k) \quad \text{für} \quad n \ge 2$$

Die Lösung der Rekursionsgleichung ergibt

$$V(n) \le 2\cdot n\cdot\ln(n) = O(n\cdot\log(n))$$

Da die Anzahl der Vergleiche auch die Anzahl der Vertauschungen bestimmt, und die Anzahl Vertauschungen plus die Anzahl der Vergleiche die Laufzeit bestimmen, läuft der Quicksort-Algorithmus in Laufzeit $O(n\cdot\log(n))$.

4.5 Bucket Sort

Bisher wurden nur Sortieralgorithmen vorgestellt, die eine Laufzeit von $O(n \cdot \log n)$ haben. Mit entsprechenden Einschränkungen geht es aber auch schneller. Ein Beispiel dafür ist der Bucketsort-Algorithmus mit der Einschränkung, dass er nur Zahlen einer vorgegebenen Zahlenmenge sortieren kann:

Einschränkung: $a_1, ..., a_n \in 1, ..., M$
Nutze Array $L[1 : M]$ von linearen Listen, d.h. jeder Einrag von L ist eine Liste.
BucketSort (Eingabe $a_1, ..., a_n \in 1, ..., M$)
zu Beginn: $L[i]$ ist leer für alle $i = 1, ..., M$.

1. Für $j = 1, ..., n$: Hänge a_j an Liste $L[a_j]$ an.
2. Hänge die Listen $L[1], ..., L[M]$ zur Liste L hintereinander.
3. Durchlaufe L von vorne nach hinten und gebe die gelesenen Werte aus.

Für die Laufzeit und den Speicherbedarf ergibt sich $O(n + M)$.

4.6 Laufzeitvergleich

Es lässt sich nun also die Frage beantworten, ob es Sortieralgorithmen mit einer Laufzeit gibt, die schneller als $O(nlog(n))$ ist:

- „Ja“, wenn man Sortieralgorithmen entwirft, die nicht auf paarweisen Schlüsselvergleichen beruhen. Das erfordert spezielle Annahmen, wie z.B. beim Bucket-Sort.
- „Nein“, wenn man sich auf Algorithmen beschränkt, die paarweise Schlüssel vergleichen. In diesem Sinne sind also Heapsort und Mergesort Optimale Algorithmen.

Dennoch lässt sich für die Laufzeit von Sortieralgorithmen eine unter Schranke finden, dies zeigt der folgende Satz.

Satz 4.1. *Jeder deterministische Sortieralgorithmus, der auf paarweisen Vergleichen von Schlüsseln basiert, braucht zum Sortieren eines n-elementigen Arrays Sowohl im Worst-Case also auch im Mittel (bei Gleichverteilung) $\Omega(n \log(n))$ Vergleiche.*

Bemerkungen zu diesem Satz:

- Der Satz zeigt, dass Mergesort und Heapsort bzgl. Der Größenordnung ihrer Laufzeiten optimal sind, und dass Laufzeitunterschiede lediglich den O-Konstanten zuzuschreiben sind.
- Man beachte den Unterschied zu den bisherigen asymptotischen Abschätzungen für ein Problem:
  - Jene erhielten wir, indem ein konkreter Algorithmus, der das Problem löst, analysiert wurde.
  - Die im Satz formulierte Ω-Abschätzung bezieht sich jedoch auf alle möglichen Sortierverfahren (bekannte und unbekannte)!

Der Beweis des Satz soll nicht genau gezeigt werden, allerdings wird nun die Beweisidee erläutert:

- für ein Array mit n Elementen gibt es $n!$ verschiedene Anordnungen.
- wenn man ein Array nur durch paarweise Vergleiche und paarweise Vertauschungen sortiert, lassen sich die Datenbewegungen mit protokollieren und man bekommt heraus, welche Permutation man auf die Ausgangsfolge anwenden muss, um eine sortierte Folge zu erhalten.
- pro Vergleichsabfrage läßt sich jeweils die Hälfte der noch verbleibenden Permutationsmöglichkeiten ausschließen.
  Beispiel: Wenn man erfährt, dass $A[5] \geq A[6]$ ist, lassen sich alle Permutationen, die zu $A[5] < A[6]$ führen, ausschliessen.
- betrachtet man nun einen Algorithmus, der durch Vergleiche und Vertauschungen die sortierte Folge erzeugt, bewegt sich dieser durch einen Baum von möglichen Ergebnissen und Abfragen:
  Dieser Binärbaum hat an jedem Blatt eine der möglichen Permutationen, an allen anderen Knoten befinden sich Abfragen.
- Ein binärer Baum mit $n!$ Blättern hat eine Höhe von $\log(n!) \in \Omega(n \log(n))$. Somit lässt sich mit $\Omega(n \log(n))$ Abfagen die richtige der $n!$ Permutationen finden.

Ende