Skriptum zu Algorithmen, Datenstrukturen und Graphentheorie

Dr. Andreas Hohenauer

Version vom 14. September 2025

Inhaltsverzeichnis

Vorwort

Dieses Skriptum zu *Algorithmen, Datenstrukturen und Graphentheorie* ist als begleitende Unterlage zu verschiedenen Kursen für Studierende an HTLs, Kolleges, Aufbaulehrgängen, Fachhochschulen oder Universitäten konzipiert.

Je nach Lehrveranstaltung sind stets nur bestimmte Teile des Skriptums relevant. Andere Abschnitte können zur Vertiefung oder für das Selbststudium genutzt werden. Die Anhänge enthalten Übungsbeispiele samt Lösungen.

Einige Kapitel behandeln grundlegende Inhalte (z. B. Sortierverfahren, Datenstrukturen, Graphen), während andere Abschnitte bereits fortgeschrittenere Themen wie Komplexitätstheorie, Optimierungsverfahren oder mathematische Programmierung zum Gegenstand haben.

Welche Teile des Skriptums für eine konkrete Lehrveranstaltung relevant sind, wird jeweils im Rahmen der entsprechenden Lehrveranstaltung bekanntgegeben.

1 Grundlagen

1.1 Allgemeine Einleitung

Algorithmen sind präzise, endliche Handlungsanweisungen zur Lösung wohldefinierter Probleme. Sie bilden das Herzstück der Informatik: Von der Sortierung von Daten über die Routenplanung bis zur Analyse sozialer Netzwerke – überall kommen Algorithmen zum Einsatz. In diesem Abschnitt werden zentrale Grundbegriffe eingeführt und ein kurzer Überblick gegeben.

Historisch lässt sich der Begriff des algorithmischen Vorgehens bis ins 9. Jahrhundert zurückverfolgen: Der persische Gelehrte al-Chwarizmi schrieb frühe Beispiele systematischer Rechenanweisungen und beeinflusste damit die spätere Entwicklung der Algebra und der algorithmischen Denkweise.

1.1.1 Grundbegriffe

Definition 1.1 (Problemstellung)**:** Die **Problemstellung** ist die formale Spezifikation, welche Beziehung zwischen Eingaben und gültigen Ausgaben gefordert ist (z. B. „sortiere die Elemente nichtabsteigend“ oder „finde einen kürzesten Pfad“).

Definition 1.2 (Algorithmus)**:** Ein **Algorithmus** ist eine endliche, eindeutig beschriebene Folge von Anweisungen, die für jede zulässige Eingabe nach endlich vielen Schritten eine Ausgabe liefert (Terminierung) und dabei die geforderte Aufgabe korrekt löst (Korrektheit).

Definition 1.3 (Instanz)**:** Eine **Instanz** eines Problems ist eine konkrete Eingabe, auf die das Problem angewandt werden soll (z. B. eine Liste konkreter Zahlen für das Sortierproblem oder ein bestimmter Graph für ein Pfadproblem).

Definition 1.4 (Ausgabe/Ergebnis)**:** Die **Ausgabe** (das **Ergebnis**) eines Algorithmus ist das von ihm für eine gegebene Instanz produzierte Objekt (Wert, Struktur, Zeuge), das die Spezifikation der Problemstellung erfüllt. Korrektheit bedeutet, dass jede erzeugte

Ausgabe eine *gültige Lösung* ist, und Vollständigkeit (wo gefordert), dass eine Lösung gefunden wird, sofern eine existiert.

Zusammengefasst: Die Problemstellung beschreibt, was gelöst werden soll; eine Instanz ist ein konkreter Eingabefall; ein Algorithmus verarbeitet die Eingabedaten und liefert als Ausgabe eine gültige Lösung gemäß Spezifikation.

1.1.2 Arten von Algorithmen

Es gibt eine Reihe verbreiteter Entwurfsparadigmen. Greedy-Verfahren treffen in jedem Schritt eine lokal optimale Entscheidung und zielen auf eine gute (manchmal optimale) Gesamtlösung. Divide-and-Conquer zerlegt das Problem in Teilprobleme, löst diese rekursiv und setzt die Teilergebnisse effizient zusammen (klassisch: Mergesort oder Quicksort). Dynamische Programmierung nutzt überlappende Teilprobleme und speichert Zwischenergebnisse (etwa bei Varianten von kürzesten Wegen in DAGs oder beim Rucksackproblem). Suchbasierte Ansätze wie Backtracking und Branch-and-Bound durchforsten systematisch den Lösungsraum und verwerfen Teilräume mithilfe von Schranken. Spezialisierte Graphalgorithmen arbeiten direkt auf Graphstrukturen (z. B. BFS/DFS, Dijkstra, Minimal-Spannbäume). Für kontinuierliche Fragestellungen kommen numerische Verfahren zum Einsatz. Darüber hinaus nutzen parallele und verteilte Algorithmen mehrere Recheneinheiten beziehungsweise Systeme, und Online-Algorithmen treffen Entscheidungen ohne Kenntnis der Zukunft. Für harte Optimierungsprobleme liefern Approximationsalgorithmen mit garantierten Gütefaktoren oft praktikable Ergebnisse.

1.1.3 Anwendungsgebiete

Algorithmen durchziehen nahezu alle Bereiche der Informatik und angrenzender Disziplinen: In der Datenverarbeitung sorgen sie für effizientes Sortieren, Suchen, Indizieren und Komprimieren. In Netzwerken und der Routenplanung bestimmen sie kürzeste Wege, maximale Flüsse, Matchings und steuern das Routing von Paketen. In Kryptographie und IT-Sicherheit stehen Zahlentheorie-basierte Verfahren, Primzahltests, Schlüsselaustausch und digitale Signaturen im Vordergrund. Die Bioinformatik verwendet Algorithmen für Sequenz-Alignment, Netzwerkanalysen und phylogenetische Rekonstruktionen. In Wirtschaft und Logistik (Operations Research) unterstützen sie Zeit- und Ressourcenplanung, Tourenplanung und Zuweisungsprobleme. In Computergrafik und Simulation kommen Verfahren für Rendering, Kollisionsdetektion und numerische Lösung von Differentialgleichungen zum Einsatz, und in Datenbanken sowie Suchmaschinen helfen sie bei

Abfrageoptimierung, Ranking und Deduplikation.

1.2 Mengenlehre

Die Mengenlehre bildet die Grundlage für viele Konzepte in der Informatik und Mathematik. Sie beschäftigt sich mit der Untersuchung von Mengen, ihren Eigenschaften und den Beziehungen zwischen ihnen. In diesem Abschnitt werden grundlegende Begriffe und Operationen der Mengenlehre eingeführt, die für das Verständnis von Algorithmen und Datenstrukturen unerlässlich sind.

Als *Elemente* bezeichnet man die Objekte, die zu einer Menge gehören. Mengen werden durch Aufzählung in geschweiften Klammern angeben, oder beschreibend über eine Eigenschaft. Die Schreibweise $\{x : \text{Bedingung}\}$ liest sich als: „Alle x, die die Bedingung erfüllen“. Dabei gilt stets: Wiederholungen spielen keine Rolle, und die Reihenfolge der Elemente ist unerheblich.

Definition 1.5 (Menge nach Cantor)**:** Eine Menge ist eine „Zusammenfassung bestimmter, wohlunterschiedener Objekte unserer Anschauung oder unseres Denkens zu einem Ganzen”.

Wichtige Eigenschaften von Mengen:

- Jedes Element kann nur einmal in einer Menge enthalten sein
- Die Reihenfolge der Elemente ist irrelevant
- Es wird nur festgehalten, ob ein Element zur Menge gehört oder nicht

Notation:

- Leere Menge: $\emptyset$ oder $\{\}$
- Zugehörigkeit: $x \in A$ (x ist Element von A)
- Nicht-Zugehörigkeit: $x \notin A$

Zum Beispiel gilt: $\{1, 1, 2\} = \{1, 2\}$. Die Aussage $x \in \{1, 2, 3\}$ bedeutet, dass x eines der aufgeführten Elemente ist.

1.2.1 Operationen auf Mengen

Betrachten wir die Mengen:

$$A = \{1, 2, 3, 4, 5\}$$
$$B = \{4, 5, 6, 7, 8\}$$
$$D = \{6, 7, 8\}$$
$$E = \{7, 8\}$$

Im folgenden Venn-Diagramm sind diese Mengen dargestellt:

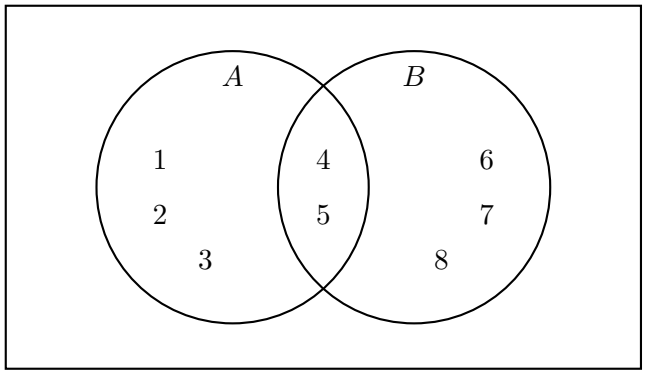


Abbildung 1.1: Venn-Diagramm der Mengen A und B

Zwischen den Kreisen befindet sich die Schnittmenge: Die dort stehenden Zahlen gehören sowohl zu A als auch zu B. Links liegen Elemente nur in A, rechts nur in B.

Definition 1.6 (Wichtige Mengenoperationen)**:** Seien A und B zwei Mengen. Dann definiert man:

- **Vereinigung:** $A \cup B = \{x : x \in A \text{ oder } x \in B\}$
- **Durchschnitt:** $A \cap B = \{x : x \in A \text{ und } x \in B\}$
- **Differenz:** $A \setminus B = \{x : x \in A \text{ und } x \notin B\}$
- **Symmetrische Differenz:** $A \triangle B = (A \setminus B) \cup (B \setminus A)$

Anschaulich: Die Vereinigung enthält alles, was in mindestens einer der beiden Mengen liegt. Der Durchschnitt enthält genau die gemeinsamen Elemente. Die Differenz $A \setminus B$

enthält die Elemente, die in A, aber nicht in B liegen (umgekehrt für $B \setminus A$). Die symmetrische Differenz enthält alle Elemente, die *genau in einer* der beiden Mengen liegen.

Für unser Beispiel:

$$A \cup B = \{1, 2, 3, 4, 5, 6, 7, 8\} \tag{1.1}$$
$$A \cap B = \{4, 5\} \tag{1.2}$$
$$A \setminus B = \{1, 2, 3\} \tag{1.3}$$
$$B \setminus A = \{6, 7, 8\} \tag{1.4}$$

Man erkennt: Nur 4 und 5 gehören zu beiden Mengen, die übrigen liegen jeweils nur in einer der beiden Mengen.

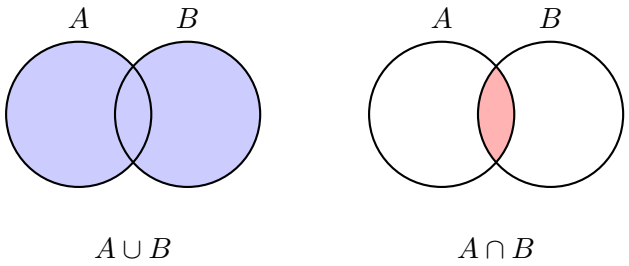


Abbildung 1.2: Mengenoperationen: Vereinigung und Durchschnitt

1.2.2 Teilmengen und Potenzmenge

Definition 1.7 (Teilmenge)**:** Eine Menge A heißt **Teilmenge** von B (Notation: $A \subseteq B$), wenn jedes Element von A auch Element von B ist:

$$A \subseteq B \Leftrightarrow \forall x \in A : x \in B$$

Dabei ist Gleichheit erlaubt: Jede Menge ist Teilmenge von sich selbst ($A \subseteq A$). Mit den oben definierten Mengen gilt beispielsweise, dass D vollständig in B liegt sowie E vollständig in D.

Definition 1.8 (Echte und unechte Teilmenge)**:** *Unechte* Teilmenge wird mit $A \subseteq B$ notiert und erlaubt Gleichheit ($A = B$ ist möglich). Eine *echte* Teilmenge wird mit $A \subset B$

(bzw. präziser $A \subsetneq B$) notiert und bedeutet:

$$A \subsetneq B \iff A \subseteq B \text{ und } A \neq B.$$

Anschaulich: A ist echte Teilmenge von B, wenn A in B *enthalten* ist und B mindestens ein Element mehr hat. Mit den oben definierten Mengen gilt z. B. $E \subsetneq D \subsetneq B$ sowie $B \subseteq B$ (unechte Teilmenge wegen Gleichheit).

Definition 1.9 (Potenzmenge)**:** Die **Potenzmenge** $\mathcal{P}(A)$ einer Menge A ist die Menge aller Teilmengen von A:

$$\mathcal{P}(A) = \{B : B \subseteq A\}$$

Beispiel 1.1: Für $A = \{1, 2\}$ ist:

$$\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$$

Allgemein gilt: Wenn $|A| = n$, dann $|\mathcal{P}(A)| = 2^n$.

Definition 1.10 (Kartesisches Produkt)**:** Das **kartesische Produkt** zweier Mengen A und B ist:

$$A \times B = \{(a, b) : a \in A \text{ und } b \in B\}$$

Hierbei ist die Reihenfolge wichtig: Im Allgemeinen gilt $(a, b) \neq (b, a)$.

Das kartesische Produkt ist fundamental für die Definition von Relationen und damit auch von Graphen, da Kanten als Elemente des kartesischen Produkts der Knotenmenge aufgefasst werden können.

1.3 O-Notation und Asymptotische Analyse

Software bearbeitet sehr unterschiedlich große Datenmengen: kleine Testbeispiele, große Datenbanken, umfangreiche Log-Dateien oder komplexe numerische Berechnungen. Bei kleinen Eingaben merkt man Performance-Unterschiede oft nicht; bei großen Eingaben können sie entscheidend sein. Die theoretische Laufzeitanalyse hilft, Engpässe (*Bottlenecks*) bereits beim Entwurf zu erkennen. Uns interessiert dabei nicht die exakte Ausführungszeit in Sekunden (die hängt auch von Hardware, Compiler, Sprache, Cache, ... ab), sondern wie stark die Laufzeit *wächst*, wenn die Eingabegröße größer wird. Dieses Wachstumsverhalten beschreiben wir mit der sogenannten Bachmann–Landau- oder kurz **O-Notation**. In der praktischen Softwareentwicklung kommen zusätzlich Werkzeuge wie *Profiler* zum Einsatz, um genau zu messen, wo Zeit verbraucht wird. Theorie (Asymptotik) und Praxis (Messung) ergänzen sich.

Was wird analysiert? Bei Algorithmen betrachten wir typischerweise:

- **Zeitkomplexität** (Anzahl elementarer Schritte / Vergleiche / Operationen)
- **Speicherplatzkomplexität** (zusätzlicher Speicherbedarf, z. B. Hilfsarrays, Rekursionsstack)
- Weitere zählbare Parameter (z. B. Anzahl Vergleiche, Anzahl Verschiebungen beim Sortieren)

Unterscheidung von Fällen:

- **Worst-Case**: ungünstigste Eingabe
- **Average-Case**: durchschnittliche (erwartete) Eingabe
- **Best-Case**: günstigste Eingabe

Die Eingabegröße, meist n, oder ggf. auch m beschreibt die Anzahl der relevanten Datensätze (Problemgröße).

Beispiele:

- Sortieren von n Zahlen
- Suchen eines Elementes unter n Einträgen
- Graphen mit n Knoten und m Kanten

Frage: Wie hängt die Laufzeit von n (und ggf. m) ab? Uns interessiert nur die **Größenordnung** – konstante Faktoren und niedrige Summanden lassen wir weg.

Definition 1.11 (O-Notation)**:** Eine Funktion $f(n)$ ist in $O(g(n))$, wenn es Konstanten $c > 0$ und $n_0 \geq 0$ gibt, so dass $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$.

Intuition: f wächst (ab einer genügend großen Eingabegröße) höchstens so schnell wie g bis auf einen konstanten Faktor. Beispiel: $3n^2 + 5n + 2 \in O(n^2)$, denn für $n \geq 1$ gilt $3n^2 + 5n + 2 \leq 10n^2$. Die O-Notation gibt also eine **obere Schranke** für das Wachstum an.

Definition 1.12 (Theta-Notation)**:** $f(n) \in \Theta(g(n))$, wenn es Konstanten $c_1, c_2 > 0$ und $n_0 \geq 0$ gibt mit $c_1 g(n) \leq f(n) \leq c_2 g(n)$ für alle $n \geq n_0$.

Intuition: f wächst (asymptotisch) genau so schnell wie g, weder wesentlich schneller noch langsamer. Wenn man die Laufzeit eines Algorithmus präzise klassifizieren kann, bevorzugt man $\Theta(\cdot)$ statt nur $O(\cdot)$. Die Theta-Notation gibt also eine **obere und untere Schranke** für das Wachstum an, und beschreibt somit die exakte Größenordnung.

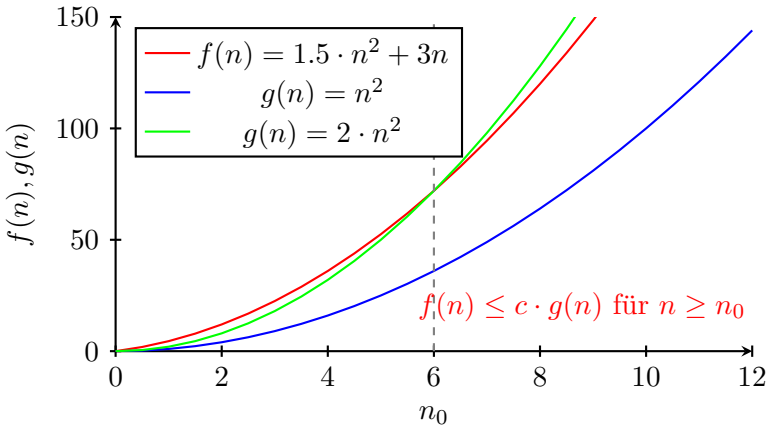


Abbildung 1.3: Grafische Darstellung der O-Notation: Ab einer gewissen Eingabegröße n_0 wächst $f(n)$ höchstens so schnell wie $g(n)$ bis auf einen konstanten Faktor c.

Häufige Komplexitätsklassen

Notation	**Name**	**Beispiel**
$O(1)$	konstant	Array-Zugriff
$O(\log n)$	logarithmisch	Binäre Suche
$O(n)$	linear	Lineare Suche
$O(n \log n)$	linearithmisch	Merge Sort
$O(n^2)$	quadratisch	Bubble / Insertion / Selection Sort (Worst)
$O(n^3)$	kubisch	Naive Matrixmultiplikation
$O(2^n)$	exponentiell	Teilmengen aufzählen
$O(n!)$	faktoriell	Alle Permutationen

Für große n:

$$O(1) \ll O(\log n) \ll O(n) \ll O(n \log n) \ll O(n^2) \ll O(2^n) \ll O(n!).$$

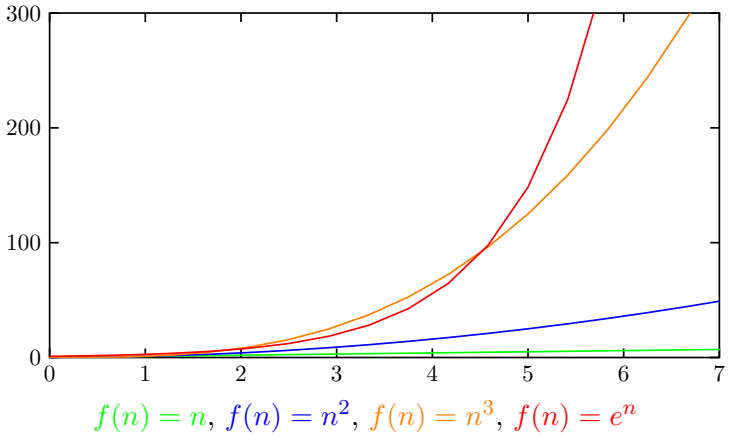


Abbildung 1.4: Darstellung von $f(n)$ für verschiedene Funktionen

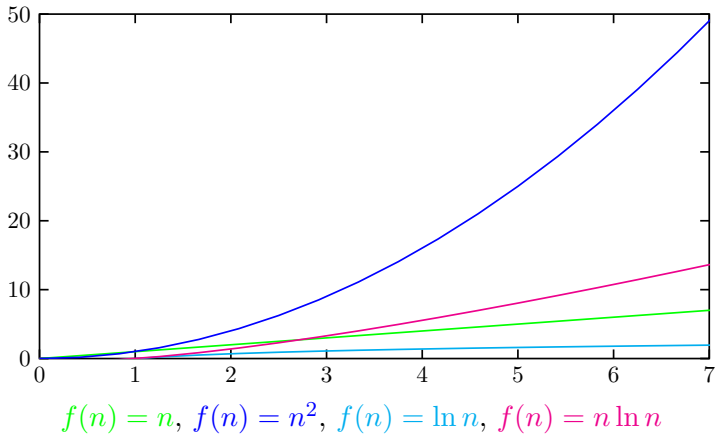


Abbildung 1.5: Weiteres Beispiel: die Logarithmus-Funktion. Diese strebt zwar gegen Unendlich, aber extrem langsam! Wichtige Konsequenz: $n \ln n$ verhält sich "fast" wie n.

Die Abbildungen 1.4 und 1.5 verdeutlichen die Unterschiede zwischen den verschiedenen Wachstumsraten. Insbesondere zeigt sich, dass die logarithmische Funktion zwar gegen Unendlich strebt, aber dies sehr langsam tut. Dies hat zur Folge, dass $n \ln n$ asymptotisch fast wie n wirkt. In der Praxis bedeutet dies, dass Algorithmen mit Laufzeit $O(n \log n)$ für viele praktische Eingabegrößen kaum langsamer sind als solche mit Laufzeit $O(n)$.

Rechenregeln (Faustregeln)

Theorem 1.1 (Addition)**:** $O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$ *(die größere Größenordnung dominiert)*

Theorem 1.2 (Multiplikation)**:** $O(f(n)) \cdot O(g(n)) = O(f(n)g(n))$

Beispiel 1.2: Beispiele:

- $3n^2 \in O(n^2)$
- $\frac{3}{4}n^3 + 5n^2 \in O(n^3)$
- $42n + \ln n + 5150 \in O(n)$
- $n \log n \in O(n^2)$
- $n \log n \in O(n \log n)$ (!) die kleinste obere Schranke!
- $n + m^2 \in O(n + m^2)$

Beispiel 1.3: Verschachtelte Schleifen:

```
for i = 1 to n:          // O(n)
    for j = 1 to n:      // O(n)
        konstante Operation   // O(1)
```

Gesamt: $O(n) \cdot O(n) \cdot O(1) = O(n^2)$. Hier liegt auch $\Theta(n^2)$ vor, weil wir asymptotisch sowohl obere als auch untere Schranke in der gleichen Größenordnung haben.

Beispiel 1.4 (Lineare Suche)**:** Suche nach Element x in Array der Größe n:

- Best-Case: x steht an erster Position $\Rightarrow O(1)$
- Worst-Case: x ist letzte Position oder nicht vorhanden $\Rightarrow O(n)$
- Average-Case (gleichverteilte Position): erwartete Position $n/2 \Rightarrow O(n)$

Die exakte zu erwartende Vergleichszahl ist $\approx n/2$, asymptotisch $\Theta(n)$.

Bei kleinen n kann ein Algorithmus mit schlechterer asymptotischer Ordnung praktisch schneller sein (z. B. Insertion Sort vs. Merge Sort für sehr kleine Arrays). Ab einer gewissen Schwelle dominiert jedoch die Ordnung.

Neben der Laufzeit betrachten wir zusätzlichen Speicher:

- **In-Place**: $O(1)$ zusätzlicher Speicher (z. B. Selection Sort)
- **Nicht in-place**: zusätzlicher Speicher $O(n)$ (z. B. Merge Sort)
- **Rekursion**: Stack-Bedarf proportional zur Rekursionstiefe

Wenn die exakte Größenordnung bekannt ist, drückt die Verwendung der Θ-Notation dies präziser aus. Ist nur die obere Schranke bekannt, wird dies durch die O-Notation ausgedrückt. In Referenzen und API-Dokumentationen wird oft nur die O-Notation angegeben und auf eine genauere Differenzierung verzichtet.

Bei der normalen Laufzeitanalyse betrachtet man die Kosten einer einzelnen Operation isoliert. Manche Datenstrukturen haben jedoch einzelne teure Operationen, die durch viele kostengünstige ausgeglichen werden. Die **amortisierte Laufzeit** einer Operation ist die *durchschnittliche Kosten* pro Operation über eine Folge von Operationen, wobei teure Operationen durch viele günstigere ausgeglichen werden.

1.4 Einfache Sortieralgorithmen

In diesem Abschnitt betrachten wir drei elementare Sortierverfahren: Bubble Sort, Insertion Sort und Selection Sort. Wir geben jeweils eine Beschreibung, Pseudocode sowie ein Beispiel mit den Zuständen des Arrays nach jedem Durchlauf der äußeren Schleife. Pseudocode ist eine beschreibungssprachliche Darstellung von Algorithmen, die wie Programmcode aussieht, aber unabhängig von einer konkreten Programmiersprache ist. Er dient dazu, logische Abläufe klar und verständlich zu formulieren, ohne sich um Syntaxdetails kümmern zu müssen. Alle Pseudocodes arbeiten mit *0-basierten* Indizes (wie in vielen Programmiersprachen üblich). `a.length` bezeichnet die Feldlänge, `swap(a,i,j)` vertauscht zwei Elemente.

Als Beispiel-Array verwenden wir für alle drei Verfahren dieselbe Ausgangsfolge von 10 Zahlen im Bereich 0–20:

$$A_0 = [9,\ 4,\ 12,\ 1,\ 7,\ 3,\ 10,\ 5,\ 2,\ 8].$$

1.4.1 Bubble Sort

Idee: In aufeinanderfolgenden Durchläufen werden benachbarte Elemente verglichen und bei Bedarf vertauscht. Große Elemente „blubbern“ nach rechts. Wenn in einem Durchlauf keine Vertauschung erfolgt, ist das Array bereits sortiert und der Algorithmus kann abbrechen.

Algorithmus 1: Bubble Sort

```
Input: Array a
Result: Sortiertes Array a
Function BubbleSort(a)
    for i = 0; i < a.length − 1; i++ do
        swapped = false
        for j = 0; j < a.length − 1 − i; j++ do
            if a[j] > a[j + 1] then
                Swap(a, j, j+1)
                swapped = true
        if ¬swapped then
            break
```

Beispielzustände pro Durchlauf (nach Abschluss des jeweiligen äußeren Durch-

laufs i):

Durchlauf i	Array-Zustand
Initial	9 4 12 1 7 3 10 5 2 8
0	4 9 1 7 3 10 5 2 8 12
1	4 1 7 3 9 5 2 8 10 12
2	1 4 3 7 5 2 8 9 10 12
3	1 3 4 5 2 7 8 9 10 12
4	1 3 4 2 5 7 8 9 10 12
5	1 3 4 2 5 7 8 9 10 12
6	1 3 2 4 5 7 8 9 10 12
7	1 2 3 4 5 7 8 9 10 12
8	1 2 3 4 5 7 8 9 10 12 (sortiert)

In Durchlauf 5 ändert sich die Position von Elementen in den ersten fünf Indizes nicht mehr außer einer internen Verschiebung später; vollständige Sortierung wird in Durchlauf 8 erreicht, danach würde der Algorithmus (im nächsten Durchlauf ohne Vertauschung) abbrechen.

1.4.2 Insertion Sort

Idee: Das Array wird logisch in einen bereits sortierten linken Teil und einen unsortierten rechten Teil zerlegt. In jedem Schritt wird das nächste Element durch Verschieben größerer Elemente nach rechts an die passende Stelle „eingefügt“.

Algorithmus 2: Insertion Sort

Input: Array a
Result: Sortiertes Array a
Function InsertionSort*(a)*
  $i = 1$
  while $i < a.length$ **do**
    $j = i$
    while $j > 0 \wedge a[j-1] > a[j]$ **do**
      Swap(a, j, j-1)
      $j = j - 1$
    $i = i + 1$

Beispielzustände (nach jedem abgeschlossenen äußeren Schritt i):

i (nächster Index)	Array-Zustand (linker Teil sortiert)
Initial	9 4 12 1 7 3 10 5 2 8
1	4 9 12 1 7 3 10 5 2 8
2	4 9 12 1 7 3 10 5 2 8
3	1 4 9 12 7 3 10 5 2 8
4	1 4 7 9 12 3 10 5 2 8
5	1 3 4 7 9 12 10 5 2 8
6	1 3 4 7 9 10 12 5 2 8
7	1 3 4 5 7 9 10 12 2 8
8	1 2 3 4 5 7 9 10 12 8
9	1 2 3 4 5 7 8 9 10 12 (sortiert)

Nach Schritt i ist das Teilarray $a[0..i]$ jeweils sortiert.

1.4.3 Selection Sort

Idee: In jedem Schritt wird das Minimum des unsortierten Bereichs bestimmt und an die aktuelle Position am Anfang dieses Bereichs getauscht. Anzahl der Vertauschungen ist höchstens $n - 1$.

Algorithmus 3: Selection Sort

Input: Array a
Result: Sortiertes Array a
Function SelectionSort*(a)*

```
for i = 0; i < a.length − 1; i++ do
    jMin = i
    for j = i + 1; j < a.length; j++ do
        if a[j] < a[jMin] then
            jMin = j
    if jMin ≠ i then
        Swap(a, i, jMin)
```

Beispielzustände (nach jedem äußeren Durchlauf i):

i	Array-Zustand
Initial	9 4 12 1 7 3 10 5 2 8
0	1 4 12 9 7 3 10 5 2 8
1	1 2 12 9 7 3 10 5 4 8
2	1 2 3 9 7 12 10 5 4 8
3	1 2 3 4 7 12 10 5 9 8
4	1 2 3 4 5 12 10 7 9 8
5	1 2 3 4 5 7 10 12 9 8
6	1 2 3 4 5 7 8 12 9 10
7	1 2 3 4 5 7 8 9 12 10
8	1 2 3 4 5 7 8 9 10 12 (sortiert)

Nach Durchlauf i ist das Präfix $a[0..i]$ sortiert und enthält die $i+1$ kleinsten Elemente.

Komplexitätsvergleich

- Bubble Sort: Vergleiche worst/avg $\Theta(n^2)$, best $\Theta(n)$ (früher Abbruch); Vertauschungen worst/avg $\Theta(n^2)$, best 0.
- Insertion Sort: Vergleiche / Verschiebungen worst $\Theta(n^2)$, avg $\Theta(n^2)$ (genauer $\approx n^2/4$), best $\Theta(n)$.
- Selection Sort: Vergleiche deterministisch $\Theta(n^2)$; Vertauschungen $\leq n-1$ ($\Theta(n)$); Laufzeit dominiert durch Vergleiche $\Theta(n^2)$.

2 Grundlagen der Graphentheorie

2.1 Einleitung in die Graphentheorie

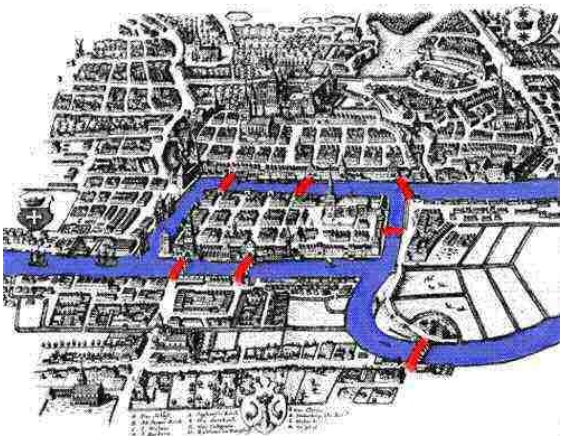
In diesem Kapitel führen wir in die Grundlagen der Graphentheorie ein. Dabei betrachten wir zunächst die historischen Ursprünge, die wichtigsten Anwendungsgebiete und die mathematischen Grundlagen, die für das Verständnis der Graphentheorie erforderlich sind.

2.1.1 Geschichte der Graphentheorie

Die Graphentheorie entstand im Jahr 1736 durch Leonhard Euler mit seiner Untersuchung des berühmten **Königsberger Brückenproblems**. Diese erste systematische Behandlung eines graphentheoretischen Problems legte den Grundstein für ein ganzes mathematisches Teilgebiet.

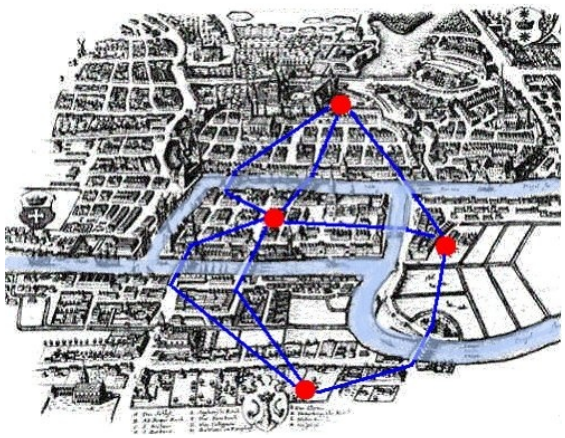
Das historische Königsberg (heute Kaliningrad) lag an beiden Ufern des Flusses Pregel und umfasste zwei Inseln, die durch sieben Brücken mit dem Festland und untereinander verbunden waren. Die Einwohner stellten sich die Frage:

Gibt es einen Weg, der jede der sieben Brücken über den Fluss Pregel genau einmal benutzt?



Euler erkannte, dass die spezifische geografische Anordnung der Brücken irrelevant war und abstrahierte das Problem zu einem *mathematischen Modell*. Er repräsentierte:

- Die vier Landmassen als **Knoten**
- Die sieben Brücken als **Kanten**



Diese Abstraktion war revolutionär, da sie erstmals ein reales Problem durch eine mathematische Struktur - einen Graphen - modellierte.

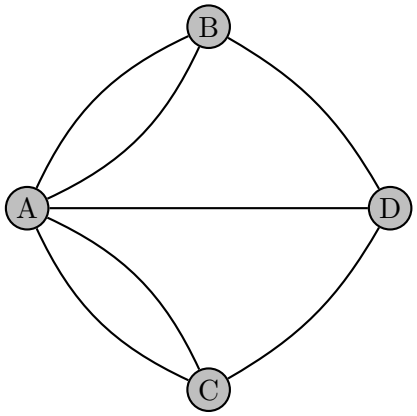
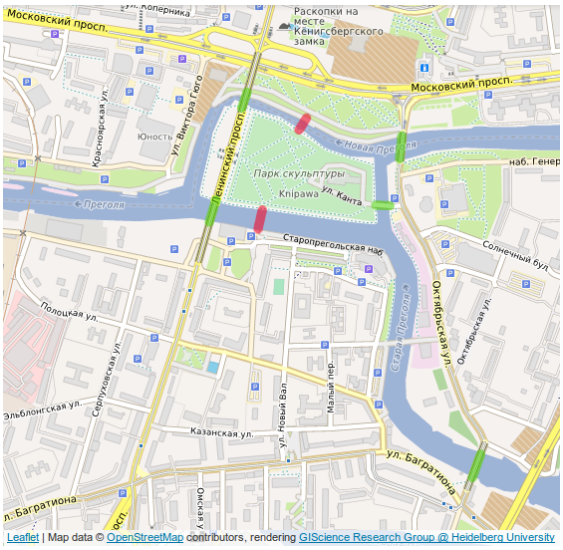


Abbildung 2.1: Königsberger Brückenproblem als Graph mit 7 Brücken

Eulers Lösung: Euler bewies, dass ein solcher Weg nicht existiert. Seine Argumentation basierte auf der Beobachtung, dass für einen geschlossenen Weg, der jede Kante genau einmal durchläuft (einen *Eulerschen Kreis*, bzw. *Euler-Zyklus*), jeder Knoten einen geraden Grad haben muss. Da im Königsberger Problem alle vier Knoten einen ungeraden Grad haben, kann ein solcher Weg nicht existieren.

Von den historischen sieben Brücken existieren heute nur noch fünf. Königsberg trägt seit 1946 den Namen Kaliningrad und ist eine russische Exklave. Das ursprüngliche Problem hat damit seine praktische Relevanz verloren, jedoch legte Eulers Herangehensweise den

Grundstein für die moderne Graphentheorie.



2.1.2 Anwendungen der Graphentheorie

Die Graphentheorie hat sich zu einem fundamentalen Werkzeug in vielen Bereichen der Mathematik, Informatik und darüber hinaus entwickelt. Die Vielseitigkeit von Graphen als Modellierungsinstrument zeigt sich in zahlreichen praktischen Anwendungen.

2.1.3 Navigationssysteme

Moderne Navigationssysteme basieren auf graphentheoretischen Algorithmen:

- **Modellierung:** Das Straßennetz wird als Graph repräsentiert
- **Knoten:** Kreuzungen, Abzweigungen, wichtige Punkte
- **Kanten:** Straßensegmente mit Gewichtungen (Entfernung, Fahrtzeit, etc.)
- **Algorithmen:** Kürzeste-Wege-Algorithmen wie Dijkstra oder A*

Die Berechnung erfolgt in Echtzeit und berücksichtigt aktuelle Verkehrsinformationen, Baustellen und andere dynamische Faktoren.

Mobilfunk und Frequenzplanung

In der Mobilfunkindustrie entstehen Optimierungsprobleme, die elegant durch Graphenfärbung gelöst werden:

- **Problem:** Senderstandorte in räumlicher Nähe dürfen nicht dieselbe Frequenz verwenden (Interferenzen)
- **Modellierung:**
  - Knoten repräsentieren Senderstandorte
  - Kanten verbinden Standorte, die sich gegenseitig stören würden
  - Farben repräsentieren Frequenzen
- **Ziel:** Minimale Anzahl von Frequenzen (Farben) finden

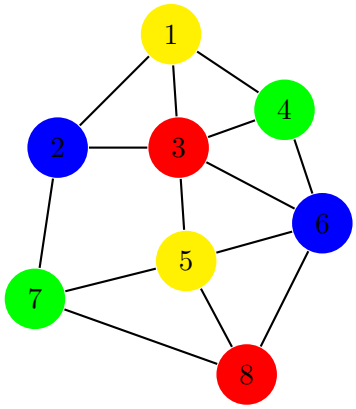


Abbildung 2.2: Färbung eines Interferenzgraphen mit 4 Farben (Frequenzen)

Das Problem des Handlungsreisenden (TSP)

Das *Travelling Salesman Problem* ist ein archetypisches Optimierungsproblem:

- **Problem:** Ein Handlungsreisender möchte n Städte besuchen und dabei den kürzesten Rundweg finden
- **Komplexität:** Das Problem ist NP-schwer, d.h. es ist vermutlich nicht in polynomieller Zeit optimal lösbar [1]
- **Anwendungen:**
  - Logistik und Tourenplanung
  - Leiterplatten-Design (Bohrlöcher)
  - DNA-Sequenzierung
  - Halbleiterproduktion

[1] Mehr Informationen zum Thema Komplexitätsklassen und Komplexitätstheorie findet sich im entsprechenden Kapitel 7.

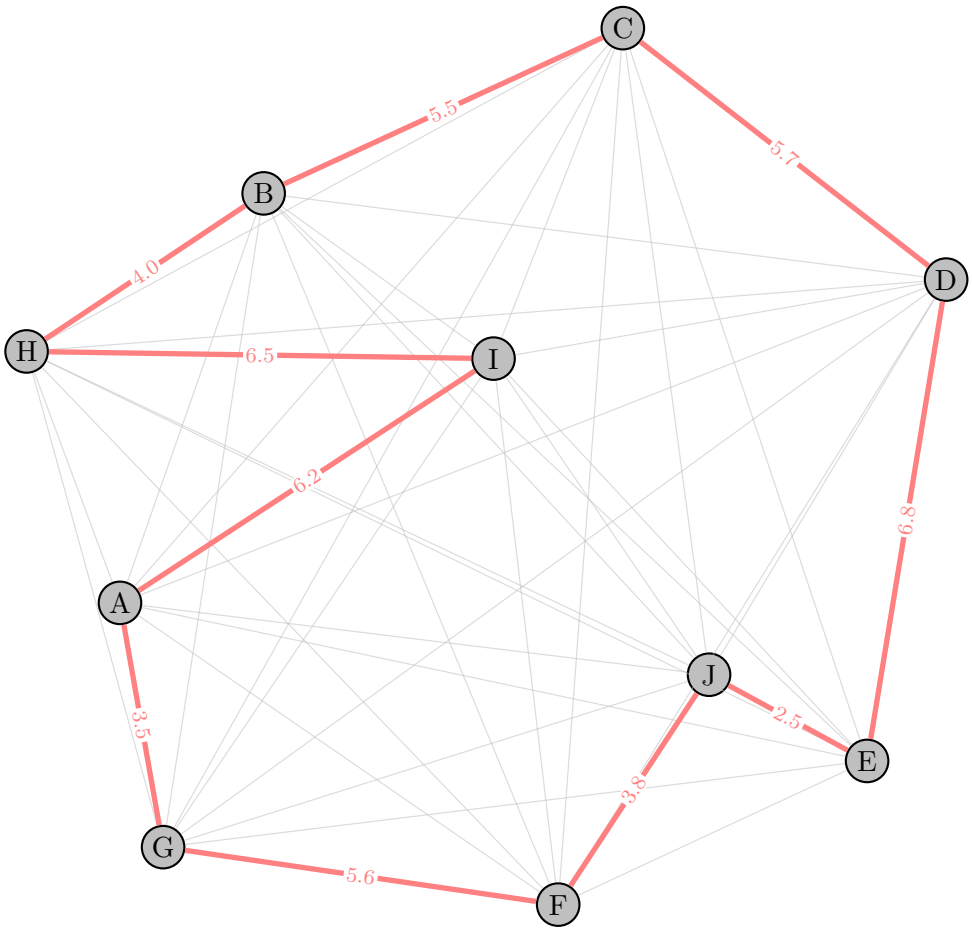


Abbildung 2.3: Lösung des TSPs (rote Kanten) mit Länge 50.02 mit zugrundeliegendem vollständigen Graphen.

Für praktische Anwendungen werden Heuristiken und Approximationsalgorithmen eingesetzt, die gute, aber nicht notwendigerweise optimale Lösungen in akzeptabler Zeit liefern.

Vehicle Routing Problem (VRP)

Eine Erweiterung des TSP für praktische Logistikanwendungen:

- **Erweiterungen:** Mehrere Fahrzeuge, Kapazitätsbeschränkungen, Zeitfenster
- **Anwendungen:** Paketdienste, Müllabfuhr, Krankentransport
- **Lösungsansätze:** Genetische Algorithmen, Simulated Annealing, Branch-and-Bound

Endliche Automaten

Graphen modellieren auch abstrakte Systeme wie endliche Automaten [2]:

- **Knoten:** Zustände des Systems
- **Kanten:** Zustandsübergänge
- **Anwendungen:** Compiler-Design, Protokollverifikation, Modellprüfung

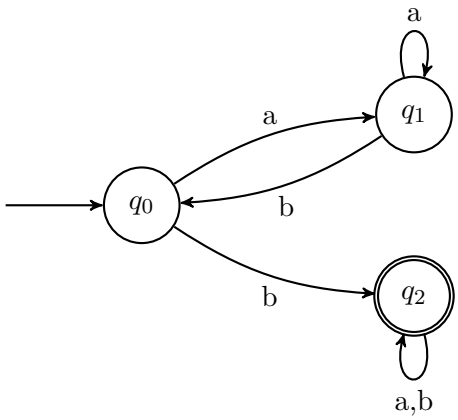


Abbildung 2.4: Endlicher Automat als gerichteter Graph

Weitere Anwendungsgebiete

Die vorgestellten Anwendungen sind nur exemplarisch, es gibt zahlreiche weitere Anwendungen von Graphen im Bereich der Modellierung von konkreten Problemstellungen, aber ebenso abstrakteren Bereichen wie Datenstrukturen oder Algorithmik. Die Graphentheorie findet Anwendung in vielen weiteren Bereichen:

- **Informatik:** Datenstrukturen, Netzwerkanalyse, Abhängigkeitsgraphen
- **Bioinformatik:** Proteinstrukturen, Genregulationsnetzwerke
- **Sozialwissenschaften:** Soziale Netzwerke, Epidemiologie
- **Ingenieurwesen:** Schaltungsdesign, Strukturanalyse
- **Operations Research:** Projektplanung, Ressourcenallokation

[2] Endliche Automaten haben eine wichtige Rolle in der Informatik, sowohl in der theoretischen Informatik als auch im Compilerbau (z.B. in der lexikalischen Analyse).

2.1.4 Grundlegende Definitionen

Bevor wir in die speziellen Graphtypen einsteigen, führen wir hier bereits die wichtigsten Grundbegriffe ein.

Definition 2.1 (Adjazente Knoten)**:** Zwei Knoten u und v sind **adjazent** (benachbart), wenn sie durch eine Kante verbunden sind.

Definition 2.2 (Adjazente Kanten)**:** Zwei Kanten sind **adjazent**, wenn sie einen gemeinsamen Knoten besitzen.

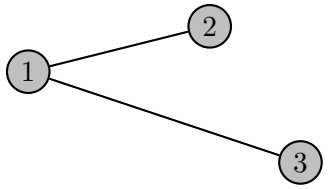


Abbildung 2.5: Adjazenz: Knoten 1 und 2 sind adjazent, ebenso die Kanten $\{1, 2\}$ und $\{1, 3\}$

Definition 2.3 (Inzidenz)**:** Ein Knoten ist **inzident** mit einer Kante, wenn der Knoten Endpunkt dieser Kante ist.

Definition 2.4 (Knotengrad)**:** Der **Grad** $d(v)$ eines Knotens v ist die Anzahl der Kanten, die mit v inzident sind.

Definition 2.5 (Endpunkt)**:** Ist $d(v) = 1$, nennt man v einen **Endpunkt** des Graphen.

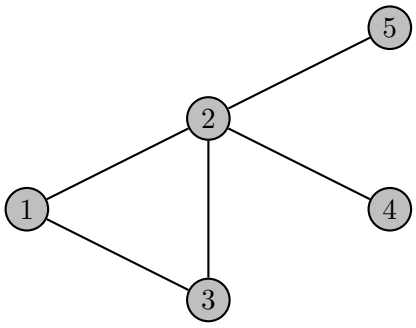


Abbildung 2.6: Knotengrade: $d(1) = 2$, $d(2) = 4$, $d(3) = 2$, $d(4) = 1$, $d(5) = 1$

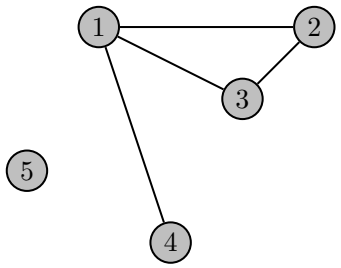


Abbildung 2.7: Beispiel eines isolierten Knotens (Knoten 4)

Knoten 4 und 5 sind Endpunkte, da sie Grad 1 haben.

Definition 2.6 (Isolierter Knoten)**:** Einen Knoten v mit $d(v) = 0$ nennt man *isolierten Knoten*.

Definition 2.7 (Gerichtete Kante)**:** Eine **gerichtete Kante** (i, j) ist eine Verbindung vom Knoten i zum Knoten j. Sie wird auch **Bogen** oder **Pfeil** genannt.

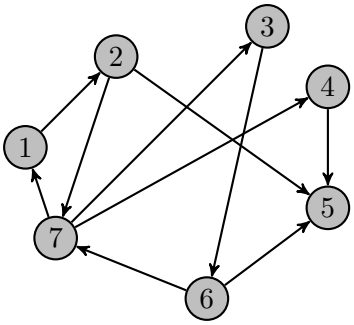


Abbildung 2.8: Beispiel eines gerichteten Graphen

Definition 2.8 (Mehrfachkante)**:** Verlaufen zwischen zwei Knoten mindestens zwei verschiedene Kanten, so heißt diese Menge von Kanten eine *Mehrfachkante* (oder *Multikante*).

Beispiel. Mehrfachkante zwischen den Knoten 1 und 4:

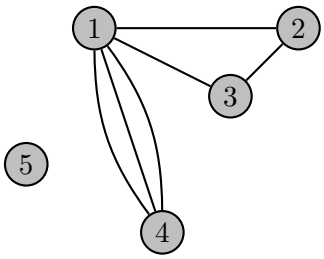


Abbildung 2.9: Mehrfachkante (drei parallele Kanten) zwischen den Knoten 1 und 4

Definition 2.9 (Schlinge)**:** Eine *Schlinge* ist eine Kante, die einen Knoten mit sich selbst verbindet.

Beispiel. Schlinge an Knoten 1:

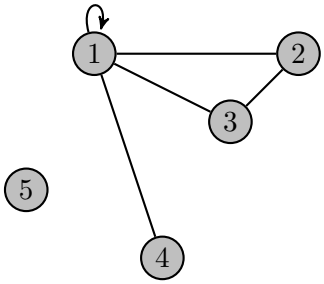


Abbildung 2.10: Schlinge am Knoten 1

Definition 2.10 (Schlichter Graph)**:** Ein *schlichter Graph* (auch: einfacher Graph) ist ein Graph ohne Schlingen und ohne Mehrfachkanten.

Konvention. Sofern nicht ausdrücklich anders angegeben, betrachten wir im Folgenden immer schlichte (einfache) Graphen.

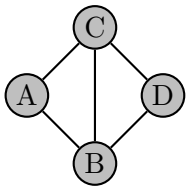
Darstellung eines Graphen

Ein Graph G ist ausschließlich durch die *Menge* V der Knoten und die *Menge* E der Kanten bestimmt. Wie genau die Knoten und Kanten gezeichnet werden, ist nicht relevant.

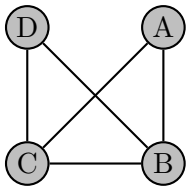
Beispiel. Graph $G = (V, E)$ mit

$$V = \{A, B, C, D\}, \qquad E = \{\{A, B\}, \{A, C\}, \{B, C\}, \{B, D\}, \{C, D\}\}.$$

Nachfolgend zwei unterschiedliche (aber gleichwertige) Darstellungen desselben Graphen:



Erste Darstellung



Zweite Darstellung

Beide Zeichnungen beschreiben denselben abstrakten Graphen. Unterschiede in Position oder Krümmung der Kanten ändern den Graphen nicht, solange die Inzidenzrelation (welche Knoten durch eine Kante verbunden sind) identisch bleibt.

2.2 Spezielle Graphen

In diesem Kapitel werden spezielle Graphenklassen und ihre Eigenschaften behandelt.

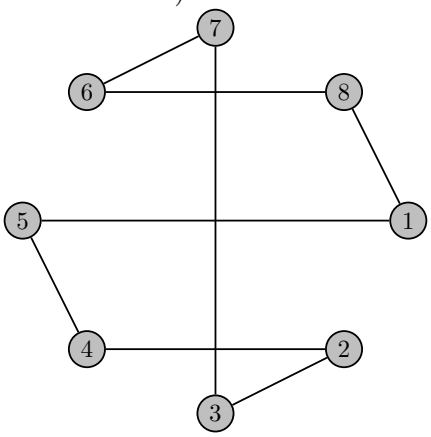
2.2.1 Reguläre Graphen

Definition 2.11 (Regulärer Graph)**:** Ein einfacher ungerichteter Graph $G = (V, E)$ heißt *regulär* vom Grad k, wenn alle Knoten $v \in V$ den gleichen Knotengrad k haben, also

$$\forall v \in V : \ d(v) = k.$$

Ist $k = 0$, so besteht G nur aus isolierten Knoten; bei $k = 1$ ist jede Zusammenhangskomponente eine einzelne Kante; bei $k = 2$ ist jede Zusammenhangskomponente ein Kreis.

Beispiel 2.1 (2-regulärer Graph)**:** Nachstehend ein zusammenhängender 2-regulärer Graph (ein einfacher Kreis mit 8 Knoten):



Jeder Knoten hat genau zwei inzidente Kanten.

2.2.2 Handshaking-Lemma

Intuitiv trägt jede Kante zu genau zwei Knotengraden (je ein Endpunkt) bei. Damit muss die Summe aller Knotengrade gerade sein.

Lemma 2.1 (Handshaking-Lemma)**:** *Für jeden ungerichteten Graphen $G = (V, E)$ gilt*

$$\sum_{v \in V} d(v) = 2\,|E|.$$

Folgerung: Jeder ungerichtete Graph besitzt eine gerade Anzahl von Knoten ungeraden Grades.

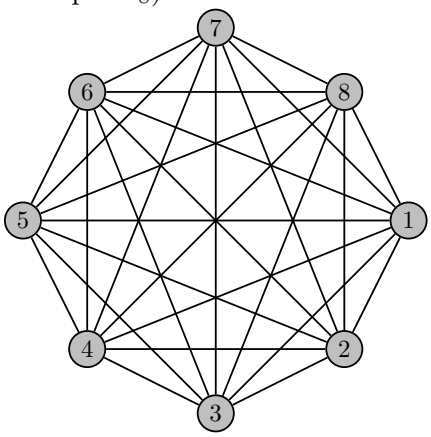
Übung 2.1 (Unmöglichkeit)**:** Zeigen Sie: Ein schlichter zusammenhängender 3-regulärer Graph mit 5 Knoten existiert nicht.

2.2.3 Vollständige Graphen

Definition 2.12 (Vollständiger Graph K_n)**:** Ein Graph G mit $|V| = n$ heißt *vollständig* (oder K_n), wenn er regulär vom Grad $n - 1$ ist (alle Knoten sind paarweise adjazent).

Eigenschaft: $|E| = \frac{n(n-1)}{2}$ (jede ungeordnete Knotenpaarung genau eine Kante).

Beispiel 2.2 (Vollständiger Graph K_8)**:**



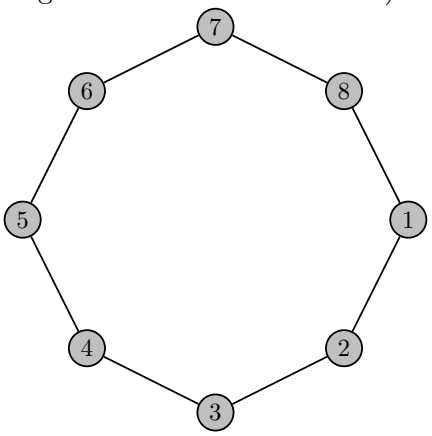
2.2.4 Komplementärgraph

Sei $G = (V, E)$ ein schlichter Graph und K die Menge aller 2-elementigen Teilmengen von V.

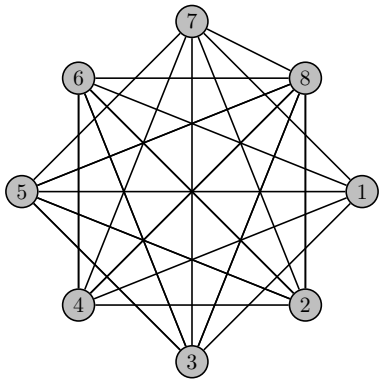
Definition 2.13 (Komplementärer Graph)**:** Der komplementäre Graph zu G ist $G' = (V, K \setminus E)$, d.h. er enthält genau die Kanten, die in G nicht vorkommen.

Bemerkung 2.1: Zwischen zwei Knoten existiert in genau einem der beiden Graphen G oder G' eine Kante.

Beispiel. Graph G (ein 2-regulärer Kreis mit 8 Knoten):



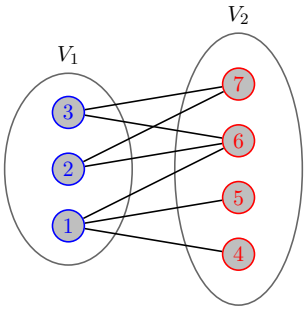
Komplement G' zu G:



2.2.5 Bipartite Graphen

Definition 2.14 (Bipartiter Graph)**:** Ein Graph $G = (V, E)$ heißt *bipartit*, wenn es eine Partition $V = V_1 \dot{\cup} V_2$ (d.h. $V_1 \cap V_2 = \emptyset$) gibt, so dass jede Kante $\{i, j\} \in E$ genau einen Endpunkt in V_1 und den anderen in V_2 besitzt.

Bemerkung 2.2: Man kann die Knoten in zwei Spalten (links / rechts) anordnen; Kanten verlaufen nur zwischen den Spalten, niemals innerhalb einer Spalte.



Beispiel 2.3 (Bipartiter Graph)**:**

2.2.6 Übungsaufgaben

Übung 2.2 (Gradvorgaben)**:** Konstruieren Sie einen schlichten Graphen mit 12 Knoten, der jeweils 3 Knoten vom Grad 3, 4, 5 und 6 enthält. Knoten gleichen Grades dürfen nicht adjazent sein.

Übung 2.3 (Gradvorgaben)**:** Konstruieren Sie einen schlichten, zusammenhängenden Graphen mit 14 Knoten, der jeweils

- 4 Knoten vom Grad 6,
- 2 Knoten vom Grad 5,
- 4 Knoten vom Grad 4,
- 2 Knoten vom Grad 3, und
- 2 Knoten vom Grad 2

hat. Knoten gleichen Grades dürfen nicht adjazent sein; Knoten vom Grad 2 sollen nicht adjazent zu Knoten vom Grad 3 und Grad 4 sein.

2.3 Kantenfolgen

Dieses Kapitel behandelt Kantenfolgen und Kantenzüge in ungerichteten Graphen. Diese sind zentrale Konzepte für die Analyse von Wegen und Pfaden in Graphen. Kantenfolgen ermöglichen es, die Struktur und Verbindungen zwischen Knoten zu untersuchen, während Kantenzüge eine spezielle Form von Kantenfolgen darstellen, bei denen jede Kante nur einmal verwendet wird. Diese Konzepte sind grundlegend für viele Algorithmen und Anwendungen in der Graphentheorie.

2.3.1 Kantenfolgen, Pfade, Kreise

Definition 2.15 (Kantenfolge)**:** Seien $v_1, \ldots, v_n$ Knoten eines (schlichten) ungerichteten Graphen $G = (V, E)$. Eine *Kantenfolge von* v_1 *nach* v_n ist eine endliche Folge von Kanten

$$[v_1, v_2], [v_2, v_3], \ldots, [v_{n-1}, v_n]$$

so dass jeweils zwei aufeinanderfolgende Kanten einen gemeinsamen Endknoten besitzen (also adjazent sind). Knoten und Kanten dürfen dabei grundsätzlich mehrfach vorkommen.

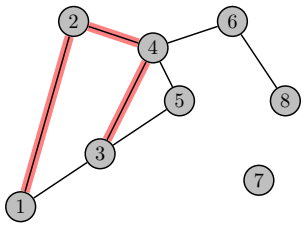
Definition 2.16 (Offene / geschlossene Kantenfolge)**:** Eine Kantenfolge $[v_1, v_2], \ldots, [v_{n-1}, v_n]$ heißt *offen*, falls $v_1 \neq v_n$, und *geschlossen*, falls $v_1 = v_n$.

Bemerkung 2.3: In einer Kantenfolge dürfen einzelne Kanten auch mehrfach auftreten (dann meist mit derselben Orientierung, da der Graph ungerichtet ist spielt die Reihenfolge der Endpunkte aber keine Rolle).

Beispiel 2.4 (Kantenfolge und Nichtbeispiele)**:** Betrachte die Kantenfolge

$$K = [1, 2],\ [2, 4],\ [4, 3],\ [3, 4],\ [4, 2].$$

Sie ist gültig, da jede aufeinanderfolgende Paarung von Kanten einen gemeinsamen Knoten besitzt (2,4,3,4,2). Dagegen wären $[1, 2], [2, 3]$ *keine* vollständige Kantenfolge *von 1 nach 3*, weil die Folge endet ohne weiteren Bezug, sowie $[1, 2], [4, 3]$ keine Kantenfolge, da die Kanten nicht adjazent sind.



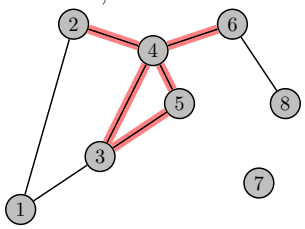
Definition 2.17 (Kantenzug)**:** Ein *Kantenzug* ist eine Kantenfolge, in der alle Kanten paarweise verschieden sind (keine Kante wird wiederverwendet).

Bemerkung 2.4: Analog zu Kantenfolgen unterscheidet man offene und geschlossene Kantenzüge je nach $v_1 \neq v_n$ bzw. $v_1 = v_n$.

Beispiel 2.5 (Kantenzug)**:** Kantenzug

$$K = [2,4],[4,5],[5,3],[3,4],[4,6]$$

ist offen (Start 2, Ende 6). Der Knoten 4 erscheint mehrfach, was zulässig ist (Beschränkung bezieht sich nur auf Kanten, nicht auf Knoten).



Definition 2.18 (Weg / Pfad)**:** Ein Weg oder Pfad $P(s,t)$ ist ein offener Kantenzug von s nach t, in dem zusätzlich alle Knoten (bis auf eventuell Start/Ende, die verschieden sind) paarweise verschieden sind. Es werden also keine Knoten (außer implizit verschiedenen Start/Endknoten) wiederholt.

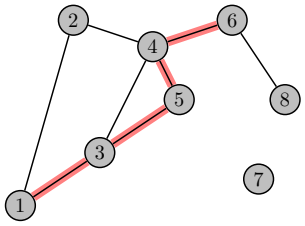
Bemerkung 2.5: In der Literatur existieren unterschiedliche Konventionen: Mitunter bezeichnet *Weg* bereits jede Kantenfolge (ggf. mit Knotenwiederholungen), *Pfad* dagegen eine Folge mit paarweise verschiedenen Knoten. Man findet auch die Bezeichnung *elementarer Pfad*. Hier verwenden wir Weg/Pfad synonym im Sinne der strikt knotendistinkten Variante.

Definition 2.19 (Weglänge)**:** Die (Kanten-)Länge $|P(s,t)|$ eines Weges ist die Anzahl seiner Kanten.

Definition 2.20 (Kürzester Weg)**:** Unter allen Wegen von s nach t heißen diejenigen mit minimaler Länge *kürzeste Wege* (oder kürzeste Pfade).

Beispiel 2.6 (Pfad)**:** Pfad

$$P(1,6) = [1,3],[3,5],[5,4],[4,6], \qquad |P(1,6)| = 4.$$



Es können mehrere unterschiedliche (gleich lange oder längere) Wege zwischen zwei Knoten existieren.

Definition 2.21 (Zyklus)**:** Ein Zyklus $(v_1, v_2, \ldots, v_{n-1}, v_n)$ ist ein geschlossener Kantenzug, also $v_1 = v_n$.

Definition 2.22 (Kreis)**:** Ein Kreis ist ein Zyklus, in dem alle Knoten bis auf Start- und Endknoten (die gleich sind) paarweise verschieden sind.

Achtung

Die Begriffe *Zyklus* und *Kreis* sind in der Literatur nicht vollständig einheitlich; manche Autoren verwenden beide synonym.

2.3.2 Eulersche Graphen

Definition 2.23 (Eulersche Linie)**:** Eine Eulersche Linie ist ein Kantenzug, der jede Kante des Graphen genau einmal enthält.

Definition 2.24 (Euler-Zyklus)**:** Eine geschlossene Eulersche Linie (Start- und Endknoten identisch) heißt Euler-Zyklus.

Definition 2.25 (Eulerscher Graph)**:** Besitzt ein Graph einen Euler-Zyklus, so heißt er eulersch.

Bemerkung 2.6: Bei einer Eulerschen Linie dürfen Knoten mehrfach besucht werden; einzig die Kanten dürfen nicht wiederholt werden.

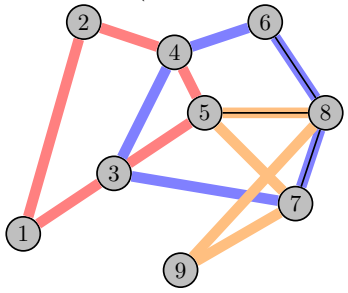
Theorem 2.1 (Satz von Euler–Hierholzer)**:** *Sei G ein zusammenhängender Graph. Dann sind äquivalent:*

1. *G ist eulersch.*
2. *Jeder Knoten von G hat geraden Grad.*
3. *Die Kantenmenge von G ist die Vereinigung paarweise kantendisjunkter Zyklen.*[3]

Bemerkung 2.7: Ein Graph besitzt genau dann eine *offene* Eulersche Linie (Euler-Pfad), wenn genau zwei Knoten ungeraden Grad besitzen; diese beiden sind Anfangs- bzw. Endknoten.

Königsberger Brückenproblem Siehe Abschnitt 2.1.1 für die Problemdefinition. Die Modellierung als Graph führt zur unmittelbaren Anwendung der Euler-Kriterien (mehr als zwei Knoten ungeraden Grades) und damit zur Verneinung.

Beispiel 2.7 (Euler-Zerlegung)**:** Die Kanten des Graphen lassen sich als Vereinigung dreier kantendisjunkter Zyklen schreiben (farblich markiert); also ist der Graph eulersch.



[3] disjunkt bezüglich ihrer Kanten

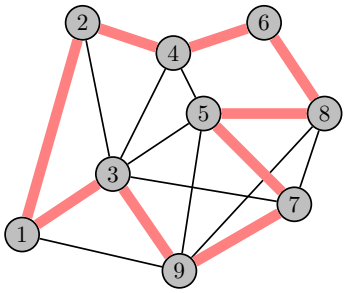
2.3.3 Hamiltonkreise

Definition 2.26 (Hamiltonsche Linie)**:** Eine *Hamiltonsche Linie* ist ein Weg, der alle Knoten des Graphen *genau* einmal enthält.

Bemerkung: In einer Hamiltonsche Linie dürfen die Knoten nicht mehrfach (wiederholt) besucht werden!

Definition 2.27 (Hamilton-Kreis)**:** Ein *Hamilton-Kreis* ist ein Kreis der alle Knoten des Graphen genau einmal enthält.

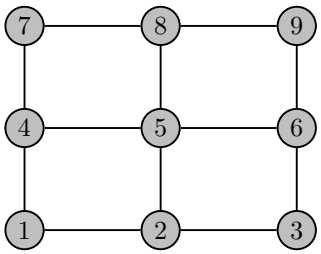
Beispiel 2.8: Ein Beispiel für einen Hamilton-Kreis ist der folgende:



Definition 2.28 (Hamiltonscher Graph)**:** Einen Graphen der einen Hamiltonkreis enthält nennt man *Hamiltonschen Graph.*

- Nicht alle Graph sind Hamiltonsch!
- **Es existiert keine einfache Charakterisierung ob ein Graph Hamiltonsch ist!**
- Im Gegensatz zum leicht lösbaren Problem des Euler-Zyklus, ist das Hamiltonkreis-problem im Allgemeinen sehr schwierig zu lösen.

Beispiel 2.9: Beispiel für einen nicht-Hamiltonschen Graphen:



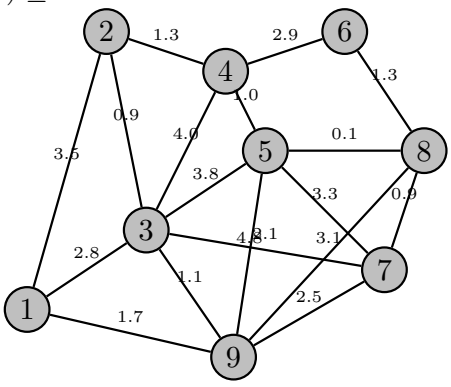
Das Hamiltonkreis-Problem liegt in der Komplexitätsklasse $NP-$vollständig:

- NP: *"nondeterministic polynomial"* (siehe Kapitel 7)
- Es kann nicht in *polynomieller Zeit* in Abhängigkeit von den Eingabedaten entschieden werden ob eine Lösung existiert
- Ist enthalten in Liste der *21 klassischen NP-vollständigen Probleme* (Richard Karp, 1972)

2.3.4 Travelling-Salesman-Problem

Definition 2.29 ((Kanten-)Gewichteter Graph)**:** Sei $w : E \to \mathbb{R}$ eine Abbildung die jeder Kante $e \in E$ eine reelle Zahl zuordnet. Man bezeichnet w als die Gewichtsfunktion.

Beispiel 2.10: Dieses Beispiel zeigt einen derartigen gewichteten Graphen, wobei für alle Kanten $e \in E$ gilt $w(e) \geq 0$.



Wir betrachten nun einen *vollständigen* Graphen mit Kantengewichten $w(e) \geq 0$ für alle $e \in E$. Existiert hier ein Hamiltonkreis? Es ist leicht ersichtlich, dass in einem

vollständigen Graphen jeder Knoten mit jedem anderen Knoten verbunden ist. Es gibt tatsächlich sogar

$$\frac{(|V|-1)!}{2}$$

verschiedene Hamiltonkreise! Alle Permutationen der Knoten ergeben einen gültigen Hamiltonkreis!

Im Gegensatz zum *Entscheidungsproblem* "gibt es einen Hamiltonkreis" wollen wir nun das *Optimierungsproblem* betrachten:

Definition 2.30 (Traveling-Salesman-Problem)**:** Finde hinsichtlich der Kantengewichte $w(e)$ den *kürzesten* Hamiltonkreis H, also

$$\min_H \sum_{e \in H} w(e).$$

Dieses schwierige Optimierungsproblem hat zahlreiche Anwendungen, z.B. in der Logistik:

Beispiel 2.11: Beliefere ausgehend von einem Lagerstandort 50 Kunden in Wien.

Modellierung:

- Die Kunden werden durch Knoten V modelliert
- Wir betrachten den vollständigen Graphen G bezüglich V
- Kantengewichte $w(e_{ij})$ ergeben sich durch Berechnung der kürzesten Wege im Straßengraphen vom Kunden i zum Kunden j
- Ziel: Minimierung der Transportkosten
- Lösung: *kürzester* Hamiltonkreis in G

Aufgrund der Komplexität des Problems gibt es keine "einfachen" Algorithmen zu dessen Lösung. Gelöst werden kann es mit Methoden im Bereich der heuristischen Optimierung oder der Mathematischen Programmierung, siehe Kapitel 8.

Abbildung 2.11: Kundenstandorte und das Straßennetz von Wien (Quelle: OpenStreet-Map)

2.4 Eigenschaften von Graphen

In diesem Abschnitt betrachten wir wichtige Eigenschaften von Graphen wie Distanz, Exzentrizitäten, Radius, Durchmesser und Zentrum.

Definition 2.31 (Distanz (ungewichteter Graph))**:** Die Distanz (Abstand) zweier Knoten u, v ist die Länge eines kürzesten Weges $P(u, v)$ zwischen ihnen:

$$d(u, v) = \min_{P(u,v)} |P(u, v)|.$$

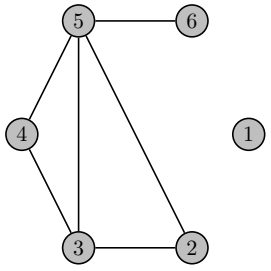
Existiert kein Weg, definieren wir $d(u, v) = \infty$. Ferner gilt $d(u, u) = 0$ und in ungerichteten Graphen Symmetrie $d(u, v) = d(v, u)$.

Definition 2.32 (Distanz (gewichteter Graph))**:** In einem gewichteten Graphen mit Kantengewichtsfunktion $w : E \to \mathbb{R}_{\geq 0}$ ist

$$d(u, v) = \min_{P(u,v)} \sum_{e \in P(u,v)} w(e),$$

wobei über alle Wege $P(u, v)$ von u nach v minimiert wird.

Beispiel 2.12 (Distanzwerte)**:**



$d(1, 1) = 0$

$d(1, 2) = \infty$ (keine Verbindung)

$d(2, 6) = 2$ via $2 - 5 - 6$

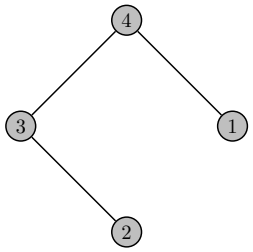
$d(3, 4) = 1$ direkt

Definition 2.33 (Exzentrizität)**:** Für einen *zusammenhängenden* ungerichteten Graphen $G = (V, E)$ ist die Exzentrizität eines Knotens v der größte Abstand zu irgendeinem Knoten:

$$ex(v) = \max_{u \in V} d(v, u).$$

In nicht zusammenhängenden Graphen ist wegen unerreichbarer Knoten $ex(v) = \infty$ für alle v.

Beispiel 2.13 (Exzentrizitäten in einem Pfad)**:**



Knoten	Exzentrizität	Entferntester Knoten
1	3	2
2	3	1
3	2	1
4	2	2

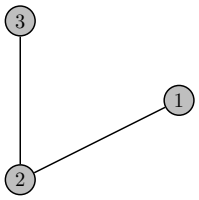
Bemerkung 2.8: Wird eine Kante entfernt und der Graph dadurch nicht zusammenhängend, wird jede Exzentrizität zu ∞.

Definition 2.34 (Durchmesser)**:** Der Durchmesser $dm(G)$ ist $\max_{v \in V} ex(v)$. Ist G nicht zusammenhängend, so $dm(G) = \infty$.

Definition 2.35 (Radius)**:** Der Radius $rad(G)$ ist $\min_{v \in V} ex(v)$. Für jeden zusammenhängenden ungerichteten Graphen gilt $rad(G) \leq dm(G) \leq 2\, rad(G)$.

Definition 2.36 (Zentrum)**:** Das Zentrum $Z(G) = \{v \in V : ex(v) = rad(G)\}$ ist die Menge aller Knoten minimaler Exzentrizität.

Beispiel 2.14 (Durchmesser und Radius)**:**

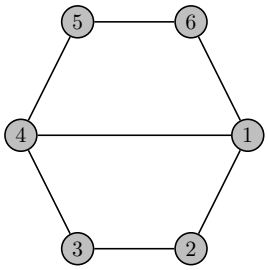


Knoten	ex
1	2
2	1
3	2

$dm(G) = 2,\ rad(G) = 1.$

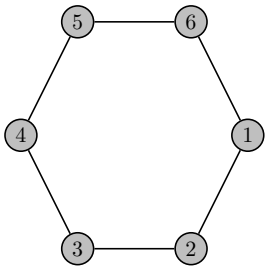
Beispiel 2.15 (Zentrum zweier Graphen)**:**

Graph G_1



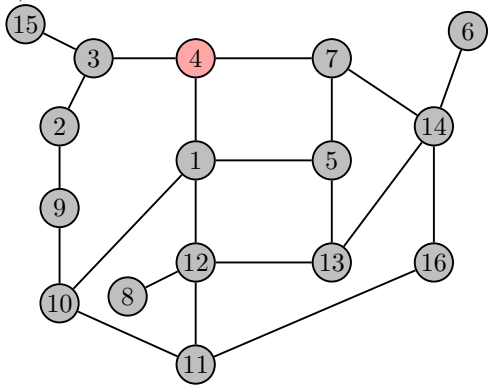
$dm(G_1) = 3,\ rad(G_1) = 2,\ Z(G_1) = \{1, 4\}.$

Graph G_2



$dm(G_2) = 3,\ rad(G_2) = 3,\ Z(G_2) = V(G_2).$

Beispiel 2.16 (Zentrum in Anwendung)**:** Optimale Position einer Feuerwache: Wähle einen Knoten im Zentrum, um maximale Reaktionszeit zu minimieren.



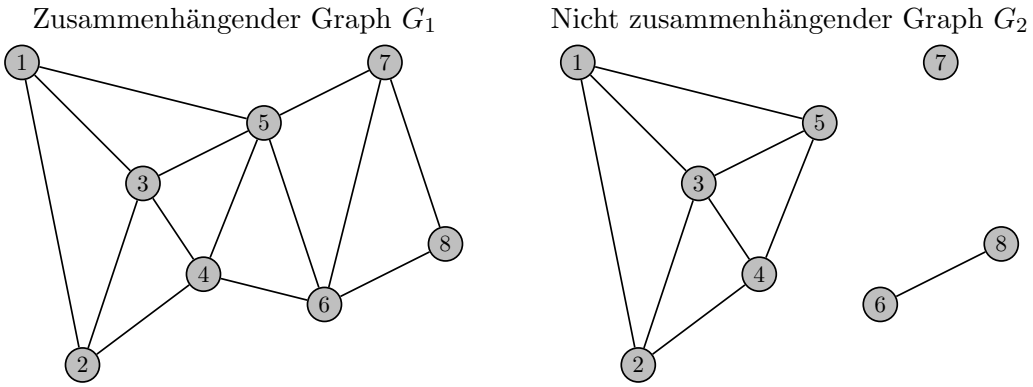
Knoten 4 (rot markiert) besitzt minimale Exzentrizität.

2.5 Zusammenhang

Definition 2.37 (Zusammenhang)**:** Ein ungerichteter Graph $G = (V, E)$ heißt *zusammenhängend*, wenn für alle $u, v \in V$ ein Weg zwischen u und v existiert (schreibe auch $u \rightsquigarrow v$).

Ein Graph ist also genau dann nicht zusammenhängend, wenn sich seine Knotenmenge in zwei (oder mehr) nichtleere Teile zerlegen lässt, so dass keine Kante zwischen diesen Teilen verläuft.

Beispiel 2.17 (Zusammenhängend vs. nicht zusammenhängend)**:**

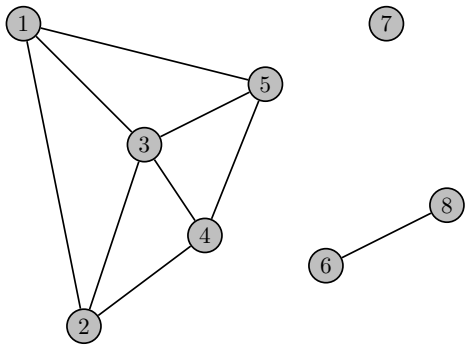


Definition 2.38 (Zusammenhangskomponente)**:** Die Zusammenhangskomponente eines Knotens v ist die Menge

$$K(v) = \{u \in V(G) \mid u \rightsquigarrow v\}.$$

Es ist die maximal zusammenhängende Menge von Knoten, die v enthält.

Beispiel 2.18 (Drei Komponenten in einem Graphen)**:** Graph G_3 besitzt drei Zusammenhangskomponenten.



$Z_1 = \{1, 2, 3, 4, 5\}$

$Z_2 = \{6, 8\}$

$Z_3 = \{7\}$

Definition 2.39 (Komponenten)**:** Die von Zusammenhangskomponenten aufgespannten gesättigten Teilgraphen heißen die *Komponenten* von G. Ihre Anzahl bezeichnen wir mit $c(G)$.

Beispiel 2.19 (Komponenten eines Graphen)**:** Für G_3 gilt $c(G_3) = 3$. Die Komponenten sind

$$K_1 = (\{1, 2, 3, 4, 5\}, \{12, 13, 15, 23, 24, 34, 35, 45\}),$$
$$K_2 = (\{6, 8\}, \{68\}),$$
$$K_3 = (\{7\}, \emptyset).$$

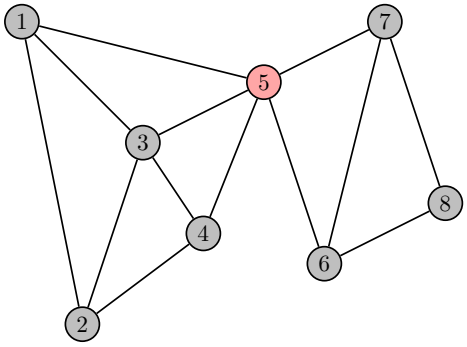
(Hier sind Kanten als konkatenierte Endpunkte notiert.)

2.5.1 Artikulationen, Brücken und Blöcke

Definition 2.40 (Artikulation)**:** Ein Knoten v heißt *Artikulation*, wenn $c(G-\{v\}) > c(G)$ gilt, also das Entfernen von v die Anzahl der Komponenten erhöht.

Artikulationen verbinden Blöcke des Graphen und haben immer mindestens Grad 2.

Beispiel 2.20 (Artikulation illustriert)**:** Graph G_4 mit Artikulation (rot markiert):

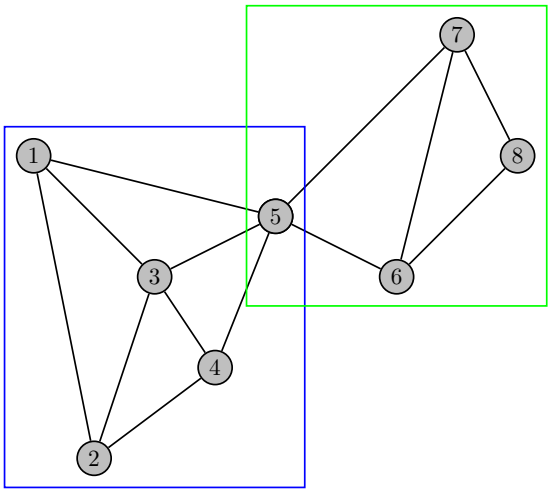


Entfernt man Knoten 5 zerfällt der Graph in zwei Komponenten.

Definition 2.41 (Block)**:** Ein Block ist ein zusammenhängender Teilgraph ohne Artikulationen, der in keinem echt größeren zusammenhängenden Teilgraphen ohne Artikulationen enthalten ist (maximal bzgl. Knotenmenge).

Bemerkung 2.9: Jede Kante und jeder Kreis liegen in genau einem Block; zwei Blöcke schneiden sich in höchstens einem (Artikulations-)Knoten; ein isolierter Knoten ist ein Block.

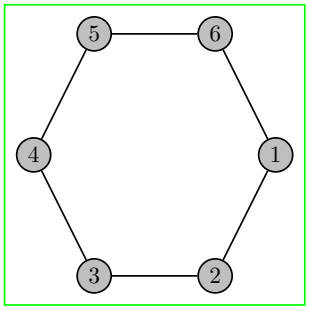
Beispiel 2.21 (Zwei Blöcke)**:** Graph G_5 mit zwei Blöcken (blau, grün umrahmt):



Knotenmenge: $B_1 = \{1, 2, 3, 4, 5\}$, $B_2 = \{5, 6, 7, 8\}$.

Beispiel 2.22 (Extremfälle von Blöcken)**:** Links: kleinster Block (isolierter Knoten). Rechts: zusammenhängender Graph ohne Artikulationen (ganzer Graph ist ein Block).

$B_1 = \{1\}$

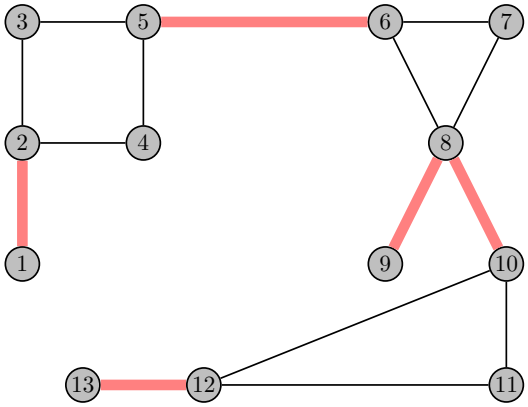


$B_1 = \{1, 2, 3, 4, 5, 6\}$

Definition 2.42 (Brücke)**:** Eine Kante e heißt Brücke, wenn $c(G - \{e\}) > c(G)$, also ihr Entfernen die Anzahl der Komponenten erhöht.

Bemerkung 2.10: Eine Brücke ist selbst ein Block und eine Schwachstelle (wie Artikulationen).

Beispiel 2.23 (Brücken in einem Graphen)**:** Graph G_8 mit fünf Brücken (rot markiert):



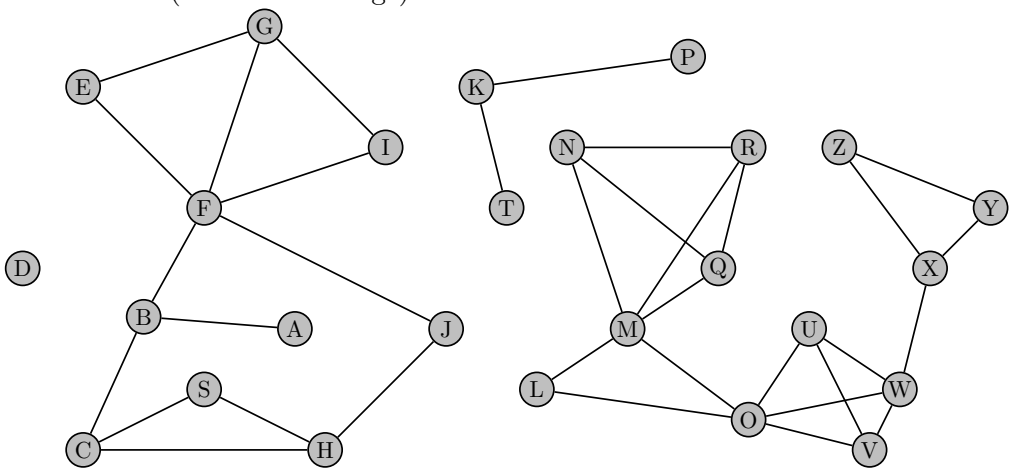
2.5.2 Aufgaben

Übung 2.4 (Bestimmung von Artikulationen und Brücken I)**:** Gegeben sei Graph G_8. Bestimmen Sie alle Artikulationen, Brücken und Blöcke. (Graph wie im vorherigen Beispiel.)

Übung 2.5 (Bestimmung von Artikulationen und Brücken II)**:** Graph G_9 (wie G_8 mit zusätzlicher Kante 1–12). Bestimmen Sie alle Artikulationen, Brücken und Blöcke.

Übung 2.6 (Umfassende Analyse eines Graphen)**:** Gegeben sei der Graph G_{10} (unten). Bestimmen Sie

1. alle Zusammenhangskomponenten,
2. alle Artikulationen,
3. alle Brücken,
4. alle Blöcke (als Knotenmenge).

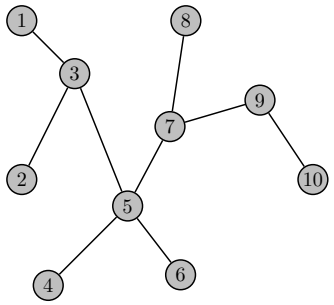


2.6 Bäume

Bäume modellieren hierarchische Strukturen und eindeutige Verzweigungen: von Dateisystemen und Organisationsdiagrammen über Such- und Indexstrukturen (binäre Suchbäume, Heaps, B-Bäume) bis zu Syntax- und Entscheidungsbäumen. Ihre charakteristischen Eigenschaften (genau ein einfacher Weg zwischen zwei Knoten, $n-1$ Kanten, jede Kante eine Brücke) machen sie zu einer zentralen Grundlage für viele Algorithmen.

Definition 2.43 (Baum)**:** Ein zusammenhängender, kreisfreier ungerichteter Graph $T = (V, E)$ heißt *Baum*. Ein nicht notwendigerweise zusammenhängender kreisfreier Graph heißt *Wald*.

Beispiel 2.24 (Baum)**:**



Satz 2.1 (Charakterisierung von Bäumen)**:** *Für einen Graphen T mit $n = |V| \geq 1$ sind folgende Aussagen äquivalent:*

1. *T ist ein Baum.*
2. *T ist zusammenhängend und kreisfrei.*
3. *T ist kreisfrei und besitzt genau $n-1$ Kanten.*
4. *T ist zusammenhängend und besitzt genau $n-1$ Kanten.*
5. *Zwischen je zwei verschiedenen Knoten von T existiert genau ein Weg.*
6. *Jede Kante von T ist eine Brücke.*

Bemerkung 2.11: Die Äquivalenzen erlauben unterschiedliche Beweisstrategien: je nach Situation ist es einfacher, Kreisfreiheit oder die Kantenanzahl nachzuweisen.

Definition 2.44 (Blatt, innerer Knoten, Grad)**:** Ein Knoten Grad 1 in einem Baum heißt *Blatt*. Knoten, die keine Blätter sind, heißen *innere Knoten*. Der maximale Knotengrad eines Baumes heißt (maximaler) *Grad des Baumes*.

Bemerkung 2.12: Jeder Baum mit mindestens zwei Knoten hat mindestens zwei Blätter.

Definition 2.45 (Ast)**:** Die Kanten eines Baumes werden oft als *Äste* bezeichnet.

2.6.1 Wurzelbäume und Arboreszenzen

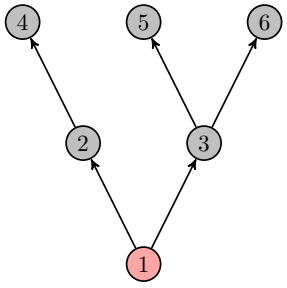
Definition 2.46 (Wurzelbaum / Arboreszenz)**:** Eine *Arboreszenz* (gerichteter Wurzelbaum) ist ein gerichteter Graph $A = (V, E)$, der

1. schwach zusammenhängend ist (der zugrunde liegende ungerichtete Graph ist zusammenhängend),
2. genau einen Knoten w mit Eingangsgrad $d^-(w) = 0$ besitzt (*Wurzel*),
3. und für alle $v \neq w$ gilt $d^-(v) = 1$.

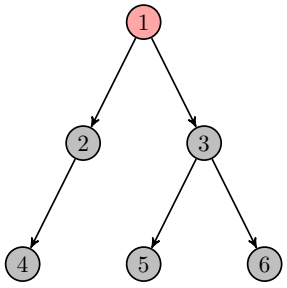
Der zugrunde liegende ungerichtete Graph ist dann ein Baum. Man spricht auch von einem *Wurzelbaum* (ungerichtet), wenn ein Knoten eines Baumes ausgezeichnet wird.

Bemerkung 2.13: In einer Arboreszenz sind alle Kanten von der Wurzel weg orientiert; es existiert genau ein gerichteter Weg von der Wurzel zu jedem anderen Knoten.

Beispiel 2.25 (Historische und heutige Darstellung eines Wurzelbaums)**:** Früher wurde die Wurzel unten gezeichnet (links), heute üblicherweise oben (rechts).



Alte Darstellung



Heutige Darstellung

2.6.2 Binärbäume

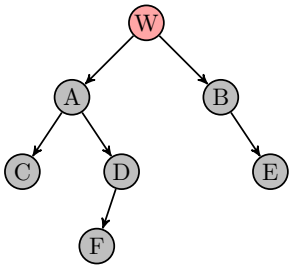
Ein *Binärbaum* ist ein (gerichteter) Wurzelbaum, in dem jeder Knoten höchstens zwei Kindknoten besitzt. Manchmal wird eine explizite Unterscheidung zwischen linkem und rechtem Kind vorgenommen. Wir unterscheiden:

- **Voller Binärbaum** (auch: *strikt* oder *saturiert* gennant): Jeder innere Knoten besitzt *genau* zwei Kinder. (Ein Blatt hat keine Kinder.)
- **Vollständiger Binärbaum**: Alle Ebenen sind gefüllt, außer evtl. die letzte, die von links nach rechts gefüllt wird.
- **Perfekter Binärbaum**: Ist sowohl voll als auch vollständig; dann besitzt die letzte Ebene genau 2^h Blätter.

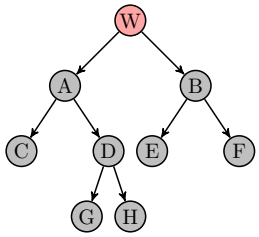
Bemerkung 2.14: Literaturhinweis: Manche Quellen verwenden die Begriffe *voll* und *vollständig* synonym oder vertauscht. Hier gilt: "voll"⇒ jeder innere Knoten hat zwei Kinder; "vollständig"⇒ linkslückenlose Füllung der Ebenen.

Beispiel 2.26 (Beispiele)**:**

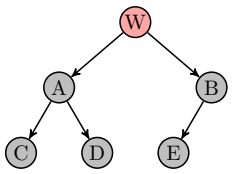
Allgemeiner Binärbaum



Voll, aber nicht vollständig



Vollständig, aber nicht voll



Links: jeder innere Knoten hat zwei Kinder (voll); mittig: volle Struktur, aber letzte Ebene nicht lückenlos (nicht vollständig); rechts: Ebenen füllen sich lückenlos von links, aber Knoten B hat nur ein Kind (nicht voll).

Definition 2.47 (Tiefe und Höhe)**:** Sei ein Wurzelbaum mit Wurzel w gegeben.

- Die *Tiefe* (auch: Ebene) eines Knotens v ist die Länge (Anzahl Kanten) des eindeutigen Weges von w zu v. Es gilt also: Tiefe$(w) = 0$, Kinder von w haben Tiefe 1 usw.
- Die *Höhe* eines Knotens v ist die Länge eines längsten Weges von v zu einem Blatt (Anzahl Kanten). Ein Blatt hat Höhe 0.

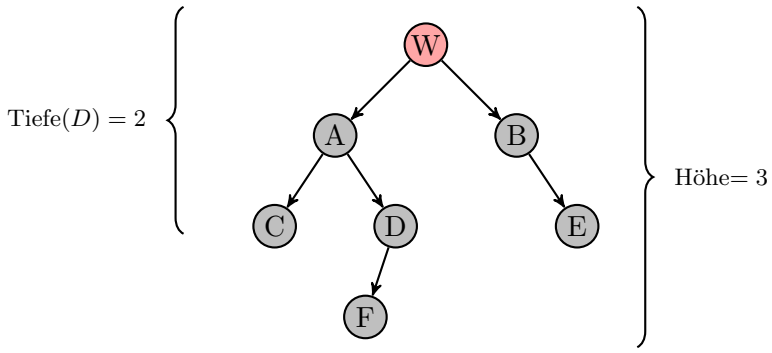


Abbildung 2.12: Höhe und Tiefe in einem Baum.

- Die *Höhe des Baumes* ist die Höhe der Wurzel und zugleich die maximale Tiefe eines Knotens. Äquivalent: maximale Länge eines Weges von w zu einem Blatt.

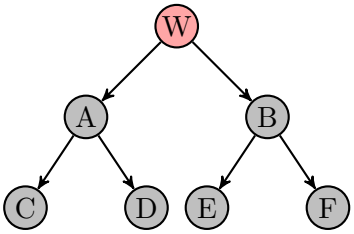
Hinweis: Manche Quellen zählen Knoten statt Kanten; dann erhöht sich jede der obigen Größen jeweils um 1. In diesem Skript wird *in Kanten* gezählt.

Satz 2.2 (Grundlegende Eigenschaften)**:** *Sei T ein Binärbaum mit Höhe h (Wurzel in Ebene 0):*

1. *In Ebene i stehen höchstens 2^i Knoten ($0 \leq i \leq h$).*
2. *$|V| \leq 2^{h+1} - 1$ mit Gleichheit genau für perfekte Binärbäume.*
3. *Ist T ein voller Binärbaum mit i inneren Knoten, so gilt: Anzahl der Blätter $= i + 1$ und insgesamt $|V| = 2i + 1$.*
4. *Für jeden vollständigen Binärbaum mit n Knoten gilt: $h = \lfloor \log_2 n \rfloor$ und $n \geq 2^h$.*

Ideenskizze. Die maximale Knotenanzahl pro Ebene ergibt sich aus Verdopplungspotential jedes Knotens. Summation der geometrischen Reihe liefert $2^{h+1} - 1$. Für volle Binärbäume erzeugt jedes Hinzufügen eines inneren Knotens exakt zwei neue Kanten zu Kindern; ein einfacher Induktionsbeweis liefert Blätter $= i + 1$. Die Höhe eines vollständigen Binärbaums ist durch die vollständig gefüllten Ebenen vorgegeben. □

Bemerkung 2.15 (Höhenabschätzungen)**:** Für jeden Binärbaum mit n Knoten gilt $h \geq \lceil \log_2(n+1) \rceil - 1$, mit Gleichheit genau für perfekte Binärbäume. Umgekehrt kann ein degenerierter (zu einer Kette entarteter) Binärbaum Höhe $n - 1$ haben.



Ebene 0: 1 Knoten = 2^0 Knoten

Ebene 1: 2 Knoten = 2^1 Knoten

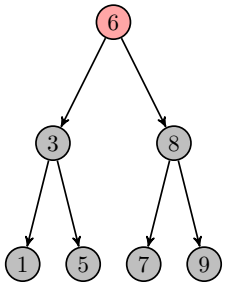
Ebene 2: 4 Knoten = 2^2 Knoten

Abbildung 2.13: Maximale Knotenanzahl pro Ebene in einem Binärbaum.

2.6.3 Suchbäume

Wir betrachten hier den Fall binärer Suchbäume, die eine effiziente Suche, Einfügen und Löschen von Elementen ermöglichen. Beim Einfügen von Elementen in den Suchbaum wird der Baum so erweitert, dass für jeden Knoten gilt: Alle Werte im linken Teilbaum sind kleiner, alle Werte im rechten Teilbaum sind größer oder gleich dem Wert des Knotens. Beim Einfügen in den Baum wird die Struktur zunächst nicht verändert. Im nächsten Abschnitt werden wir uns dann mit der Rebalancierung von Suchbäumen befassen.

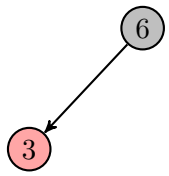
Achtung: in der Darstellung ist in den Knoten der *Inhalt* (also ein `int`-Wert) dargestellt (und nicht der Knotenname, bzw. Index)



- Ein perfekter Binärbaum mit Höhe h hat $2^h - 1$ Knoten
- Beispiel: Höhe 3 $\Rightarrow 2^3 - 1 = 7$ Knoten
- Umgekehrt hat ein perfekter Binärbaum mit n Knoten eine Höhe von $O(\text{ld } n)$

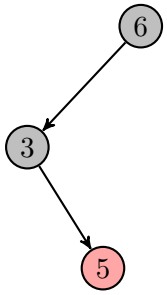
Beispiel 2.27: Wir betrachten den schrittweisen Aufbau eines binären Suchbaums durch die Einfügung der Elemente 6, 3, 5, 8, 1, 7, 9, 4.

(a) (1) Erstes Element 6 als Wurzel.

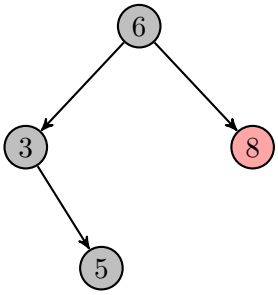


(b) (2) 3 als linkes Kind ($3 < 6$).

Abbildung 2.14: Aufbau des Suchbaums: Schritte 1–2.

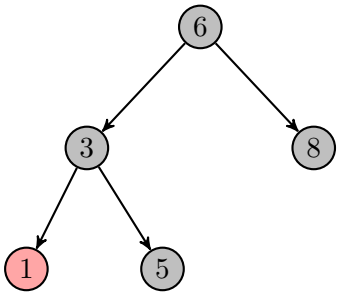


(a) (3) 5 in linken Teilbaum; rechts von 3.

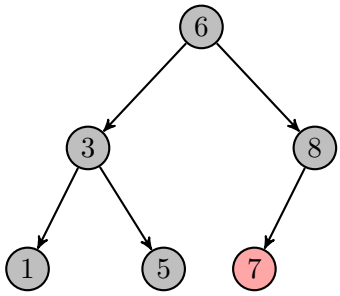


(b) (4) 8 als rechtes Kind der Wurzel.

Abbildung 2.15: Aufbau des Suchbaums: Schritte 3–4.

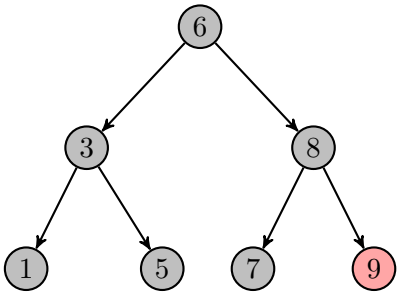


(a) (5) 1 als linkes Kind von 3.

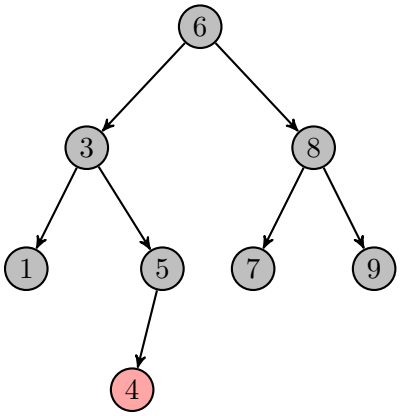


(b) (6) 7 als linkes Kind von 8.

Abbildung 2.16: Aufbau des Suchbaums: Schritte 5–6.



(a) (7) 9 als rechtes Kind von 8.



(b) (8) 4 als linkes Kind von 5.

Abbildung 2.17: Aufbau des Suchbaums: Schritte 7–8.

2.6.4 Balancierte Suchbäume

Balancierte Suchbäume sind spezielle Baumstrukturen, die eine effiziente Suche, Einfügen und Löschen von Elementen gewährleisten, indem sie die Höhe des Baumes minimieren. Zu den bekanntesten balancierten Suchbäumen gehören AVL-Bäume und Rot-Schwarz-Bäume.

- **AVL-Bäume:** Diese Bäume sind nach ihrem Erfinder Georgy Adelson-Velsky und Evgenii Landis benannt. Sie garantieren, dass für jeden Knoten die Höhen der linken und rechten Teilbäume sich um höchstens 1 unterscheiden. Dies wird durch Rotationen beim Einfügen und Löschen von Knoten sichergestellt.
- **Rot-Schwarz-Bäume:** Diese Bäume sind eine spezielle Art von binären Suchbäumen, die zusätzlich zu den Suchbaum-Eigenschaften auch Farbregeln einhalten. Jeder Knoten ist entweder rot oder schwarz, und es gelten bestimmte Regeln, die sicherstellen, dass der Baum balanciert bleibt.

Wir betrachten in weitere Folge die AVL-Bäume genauer:

- **Suchoperation:** Die Suche in einem AVL-Baum erfolgt ähnlich wie in einem normalen binären Suchbaum. Der Unterschied liegt jedoch in der zusätzlichen Bedingung, dass der Baum nach jeder Einfüge- oder Löschoperation balanciert bleibt.
- **Einfügeoperation:** Beim Einfügen eines neuen Knotens wird zunächst die normale Einfügeoperation eines binären Suchbaums durchgeführt. Anschließend wird überprüft, ob der Baum aus dem Gleichgewicht geraten ist. Falls ja, werden Rotationen durchgeführt, um den Baum wieder ins Gleichgewicht zu bringen.
- **Löschoperation:** Die Löschung eines Knotens erfolgt ebenfalls wie in einem normalen binären Suchbaum. Nach der Löschung wird der Baum auf mögliche Ungleichgewichte überprüft und gegebenenfalls rotiert.

Balance-Faktor und Invariante Für jeden Knoten v sei h_L die Höhe seines linken und h_R die Höhe seines rechten Teilbaums (-1 für leeren Baum). Der *Balance-Faktor* ist

$$bf(v) = h_L - h_R.$$

Ein binärer Suchbaum ist ein *AVL-Baum*, wenn für alle Knoten $|bf(v)| \leq 1$ gilt. Dadurch bleibt die Höhe bei n Knoten stets $O(\log n)$;

Rotationen Verletzt nach Einfügen oder Löschen ein Knoten z (der erste auf dem Rückweg zur Wurzel mit $|bf| > 1$) die Bedingung, betrachten wir dessen schwereren Kindpfad und unterscheiden vier Fälle. X bezeichnet jeweils den neu eingefügten (oder „ursächlich verschobenen") Knoten.

Fälle (symmetrisch):

- LL: Einfügen im *linken* Teilbaum des *linken* Kindes
  $\Rightarrow$ einfache Rechtsrotation.
- RR: Einfügen im rechten Teilbaum des rechten Kindes
  $\Rightarrow$ einfache Linksrotation.
- LR: Einfügen im *rechten* Teilbaum des linken Kindes
  $\Rightarrow$ Doppelrotation: Linksrotation am linken Kind, dann Rechtsrotation.
- RL: Einfügen im *linken* Teilbaum des rechten Kindes
  $\Rightarrow$ Doppelrotation: Rechtsrotation am rechten Kind, dann Linksrotation.

Strukturschemata Wir verwenden $T_1, \ldots, T_4$ für (unveränderte) Teilbäume deren Höhenrelation die Suchbaumeigenschaften respektiert.

LL-Fall (einfache Rechtsrotation)

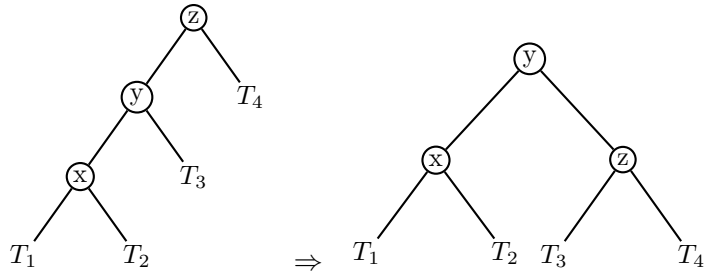


Abbildung 2.18: LL-Ungleichgewicht ($bf(z) = +2, bf(y) = +1/0$) wird durch Rechtsrotation um z behoben. In einem ausbalancierten Endzustand kann die Höhe von T_3 (linker Teilbaum von z) größer oder gleich der von T_2 sein; dies ist durch die leicht höhere Platzierung angedeutet. Die Inorder-Reihenfolge $T_1 < x < T_2 < y < T_3 < z < T_4$ bleibt erhalten.

RR-Fall (einfache Linksrotation)

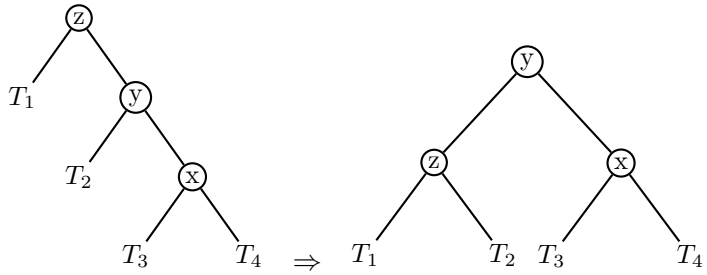


Abbildung 2.19: RR-Ungleichgewicht ($bf(z) = -2, bf(y) = -1/0$) wird durch Linksrotation um z behoben. Analog zum LL-Fall kann im Endzustand $h(T_2) \geq h(T_3)$ sein. Die Inorder-Reihenfolge $T_1 < z < T_2 < y < T_3 < x < T_4$ bleibt erhalten.

LR-Fall (Doppelrotation: Links dann Rechts)

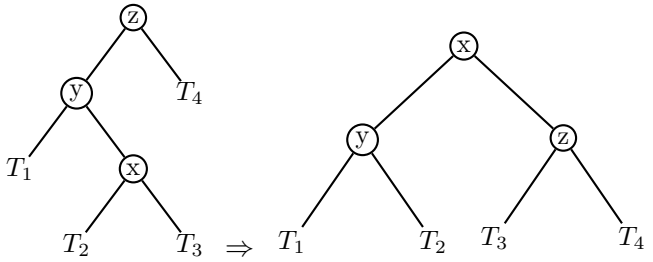


Abbildung 2.20: LR-Ungleichgewicht: Doppelrotation (Linksrotation am linken Kind, dann Rechtsrotation).

RL-Fall (Doppelrotation: Rechts dann Links)

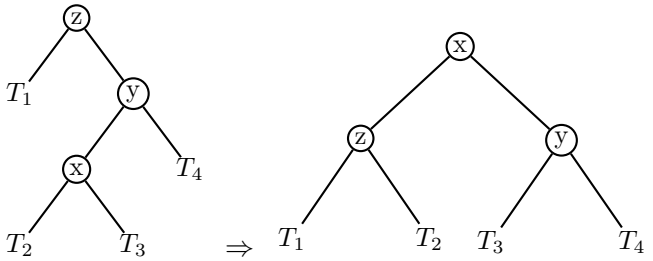


Abbildung 2.21: RL-Ungleichgewicht: Doppelrotation (Rechtsrotation am rechten Kind, dann Linksrotation).

3 Rekursive Algorithmen

Rekursive Definitionen begegnen uns in der Mathematik ständig: die natürlichen Zahlen über den Nachfolger, Listen als „leeres Objekt oder Kopf plus (kürzere) Liste“, Bäume als Wurzel mit (kleineren) Teilbäumen. Rekursion übersetzt dieses induktive Beschreibungsprinzip direkt in Programmcode. Anstatt eine Lösung Schritt für Schritt iterativ aufzubauen, formulieren wir sie als Kombination (oder Auswahl) von Lösungen kleinerer Teilprobleme.

Typische Vorteile sind Kürze und strukturelle Klarheit; typische Gefahren sind fehlende oder falsche Basisfälle sowie exponentielle Laufzeiten durch mehrfaches Berechnen identischer Teilinstanzen (wie beim naiven Fibonacci-Beispiel). Die Ausführung einer Rekursion wird vom *Call-Stack* des Laufzeitsystems getragen: Jeder Aufruf erzeugt einen neuen Aktivierungseintrag mit lokalen Variablen; beim Rücksprung werden diese automatisch entsorgt. Dieses Kapitel zeigt zentrale Muster (einfache Rekursion, Divide & Conquer, Backtracking) und Methoden zur Laufzeitanalyse rekursiver Relationen.

3.1 Grundidee und Definition

Definition 3.1 (Rekursion)**:** Eine (Unter-)Funktion / Prozedur heißt *rekursiv*, wenn sie sich (direkt oder indirekt) selbst aufruft.

Ein korrekter rekursiver Algorithmus besteht aus

- einem oder mehreren *Basisfällen* (Abbruchbedingungen),
- einem *Rekursionsschritt*, der das Problem auf kleinere (“einfachere”) Teilprobleme reduziert,
- der Gewährleistung, dass die Teilprobleme die Basisfälle erreichen (*Terminierung*).

Typisches Beweisschema für Korrektheit: vollständige Induktion über die Instanzgröße.

3.1.1 Beispiel: Fakultät

Für $n \geq 1$ sei $n! = n \cdot (n-1)!$ und $0! = 1$.

Algorithmus 4: Rekursive Fakultät

Input: $n \in \mathbb{N}_0$

Output: $n!$

Function Factorial*(n)*

```
if n = 0 then
    return 1
return n · Factorial(n − 1)
```

Beachte Eingabekontrolle: Aufrufe mit $n < 0$ müssen ausgeschlossen werden (sonst Endlosrekursion).

3.2 Negativbeispiel: Naive Fibonacci-Rekursion

Die Fibonacci-Folge ist definiert durch $F_0 = 1$, $F_1 = 1$ und $F_n = F_{n-1} + F_{n-2}$ für $n \geq 2$.

Algorithmus 5: Naive Fibonacci

Input: $n \in \mathbb{N}_0$

Output: F_n

Function Fib*(n)*

```
if n ≤ 1 then
    return 1
return Fib(n − 1) + Fib(n − 2)
```

Die Anzahl der Aufrufe erfüllt $T(n) = T(n-1) + T(n-2) + 1$ mit $T(0) = T(1) = 1$ und wächst exponentiell ($T(n) = \Theta(\varphi^n)$, $\varphi = \frac{1+\sqrt{5}}{2}$). Problem: Massive Doppelberechnungen (Überlappung von Teilproblemen).

Beispiel 3.1 (Verbesserung durch Memoisation)**:** Speichern aller bereits berechneten Werte reduziert die Laufzeit auf $O(n)$ und Speicher auf $O(n)$.

Übung 3.1: Geben Sie eine iterative $O(n)$-Variante an, welche nur konstante Zusatzspeicher benutzt.

3.3 Binäre Suche

Obwohl die binäre Suche auch iterativ (also mit Schleifen) umgesetzt werden kann, ist sie ein anschauliches Beispiel für einen rekursiven Algorithmus. Gegeben ein sortiertes

Array $A[0..n-1]$ in dem ein Element x gesucht wird. Der naive Zugang würde des Array von der ersten Stelle weg durchlaufen, und beenden sobald x gefunden wird. Dies ergibt eine Laufzeit von $O(n)$, im Erwartungsfall werden $n/2$ Elemente durchlaufen. Die binäre Suche läuft in $O(\log n)$ ab.

Die Grundidee ist in etwa wie das Suchen eines Namen in einem Telefonbuch. Man schlägt in der Mitte auf und prüft ob der gesuchte Name vor, nach oder an der aktuellen Position steht. Steht der davor oder danach, so wird die Suche in dem verbleibenden Teil genau nach dem selben Prinzip fortgesetzt. Man bestimmt in etwa die Mitte, und prüft das Element an dieser Stelle, usw.

Algorithmus 6: Rekursive Binäre Suche

Input: Array A, linke und rechte Grenze l und r, gesuchtes Element x

Function BinSuche(A, l, r, x)

```
if l > r then
    return NotFound
m = l + ⌊(r − l)/2⌋
if A[m] = x then
    return m
else if A[m] < x then
    return BinSuche(A, m + 1, r, x)
else
    return BinSuche(A, l, m − 1, x)
```

Der Ablauf der Suche kann als Binärbaum verstanden werden. Ein Beispiel wird in Abbildung 3.1 angegeben.

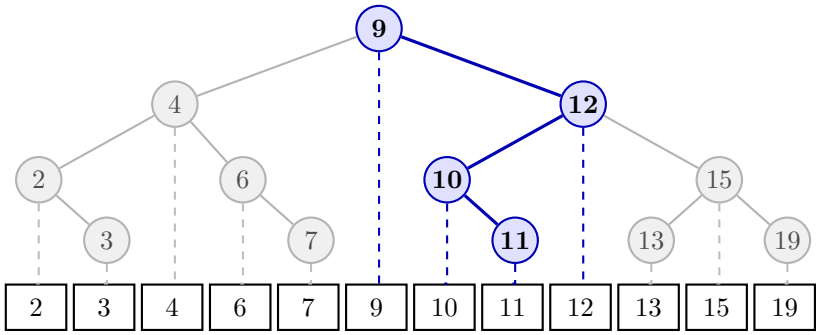


Abbildung 3.1: Binäre Suche nach 11 im Array $[2, 3, 4, 6, 7, 9, 10, 11, 12, 13, 15, 19]$. Im Wurzelknoten wird nach rechts gegangen, da $11 > 9$. Die Mitte der rechten Teilfolge enthält das Element 12. Davon muss nach links weiter gegangen werden, u.s.w.

Recurrence: $T(n) = T(n/2) + O(1) \Rightarrow T(n) = O(\log n)$. Rekursionshöhe $= \lfloor \log_2 n \rfloor$.

Ein perfekter Binärbaum mit n Blättern hat die Tiefe $\mathrm{ld}(n)$, wobei ld der *Logarithmus dualis*, also der Logarithmus zur Basis zwei ist.[1]

Beispiel 3.2: Suche in einem Telefonbuch Der beschriebene Suchalgorithmus ist durchaus vergleichbar mit der Suche in einem Telefonverzeichnis einer Stadt (Telefonbuch). Bei angenommener einer Million Einträge, liefert der Logarithmus dualis den Wert 19.93..., was sich durchaus gut mit der empirischen Erfahrung bezüglich der tatsächlich benötigten Suchschritte deckt.

3.3.1 Berechnung aller Permutationen

Backtracking erweitert schrittweise eine partielle Lösung und nimmt letzte Entscheidungen zurück, sobald kein Fortschritt möglich ist. Für Permutationen: an Position k tauschen wir jedes verbleibende Element dorthin, rekursiv weiter für $k+1$, danach Rücktausch (Zustand wiederherstellen). Dadurch werden alle $n!$ Permutationen genau einmal erzeugt.

Kernkosten: Jede Ausgabe einer Permutation benötigt $O(n)$ Zeit (Ausgabe / Kopie). Untere Schranke $\Omega(n \cdot n!)$; der Algorithmus erreicht $O(n \cdot n!)$ und verwendet nur $O(1)$ zusätzlichen Speicher außerhalb des Rekursions-Stacks.

Algorithmus 7: Permutationen durch Vertauschen

```
Input: Array A[0..n − 1]
Function Permute(k)
    if k = n − 1 then
        Verarbeitung/Ausgabe A
        return
    for i = k to n − 1 do
        vertausche A[k] und A[i]
        Permute(k + 1)
        vertausche A[k] und A[i]
        //Backtrack
Permute(0)
```

Mit der Anzahl $n!$ an Permutationen und $O(n)$ Arbeitschritten pro Permutation ergibt sich als Gesamtlaufzeit $O(n \cdot n!)$.

[1] $\mathrm{ld}(n) = \log_2(n) = \frac{\ln(n)}{\ln(2)}$

3.4 Rekursive Tiefensuche (DFS)

Zum tiefergehenden Verständnis dieses Abschnittes sei auf das Kapitel 6.1 verwiesen. Aufgrund der möglichen rekursiven Implementierung der dort behandelten Tiefensuche (DFS) wird der Algorithmus aber auch schon an dieser Stelle vorgestellt.

Die Grundidee der Tiefensuche ist, ausgehend von einem bestimmten Knoten, über *Pfade* möglichst weit in die Tiefe zu gehen. Von einem Knoten ausgehend wird geprüft, ob es unbesuchte Nachbarknoten gibt; ist dies der Fall, wird die Suche mit diesem Knoten fortgesetzt. Wenn keine unbesuchten Nachbarknoten vorhanden sind, wird soweit zurück gegangen, bis unbesuchte adjazente Nachbarn gefunden werden (und dann dort fortgesetzt).

Das beschriebene Vorgehen lässt sich sehr prägnant mittels Rekursion beschreiben:

Algorithmus 8: Rekursive Tiefensuche

Input: $G = (V, E), v \in V$

Function DFS*(G, v)*

```
markiere v als besucht
for alle u mit {v, u} ∈ E do
    if u nicht besucht then
        DFS(G, u)
```

Möchte man tatsächlich einen konkreten Knoten s mit der Tiefensuche finden, so kann man die Funktion entsprechend anpassen:

Algorithmus 9: Tiefensuche zur Suche eines Knoten s

Input: $G = (V, E)$, $v \in V$ (aktueller Knoten), s (gesuchter Knoten)

Output: s falls gefunden, sonst NULL

```
Function DFS(G, v, s)
    if v = s then
        return v
    markiere v als besucht;
    for alle u mit {v, u} ∈ E do
        if u nicht besucht then
            if DFS(G, u, s) == s then
                return s
    return NULL
```

Die Laufzeitkomplexität beträgt $O(|V| + |E|)$. Dies wird dadurch anschaulich, dass jeder Knoten und jede Kante höchstens einmal verarbeitet werden.

4 Sortierverfahren

Sortieralgorithmen sind Grundbausteine vieler Anwendungen (Suche, Datenkompression, Datenbanken, Graphalgorithmen). Wichtige Vergleichsgrößen:

- **Laufzeit**: Best / Average / Worst Case.
- **Speicher**: In-place (nur O(1) zusätzlicher Platz) vs. zusätzlicher Speicher.
- **Stabilität**: Gleiche Schlüssel behalten ihre Reihenfolge.
- **Adaptivität**: Nutzt teilweise vorhandene Ordnung aus ("fast sortiert").
- **Deterministisch vs. Randomisiert**. Randomisierung reduziert die Wahrscheinlichkeit eines schlechten Worst-Case (schlechter Pivotwahl).

Unter dem Modell *vergleichsbasiertes Sortieren* (nur Schlüsselvergleiche) gilt die Informationstheoretische Untergrenze $\Omega(n \log n)$ für das Sortieren n unterschiedlicher Elemente. Verfahren wie Counting/Radix/Bucket Sort umgehen dies nur mit zusätzlichen Annahmen über den Schlüsselraum.

4.1 Einfache Sortieralgorithmen

Der Vollständigkeit halber sind in diesem Abschnitt nochmals die drei grundlegenden Sortieralgorithmen Bubblesort, Insertionsort und Selectionsort dargestellt. Für die Praxis sind diese Algorithmen aufgrund ihres Laufzeitverhaltens von $O(n^2)$ jedoch oft nicht effizient genug und werden deshalb nur aus didaktischen Gründen vorgestellt.

Algorithmus 10: Bubblesort

```
Input: Array a
Result: Sortiertes Array a
Function BubbleSort(a)
    for i = 0; i < a.length − 1; i++ do
        swapped = false
        for j = 0; j < a.length − 1 − i; j++ do
            if a[j] > a[j + 1] then
                Swap(a, j, j+1)
                swapped = true
        if ¬swapped then
            break
```

Algorithmus 11: Insertionsort

```
Input: Array a
Result: Sortiertes Array a
Function InsertionSort(a)
    i = 1
    while i < a.length do
        j = i
        while j > 0 ∧ a[j − 1] > a[j] do
            Swap(a, j, j-1)
            j = j − 1
        i = i + 1
```

Algorithmus 12: Selectionsort

```
Input: Array a
Result: Sortiertes Array a
Function SelectionSort(a)
    for i = 0; i < a.length − 1; i++ do
        jMin = i
        for j = i + 1; j < a.length; j++ do
            if a[j] < a[jMin] then
                jMin = j
        if jMin ≠ i then
            Swap(a, i, jMin)
```

4.2 Mergesort

Mergesort teilt das Array rekursiv in zwei ungefähr gleich große Hälften, sortiert diese Teilarrays und *verschmilzt* (merged) sie anschließend zu einem vollständig sortierten Array. Das Zusammenführen zweier bereits sortierter Listen der Gesamtlänge n benötigt nur einen linearen Durchlauf, indem jeweils das kleinere der beiden aktuellen Elemente übernommen wird. Dadurch entsteht die Rekurrenz

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

(Höhe des Rekursionsbaums $= \log_2 n$, pro Ebene insgesamt $O(n)$ Merge-Arbeit). Speicher: zusätzliche temporäre Liste der Größe n.

Algorithmus 13: Mergesort

Input: Array $A[l..r]$

Function Mergesort*(A, l, r)*

```
if l = r then
    return [A[l]]
m = l + ⌊(r − l)/2⌋
L = Mergesort(A, l, m)
R = Mergesort(A, m + 1, r)
return Merge(L, R)
```

Eigenschaften.

- Laufzeit immer $\Theta(n \log n)$ (keine Abhängigkeit von Schlüsselverteilung).
- Stabil (wegen $\leq$ beim Vergleich).
- Nicht in-place in obiger Standardform (benötigt zusätzlichen Speicher für Hilfsarray). Es existieren kompliziertere In-place-Varianten mit höherem konstanten Aufwand.
- Sehr gut parallelisierbar (unabhängige Rekursionen + paralleles Merge mittels Block-Techniken).
- Cache-Verhalten: Lineare Merges sind sequentiell (lokal), Rekursion sorgt aber für Sprünge. Iterative Bottom-Up-Variante oft cache-freundlicher.

Algorithmus 14: Merge zweier sortierter Listen (stabil)

Input: sortierte Arrays $L[0..a-1]$, $R[0..b-1]$
Output: zusammengefügtes sortiertes Array $S[0..a+b-1]$

```
Function Merge(L, R)
    leftPos = 0
    rightPos = 0
    targetPos = 0
    while leftPos < a and rightPos < b do
        if L[leftPos] ≤ R[rightPos] then
            S[targetPos] = L[leftPos]
            leftPos = leftPos + 1
        else
            S[targetPos] = R[rightPos]
            rightPos = rightPos + 1
        targetPos = targetPos + 1
    while leftPos < a do
        S[targetPos] = L[leftPos]
        leftPos = leftPos + 1
        targetPos = targetPos + 1
    while rightPos < b do
        S[targetPos] = R[rightPos]
        rightPos = rightPos + 1
        targetPos = targetPos + 1
    return S
```

Beispiel 4.1: Mergesort Folge 37, 12, 45, 2, 18, 25, 7, 30, 50, 1, 19, 5

```
| 37   12   45   2   18   25   7   30   50   1   19   5 |
| 37   12   45   2   18   25 | 7   30   50   1   19   5 |
| 37   12   45 | 2   18   25 | 7   30   50 | 1   19   5 |
| 37 | 12   45 | 2 | 18   25 | 7 | 30   50 | 1 | 19   5 |
| 37 | 12 | 45 | 2 | 18 | 25 | 7 | 30 | 50 | 1 | 19 | 5 |

| 37 | 12   45 | 2 | 18   25 | 7 | 30   50 | 1 | 5   19 |
| 12   37   45 | 2   18   25 | 7   30   50 | 1   5   19 |
|  2   12   18   25   37   45 | 1   5   7   19   30   50 |
|  1   2   5   7   12   18   19   25   30   37   45   50 |
```

4.3 Quicksort

Quicksort partitioniert das Array in (typisch) zwei Teile: Elemente kleiner als der Pivot und Elemente größer/gleich Pivot, und sortiert die Teile rekursiv.

Typische Partitionierungsvarianten sind (1) Hoare-Schema (zwei Zeiger von außen nach innen), (2) Lomuto-Schema (ein Zeiger, stabilerer Code, aber mehr Swaps), (3) Dreifach-Partitionierung ("Dutch National Flag") für viele Duplikate.

Pivotwahl-Strategien.

- Zufälliger Pivot (Randomized Quicksort) reduziert Wahrscheinlichkeit eines schlechten Splits drastisch.
- Median-of-3 (erstes, mittleres, letztes Element) glättet typische reale Muster.
- Median-of-medians garantiert lineare Pivotfindung $O(n)$ mit deterministischem $O(n \log n)$ Worst-Case, hat aber hohe Konstanten.
- Introsort: Startet als Quicksort; bei zu großer Rekursionstiefe Wechsel auf Heapsort (C++ STL). Garantiert Worst-Case $O(n \log n)$ bei typischer Quicksort-Schnelligkeit.

Eigenschaften.

- Average Case: $\Theta(n \log n)$ mit kleiner Konstante – oft schneller als Mergesort in der Praxis (cache-freundliche Partitionierung, in-place).
- Worst Case: $\Theta(n^2)$ ohne Schutzmaßnahmen.
- Speicher: In-place (nur Rekursions-Stack; Tiefe im Mittel $O(\log n)$, Worst-Case $O(n)$).
- Nicht stabil (Standardvarianten). Stabilität durch aufwändige Umbauten möglich, dann aber weniger in-place.
- Sehr empfindlich gegenüber vielen gleichen Schlüsseln – hier hilft Dreifach-Partitionierung.

Algorithmus 15: Quicksort

Input: Array $A[l..r]$

Function Quicksort*(A, l, r)*

```
if l ≥ r then
  return
p = Partition(A, l, r)
Quicksort(A, l, p − 1)
Quicksort(A, p + 1, r)
```

Partition teilt in Elemente $<$ Pivot und $\geq$ Pivot.

Algorithmus 16: Partition (letztes Element als Pivot)

```
Input: Array A[l..r]
Function Partition(A, l, r)
    pivot = A[r]
    i = l
    j = r − 1
    while i < j do
        while A[i] < pivot do
            i = i + 1
        while j > l and A[j] ≥ pivot do
            j = j − 1
        if i < j then
            Swap(A, i, j) i = i + 1
            j = j − 1
    if i = j and A[j] < pivot then
        i = i + 1
    if A[i] ≠ pivot then
        Swap(A, i, r)
    return i        // finale Pivot-Position
```

Bemerkung 4.1: Die erwartete Anzahl Vergleiche in Randomized Quicksort ist $2(n+1)H_n - 4n \approx 2n \ln n + O(n)$ (harmonische Zahl H_n). Damit liegt die Konstante vor dem $n \log n$ kleiner als bei Mergesort.

Beispiel 4.2: Bei bereits sortierter Eingabe und Pivotwahl „immer letztes Element" degeneriert Quicksort zum quadratischen Verhalten – ein einfaches Mischen (Randomisierung) verhindert dies mit hoher Wahrscheinlichkeit.

Beispiel 4.3: Quicksort Folge 3, 7, 1, 8, 2, 5, 9, 4, 6

Wir zeigen die Zustände

- bei der Wahl des Pivot-Elements (in runden Klammern)
- nach der Partitionierung

- nach endgültiger Platzierung des Pivot-Elements

Die Partitionierung ist durch die Pfeile gegeben.

```
|> 3   7    1    8    2    5    9    4   6 <|

|> 3   7    1    8    2    5    9    4  (6)<|
|> 3   4    1    8    2    5    9    7   6 <|
|> 3   4    1    5    2    8    9    7   6 <|
|> 3   4    1    5    2   [6]   9    7   8 <|

|> 3   4    1    5  (2)<|  6    9    7    8
|> 1   4    3    5   2 <|  6    9    7    8
|> 1  [2]   3    5   4 <|  6    9    7    8

  1    2  |> 3   5  (4)<|  6    9    7    8
  1    2  |> 3  [4]  5 <|  6    9    7    8

  1    2    3    4    5    6  |> 9   7  (8)<|
  1    2    3    4    5    6  |> 7   9   8 <|
  1    2    3    4    5    6  |> 7  [8]  9 <|
```

4.4 Heapsort

Heapsort basiert auf der Datenstruktur *Heap*, einem binären Baum, der die Heap-Eigenschaft erfüllt: Jeder Knoten ist kleiner/gleich (Min-Heap) bzw. größer/gleich (Max-Heap) seinen Kindern. Dadurch steht das Minimum/Maximum immer an der Wurzel. Abbildung 4.1 stellt einen Max-Heap mit 8 gespeicherten Elementen dar.

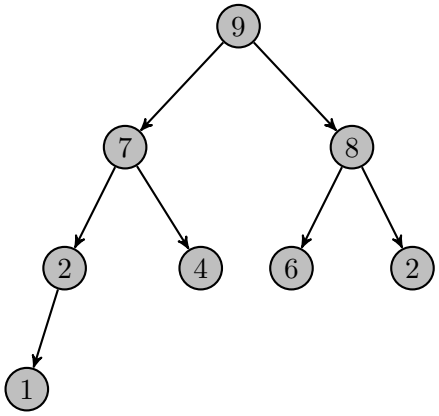


Abbildung 4.1: Max-Heap mit 8 Elementen: Die Werte in den Knoten stellen die gespeicherten Werte dar.

Mit der Annahme, dass die Elemente Ebene für Ebene in den binären Baum, eingefügt werden, muss man die Baumstruktur nicht explizit speichern, sondern kann die Elemente in einer einfachen Liste angeben. Der Max-Heap aus Abbildung 4.1 kann beispielsweise als Array $[9, 7, 8, 2, 4, 6, 2, 1]$ dargestellt werden. Für die algorithmische Betrachtung empfiehlt sich dennoch die Anschauung eines Binärbaumes. Wir betrachten in weiterer Folge Max-Heaps, da dieser zur aufsteigenden In-Place-Sortierung verwendet wird.

Der wesentliche Nutzen des Heaps ergibt sich aus der teilweisen Ordnung der Elemente. So kann insbesondere das Maximum (bei einem Max-Heap) in $O(1)$ gefunden werden!

Wird das Maximum entnommen, so muss der Heap reorganisiert werden.

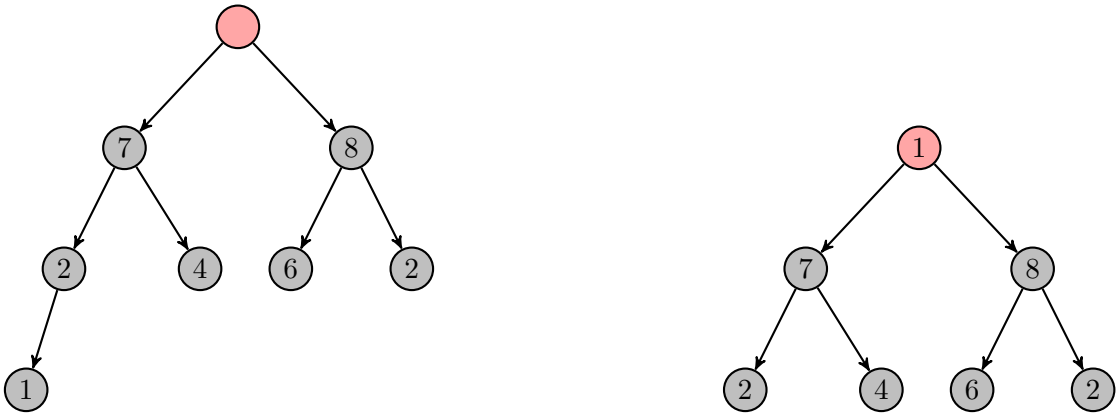


Abbildung 4.2: Entnahme des Maximums (Wurzelknoten) aus dem Heap (links) und Platzierung des letzten Elements in die Wurzel (rechts).

Die Reorganisation erfolgt durch das *Versickern* des neuen Wurzelelements, siehe Abbildung 4.3.

Abbildung 4.3: Versickern durch Vertauschung mit dem größten Kind.

Der initiale Aufbau eines Heaps aus einer unsortierten Liste von n Elementen kann in $O(n)$ Zeit erfolgen (ohne Beweis). Dabei wird der Baum von unten nach oben durchlaufen, und jedes Element wird an seine korrekte Position versickert. Die Blätter des Baumes müssen dabei nicht betrachtet werden, da diese bereits Heaps sind.

Der Algorithmus Heapsort baut im ersten Schritt einen gültigen Heap auf. Anschließend entnimmt er jeweils das maximale Element (aus der Wurzel, also der ersten Stelle), und platziert es am Ende des Arrays. Das bisherige Element der letzten Position wird dann in die Wurzel gesetzt. Nun wird der Heap um ein Element verkleinert, da sich an der letzten Position des Heaps im aktuellen Schritt schon das bereits sortierte Element befindet. Anschließend wird das Wurzel-Element versickert (heapify).

Beispiel 4.4: Wir demonstrieren den schrittweisen Aufbau eines Max-Heaps (Bottom-Up) aus der Folge [1, 9, 8, 7, 4, 6, 2, 2].

Arrayzustand	**Kommentar**
1, 9, 8, 7, 4, 6, 2, 2	Ausgangsfolge (unsortierte Anordnung)
	Heapify bei Index 3 (7 mit Kind 2): alles ok.
	Index 2 (8 mit Kindern 6 und 2): alles ok.
	Index 1 (9 mit Kindern 7 und 4): $9 \geq \max(7,4)$, bleibt.
9, 1, 8, 7, 4, 6, 2, 2	Index 0 (1 mit Kindern 9 und 8) $\rightarrow$ tausche mit 9
9, 7, 8, 1, 4, 6, 2, 2	Jetzt 1 an Index 1 (Kinder 7,4) $\rightarrow$ tausche mit 7
9, 7, 8, 2, 4, 6, 2, 1	Jetzt 1 an Index 3 (Kind 2) $\rightarrow 1 < 2 \rightarrow$ tausche
9, 7, 8, 2, 4, 6, 2, 1	Fertiger Max-Heap (Wurzel 9, Kinder kleiner gleich Eltern)

Tabelle 4.1: Schritte beim Bottom-Up-Aufbau eines Max-Heaps aus einer Permutation der Zahlen [9,7,8,2,4,6,2,1].

Beispiel 4.5: Wir demonstrieren den Heapsort Algorithmus anhand der Zahlenfolge $[9, 7, 8, 2, 4, 6, 2, 1]$

Schritt	**Arrayzustand**	**Kommentar**
1	1, 7, 8, 2, 4, 6, 2 ‖ 9	Vertausche Wurzel 9 mit letztem Heap-Element 1
	8, 7, 6, 2, 4, 1, 2 ‖ 9	Heapify bei Index 0: versickern von 1
2	2, 7, 6, 2, 4, 1 ‖ 8, 9	Vertausche Wurzel 8 mit letztem Heap-Element 2
	7, 4, 6, 2, 2, 1 ‖ 8, 9	Heapify bei Index 0: versickern von 2
3	1, 4, 6, 2, 2 ‖ 7, 8, 9	Vertausche Wurzel 7 mit letztem Heap-Element 1
	6, 4, 1, 2, 2 ‖ 7, 8, 9	Heapify bei Index 0: versickern von 1
4	2, 2, 1, 4 ‖ 6, 7, 8, 9	Vertausche Wurzel 6 mit letztem Heap-Element 2
	4, 2, 1, 2 ‖ 6, 7, 8, 9	Heapify bei Index 0: versickern von 2
5	2, 2, 1 ‖ 4, 6, 7, 8, 9	Vertausche Wurzel 4 mit letztem Heap-Element 2
	2, 2, 1 ‖ 4, 6, 7, 8, 9	Heapify bei Index 0: keine Veränderung nötig
6	1, 2 ‖ 2, 4, 6, 7, 8, 9	Vertausche Wurzel 2 mit letztem Heap-Element 1
	2, 1 ‖ 2, 4, 6, 7, 8, 9	Heapify bei Index 0: versickern von 1
7	1 ‖ 2, 2, 4, 6, 7, 8, 9	Vertausche Wurzel 2 mit letztem Heap-Element 1
	1 ‖ 2, 2, 4, 6, 7, 8, 9	Heapify nicht nötig
–	**1, 2, 2, 4, 6, 7, 8, 9**	Endergebnis: Array aufsteigend sortiert

Tabelle 4.2: Detaillierte Zwischenschritte von Heapsort mit der Eingabefolge [9, 7, 8, 2, 4, 6, 2, 1]

4.5 Laufzeitanalyse

Worst-Case Analyse einfacher Sortieralgorithmen:

Algorithmus	**Vergleiche**	**Swaps**
Bubblesort	$O(n^2)$	$O(n^2)$
Insertionsort	$O(n^2)$	$O(n^2)$
Selectionsort	$O(n^2)$	$O(n)$

Analyse weiterer Algorithmen:

Algorithmus	**Worst-Case**	**Average-Case**
Heapsort	$O(n \log n)$	$O(n \log n)$
Mergesort	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n^2)$	$O(n \log n)$

Quicksort:

- Worst-Case bei Quicksort wird nur selten erreicht
- In der Praxis schneller als Mergesort und Heapsort

Python

Python verwendet intern für `list.sort()` und `sorted()` einen Hybrid aus Mergesort und Insertionsort, genannt Timsort[a]. In `NumPy` wird allerdings standardmäßig Quicksort verwendet.

Anmerkung: bei Java kommt für primitive Datentypen ein Dual-Pivot-Quicksort zum Einsatz, für Objektarrays Timsort. Bei C++ wird in der Standard Template Library ein Hybrid aus Quicksort, Heapsort und Insertion Sort verwendet (Introsort).

[a] Benannt nach dem Erfinder Tim Peters.

5 Datenstrukturen

In Software wird mit unterschiedlichen Daten gearbeitet. Der einfachste Fall ist eine Variable, die einen Wert speichert. Weiters dienen Klassen als Vorlagen für Objekte, die strukturierte Daten (und gegebenenfalls Verhalten) beinhalten.

Oft möchte man aber mehrere Werte speichern, z.B. eine Liste von Namen, eine Menge von Zahlen, Schlüssel-Wert-Paare oder komplexere Datenstrukturen wie Bäume oder Graphen.

In diesem Kapitel betrachten wir die gängigsten dieser Datenstrukturen, und stellen ihre wesentlichen Eigenschaften vor. Auch wenn die Darstellung an der Programmiersprache Python orientiert ist, existieren diese oder sehr ähnliche Konzepte in nahezu allen Programmiersprachen.

- Arrays / geordnete Listen
- Sets (Mengen)
- Dictionaries / Maps
- Weitere Komponenten:
  - Stack
  - Queue

Weitere häufig verwendete Datenstrukturen sind Bäume, oder allgemein Graphen. Etwaige komplexere Datenstrukturen können aus diesen elementaren Datenstrukturen aufgebaut werden. Ein solides Verständnis für die Eigenschaften und Vor- und Nachteile der jeweiligen Datenstrukturen ist für die Programmierung und Softwareentwicklung essentiell.

5.1 Arrays

Ein Array ist eine Datenstruktur, die eine feste Anzahl von Elementen (des gleichen Typs) speichert. Die Elemente sind im Speicher zusammenhängend angeordnet, was einen schnellen Zugriff auf jedes Element über seinen Index ermöglicht. Arrays sind in vielen Programmiersprachen eine grundlegende Datenstruktur und werden häufig verwendet, um Sammlungen von Daten zu speichern.

Array mit 10 Integer-Elementen

7	3	12	5	9	1	4	0	0	0
0	1	2	3	4	5	6	7	8	9

Abbildung 5.1: Array mit 10 Elementen (Index unter den Zellen).

Um ein Element an einer bestimmten Stelle (Index) in ein Array einzufügen, müssen alle Elemente, die sich hinter dieser Stelle befinden, um eine Position nach hinten verschoben werden. Dies geschieht, um Platz für das neue Element zu schaffen. Es resultiert ein Aufwand von $O(n)$ im Worst Case, da die Elemente verschoben werden müssen.

Array mit 10 Integer-Elementen

7	3	12	5	7	9	14	4	0	0
0	1	2	3	4	5	6	7	8	9

Abbildung 5.2: Array nach dem Einfügen eines Elements 7 an der Stelle mit Index 4. Die Elemente 9, 14, und 4 wurden um eine Position nach rechts verschoben.

Auch beim Entfernen eines Elements an einer bestimmten Stelle (Index) in ein Array müssen alle Elemente, die sich hinter dieser Stelle befinden, um eine Position nach vorne verschoben werden. Dies geschieht, um die Lücke zu schließen, die durch das Entfernen des Elements entstanden ist. Es resultiert ebenfalls ein Aufwand von $O(n)$ im Worst Case, da die Elemente verschoben werden müssen.

Array mit 10 Integer-Elementen

7	3	5	7	9	14	4	0	0	0
0	1	2	3	4	5	6	7	8	9

Abbildung 5.3: Array nach dem Entfernen eines Elements an der Stelle mit Index 2. Die Elemente 12, 5, 7, 9, 14, 4, 0 und 0 wurden um eine Position nach links verschoben.

Die beschriebene Logik des Verschiebens der Elemente beim Einfügen und Löschen ist meist bei der elementaren Umsetzung von Arrays in Programmiersprachen nicht vorhanden. Die darauf aufbauenden höheren Datenstrukturen (in Java: `ArrayList`) setzen dieses Verhalten aber derart um.

Python

Interne Umsetzung von list in Python

- Intern ist eine `list` ein dynamisches Array (also im Speicher ein zusammenhängender Block).
- Beim Erweitern wird manchmal mehr Speicher reserviert als gerade benötigt, um häufige Kopieraktionen zu vermeiden.
- Indexzugriff O(1), aber das Einfügen in der Mitte O(n) (weil alle folgenden Elemente nach rechts verschoben werden müssen).

Operationen und ihre Komplexität

- lst.append(x) → Am Ende einfügen, amortisiert O(1).
- lst.insert(i, x) → An Position i einfügen, O(n), weil Elemente verschoben werden.
- lst[i] → Zugriff per Index, O(1).
- lst.remove(x) oder del lst[i] → ebenfalls O(n) im Worst Case (weil Verschieben nötig ist).

5.2 Verkettete Listen

Verkettete Listen setzen die Grundidee, eine größere Anzahl (an gleichartigen) Datensätzen abzuspeichern, auf eine andere Weise um. Anstatt die Daten in einem zusammenhängenden Block im Speicher zu speichern, werden die Datensätze (Knoten) einzeln im Speicher abgelegt und durch Verweise (Zeiger, Links) miteinander verbunden. Jeder Knoten enthält einen Verweis auf den nächsten Knoten in der Liste (bei doppelt verketteten Listen auch auf den vorherigen Knoten).

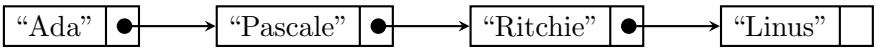


Abbildung 5.4: Einfach verkettete Liste mit vier Elementen.

Diese Herangehensweise bietet Vor- und Nachteile gegenüber Arrays. So ist das Einfügen und Löschen von Knoten in einer verketteten Liste effizienter, da nur die Verweise angepasst werden müssen, ohne dass andere Elemente verschoben werden müssen. Allerdings ist der Zugriff auf ein bestimmtes Element langsamer, da man die Liste von Anfang bis Ende durchlaufen muss.

Häufig kommen doppelt verkettete Listen zum Einsatz, bei denen jeder Knoten sowohl einen Verweis auf den nächsten als auch auf den vorherigen Knoten enthält. Dies ermöglicht eine effizientere Navigation in beide Richtungen.

Wir betrachten nun die Laufzeit der einzelnen Listen-Operationen:

- Zugriff auf ein Element per Index: $O(n)$ (weil die Liste durchlaufen werden muss)
- Einfügen an beliebiger Stelle: $O(n)$ (weil die Liste durchlaufen werden muss)
- Jedoch: Einfügen an beliebiger Stelle, wenn man an dieser Position steht: $O(1)$ (einfaches Anfügen)
- Löschen: $O(n)$ (weil die Liste durchsucht werden muss)
- Jedoch wieder $O(1)$ wenn man an der Position des zu löschenden Elements steht.

Ein typisches Anwendungsbeispiel für verkettete Listen ist die Implementierung von Warteschlangen oder Stacks, wo häufige Einfügungen und Löschungen am Anfang oder Ende der Liste erforderlich sind. In solchen Fällen bieten verkettete Listen klare Vorteile gegenüber Arrays, da sie eine konstante Zeit $O(1)$ für diese Operationen gewährleisten, während Arrays möglicherweise eine Neuordnung der Elemente erfordern.

Python

- Standardmäßig gibt es in Python keine verkettete Liste.
- Jedoch bietet die Bibliothek `collections.deque` eine Implementierung.
- `deque` ist eine doppelt verkettete Liste, optimiert für Einfügen/Entfernen am Anfang/Ende in $O(1)$.
- Aber: Kein effizienter Random Access (Indexzugriff ist $O(n)$).

5.3 Stack und Queue

5.3.1 Stack

Ein **Stack** (Kellerspeicher, Stapelspeicher) folgt dem Prinzip *Last In — First Out (LIFO)*: Das zuletzt eingefügte Element wird als erstes wieder entfernt. Typische Operationen sind:

- `push`: Einfügen eines Elements oben auf den Stack.
- `pop`: Entfernen des obersten Elements.
- `peek/top`: Zugriff auf das oberste Element, ohne es zu löschen.

Stacks lassen sich einfach mit Arrays oder verketteten Listen realisieren und werden z. B. bei Funktionsaufrufen (Call Stack) oder in Undo-Mechanismen eingesetzt.

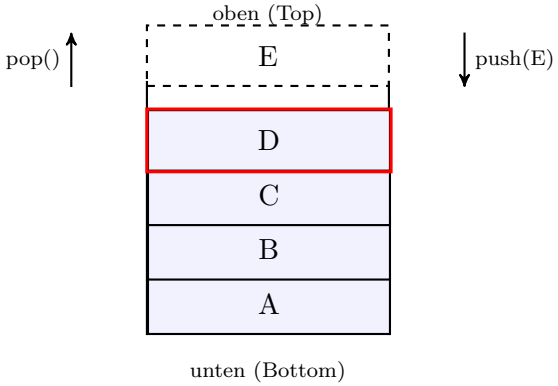


Abbildung 5.5: Schematischer Stack (LIFO): Nur das oberste Element (rot markiert) kann entfernt (pop) werden, bzw. kann auch nur oben hinzugefügt werden (push).

5.3.2 Queue

Eine **Queue** (Warteschlange) folgt dem Prinzip *First In – First Out (FIFO)*: Das zuerst eingefügte Element wird auch als erstes wieder entfernt. Typische Operationen sind:

- `enqueue`: Einfügen eines Elements am Ende der Queue.
- `dequeue`: Entfernen des ersten Elements.
- `front`: Zugriff auf das erste Element, ohne es zu löschen.

Queues werden oft in Betriebssystemen (Prozessplanung), Netzwerken (Datenpaketverwaltung) oder Druckersystemen verwendet. In Python eignet sich `collections.deque` besonders gut für die Implementierung.

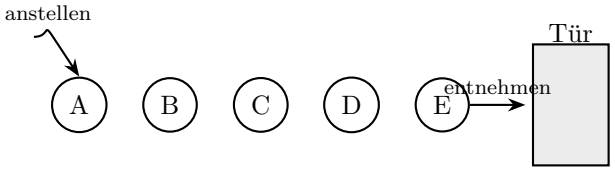


Abbildung 5.6: Schematische Warteschlange (Queue): vorne (rechts) wird entnommen, hinten (links) wird angestellt.

Python

- In Python lässt sich ein **Stack** einfach mit einer normalen Liste implementieren:
  - `append()` entspricht `push`.
  - `pop()` entfernt das oberste Element.
- Eine **Queue** lässt sich mit `collections.deque` effizient umsetzen:
  - `append()` fügt ein Element hinten an (`enqueue`).
  - `popleft()` entfernt das vorderste Element (`dequeue`).
- Vorteil von `deque`: Operationen am Anfang *und* Ende laufen in $O(1)$.

5.4 Dictionaries (Maps)

Dictionaries (oft auch Maps genannt) sind zentrale Datenstrukturen, die in allen Programmiersprachen existieren. Sie dienen dazu, Schlüssel-Wert-Paare (Key-Value-Pairs) zu speichern, wobei jeder Schlüssel eindeutig ist und auf einen bestimmten Wert verweist. Diese Struktur ermöglicht eine effiziente Zuordnung und den schnellen Zugriff auf Werte basierend auf ihren Schlüsseln. Man spricht von Dictionaries, weil man in der Regel eine Zuordnung (Mapping) zwischen Schlüsseln und Werten hat, und diese Schlüssel daher wie in einem Wörterbuch nachschlagen kann.

Beispiel:

Es werden Städten (=Schlüssel) die Mengen von Postleitzahlen (=Werte) zugeordnet. Der Wert ist hier also eine Menge (Datenstruktur Set).

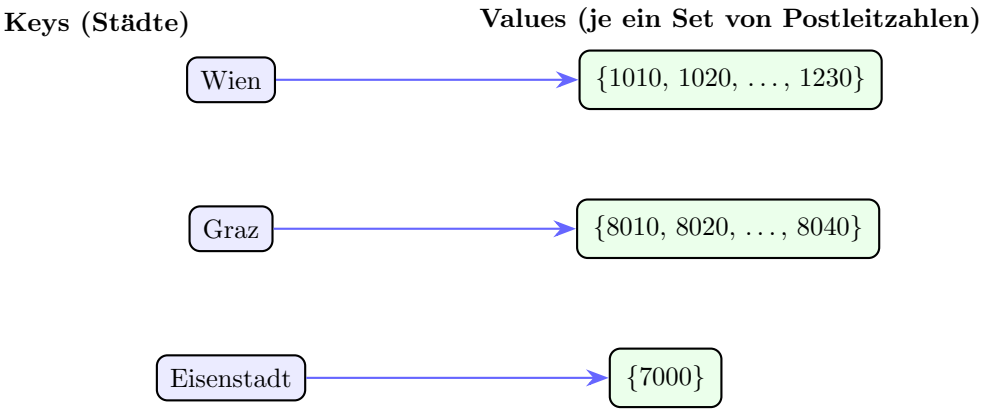


Abbildung 5.7: Dictionary-Mapping: Jeder Key (Stadt) zeigt mit genau einem Pfeil auf seinen Value, ein Set von Postleitzahlen.

Die Anforderung bei der Umsetzung einer derartigen Datenstruktur ist primär, sehr schnell (idealerweise $O(1)$) mittels Schlüssel auf die Werte zugreifen zu können. Dafür sind *Hashtabellen* ideal geeignet. Mittels *Hashfunktionen* wird zu einem Schlüssel ein Hashwert berechnet, der als Index in dieser Hashtabelle dient.

5.4.1 Hashfunktionen und Hashtabellen

Motivation Wir wollen Schlüssel (z.B. Strings) schnell auf Speicherplätze (Buckets) einer Tabelle abbilden. Dazu wird eine *Hashfunktion* h verwendet, die einen Schlüssel k auf einen ganzzahligen Index $h(k) \in \{0, \ldots, m-1\}$ (Tabellengröße m) projiziert.

Hashtabellen verbinden eine (große) Array-Struktur fester Länge m mit einer Hashfunktion. Jeder Eintrag (Bucket) kann (abhängig vom Verfahren) direkt ein Element oder eine kleine Sekundärstruktur (Liste/Baum) enthalten. Ziel: Erwartete $O(1)$ Zeit für Suchen, Einfügen und Löschen.

Anforderungen an eine Hashfunktion

- Deterministisch: Gleicher Schlüssel $\Rightarrow$ gleicher Hashwert.
- Effizient: Berechnung in (amortisiert) $O(1)$ und sehr schnell (wenige CPU-Operationen).
- Gleichmäßige Verteilung: Schlüssel sollen möglichst gleich über $\{0, \ldots, m-1\}$ verteilt werden (minimiert Kollisionen).
- Ähnliche Schlüssel $\Rightarrow$ ähnliche Hashwerte (Vermeidung von Clustern).
- Stabil bzgl. Lebensdauer der Tabelle (aber: in manchen Sprachen kann sich Hash zwischen Programmläufen absichtlich ändern, z.B. Python wegen Sicherheit).

Begriffe

- *Kollision*: Zwei verschiedene Schlüssel $k_1 \neq k_2$ mit $h(k_1) = h(k_2)$.
- *Lastfaktor* (load factor) $\alpha = \dfrac{n}{m}$ mit $n =$ Anzahl gespeicherter Elemente.
- Rehashing: Vergrößern der Tabelle (typisch Faktor 2) und Neuverteilen aller Elemente, sobald α einen Schwellwert überschreitet.

Typische Form (Zweiteilung)

$$h(k) = h_{\text{raw}}(k) \bmod m$$

Dabei produziert h_{raw} einen großen Integer (z.B. durch Kombination von Zeichen / Feldern), das Modulo faltet den Wert in den Tabellenbereich.

Einfache Beispiele

- Strings: Polynomielles Rolling-Hash $h_{\text{raw}}(s_0 s_1 \dots s_{L-1}) = \sum_{i=0}^{L-1} s_i \cdot B^{L-1-i}$ mit Basis B (z.B. 131, 257) und ggf. Reduktion mit großem Prim P vor Modulo m.
- Integer: Mischung (Bit-Mixing), z.B. Multiplikation mit großer ungerader Konstante und Bit-Shifts.

Kollisionsbehandlung (Überblick)

- *Getrennte Verkettung* (Chaining): Jeder Bucket hält eine (kurze) Liste / Baum der kollidierenden Elemente.
- *Offene Adressierung* (Sondieren): Bei Kollisionen wird versucht das Element an einer anderen Stelle im Array abzulegen (Sondieren).

Qualität der Hashfunktion

- Schlechte Verteilung führt zu langen Listen (Chaining) bzw. langen Suchfolgen (Sondieren) $\Rightarrow$ Performance-Verlust.
- Eine (annähernd) uniforme Verteilung hält erwartete Kosten pro Operation bei $O(1)$ solange α kontrolliert bleibt.

Immutabilität von Schlüsseln Ändert sich ein Schlüssel nach dem Einfügen (z.B. veränderbares Objekt mit hash-basiertem Wert), kann er nicht mehr korrekt gefunden werden. Daher müssen Schlüssel (logisch) unveränderlich sein oder ihr Hash / Vergleich darf sich nicht ändern.

Verkettung der Überläufer (Chaining)

- Array $T[0\,..\,m-1]$; jeder Bucket hält z.B. eine (kurze) verkettete Liste der dort liegenden Schlüssel/Werte.
- Insert: Finde $i = h(k)$, hänge (k,v) an Liste $T[i]$ an (falls Schlüssel schon da: Wert ersetzen).
- Lookup: Suche in Liste $T[h(k)]$ nach Schlüssel.
- Delete: Entferne Element aus dieser Liste.

- Erwartete Länge einer Liste: $\alpha = n/m$ (Lastfaktor) bei guter Hashfunktion $\Rightarrow$ erwartete Kosten $O(1 + \alpha)$.

Siehe Abbildung 5.8 für eine grafische Darstellung der Verkettung der Überläufer.

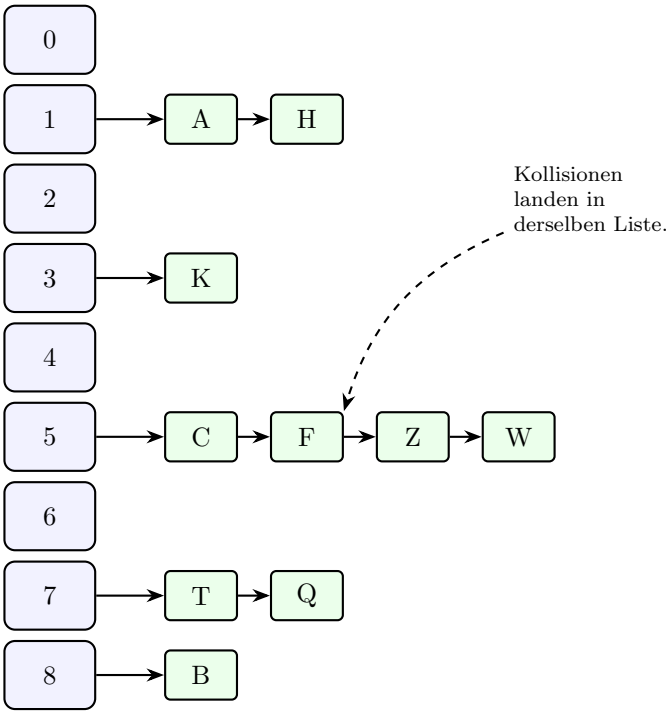


Abbildung 5.8: Hashtabelle mit Verkettung (Chaining): Jeder Bucket zeigt (falls nicht leer) auf den Kopf einer Liste kollidierender Elemente. Die erwartete Listenlänge entspricht dem Lastfaktor $\alpha = n/m$.

Offene Hashverfahren (Sondieren)

Die offenen Hashverfahren verfolgen eine andere Strategie im Falle von Kollisionen. Statt die kollidierenden Elemente in einer Liste zu speichern, werden sie in der Tabelle selbst an der nächsten geeigneten freien Stelle abgelegt. Die nächste geeignete Stelle zu berechnen wird als Sondieren bezeichnet.

Bei belegtem Platz wird eine neuer Platz wie folgt berechnet:

$$h_i(k) = (h(k) + f(i)) \bmod m, \quad i = 0, 1, 2, \ldots$$

Verwendet man als Funktion $f(i) : i$, so nennt man dies *lineares Sondieren*. Verwendet

man hingegen $f(i) = c_1 i + c_2 i^2$, wird dies als *quadratisches Sondieren* bezeichnet.

Lineares Sondieren hat den Nachteil, dass lange belegte Teilstücke in der Hashtabelle schneller wachsen als kurze (primäre Häufungen). Quadratisches Sondieren behebt dieses Problem, allerdings besteht die Gefahr, dass die Sondierfolgen nicht alle freien Plätze durchlaufen, und deshalb nicht alle freien Plätze gefunden werden können.

Eine weitere Möglichkeit ist Double Hashing: $h_i(k) = h_1(k) + i \cdot h_2(k)$. Hierbei werden zwei verschiedene Hashfunktionen h_1 und h_2 verwendet, um die Sondierfolge zu bestimmen. Dies kann helfen, Kollisionen weiter zu reduzieren und eine bessere Verteilung der Elemente in der Hashtabelle zu erreichen. Die Wahl von h_2 ist hierbei jedoch kritisch, ebenso ist das Verfahren aufwändiger als lineares oder quadratisches Sondieren.

Vor- und Nachteile der Verfahren

- Chaining: Einfach, Performance robust, benötigt Zeiger/Overhead.
- Offene Adressierung: Cache-freundlich (alles im Array), aber sensibler gegenüber hoher Auslastung; Clusterbildung möglich.

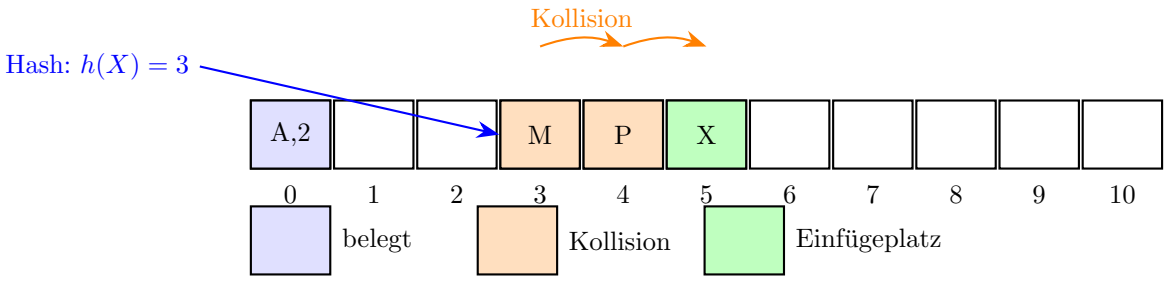


Abbildung 5.9: Die Grafik zeigt das Einfügen von Schlüssel X mit Hash 3: Position 3 ist belegt, lineares Sondieren prüft 4 (auch belegt), findet 5 frei und legt dort ab. Die Anzahl besuchter Buckets hängt vom Cluster vor der Zielposition ab; geringe Auslastung minimiert durchschnittliche Sondenlänge.

Beispiel 5.1 (Lineares Sondieren)**:**

$$m = 8, \qquad h'(k) = k \bmod 8$$

Einzufügende Schlüssel (in dieser Reihenfolge): $10, 19, 31, 22, 14, 16$

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0

Schrittweises Einfügen (lineares Sondieren)

1) 10: $h'(10) = 2 \Rightarrow$ Slot 2 frei $\Rightarrow$ einfügen.
2) 19: $h'(19) = 3$ frei $\Rightarrow$ Slot 3.
3) 31: $h'(31) = 7$ frei $\Rightarrow$ Slot 7.
4) 22: $h'(22) = 6$ frei $\Rightarrow$ Slot 6.
5) 14: $h'(14) = 6$ Kollision bei 6 (22) $\Rightarrow$ prüfe 7 (31, belegt) $\Rightarrow$ prüfe 0 (frei) $\Rightarrow$ Slot 0.
6) 16: $h'(16) = 0$ Kollision bei 0 (14) $\Rightarrow$ prüfe 1 (frei) $\Rightarrow$ Slot 1.

Endzustand der Hashtabelle

Index	0	1	2	3	4	5	6	7
Wert	14	16	10	19			22	31

Beobachtung: Die Clusterbildung zeigt sich daran, dass die Kollision von 14 (Start bei 6) die Sondenfolge bis zum Anfang (Wrap-Around) verlängert; 16 trifft danach sofort erneut auf eine belegte Zelle. Längere zusammenhängende belegte Abschnitte wachsen schneller (Primär-Cluster).

Beispiel 5.2 (Quadratisches Sondieren (mit $m = 8$))**:**

$m = 8, \qquad h'(k) = k \bmod 8, \qquad h_i(k) = \big(h'(k)+i+i^2\big) \bmod 8 \quad (i = 0, 1, 2, \ldots;\ c_1 = c_2 = 1)$

Einzufügende Schlüssel (in dieser Reihenfolge): $10, 19, 31, 22, 14, 16$

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0

Schrittweises Einfügen (quadratisches Sondieren)

1) 10: $h'(10) = 2$ frei $\Rightarrow$ Slot 2.
2) 19: $h'(19) = 3$ frei $\Rightarrow$ Slot 3.
3) 31: $h'(31) = 7$ frei $\Rightarrow$ Slot 7.
4) 22: $h'(22) = 6$ frei $\Rightarrow$ Slot 6.
5) 14: $h'(14) = 6$ belegt (22). Sondenfolge: $i = 1:\ 6 + 1 + 1 = 8 \equiv 0$ frei $\Rightarrow$ Slot 0.
6) 16: $h'(16) = 0$ belegt (14). Sondenfolge: $i = 1:\ 0 + 1 + 1 = 2$ belegt (10); $i = 2:\ 0 + 2 + 4 = 6$ belegt (22); $i = 3:\ 0 + 3 + 9 = 12 \equiv 4$ frei $\Rightarrow$ Slot 4.

Endzustand der Hashtabelle

Index	0	1	2	3	4	5	6	7
Wert	14		10	19	16		22	31

Beobachtung: Kollisionen (z.B. Startindex 6 für 22 und 14) führen zu Sprüngen, die kein zusammenhängendes primäres Cluster erzeugen (Slots: $6 \to 0$). Dennoch teilen alle Schlüssel mit gleichem Start-Hash dieselbe Sondenfolge (sekundäre Cluster). Mit $m = 8$ (Potenz von 2) muss geeignete Wahl von c_1, c_2 getroffen werden, um genügend unterschiedliche Slots zu erreichen.

Python

- Objekte sind als Dictionary-Schlüssel nutzbar, wenn sie `__hash__` und `__eq__` konsistent implementieren.
- Eingebaute unveränderliche Typen (int, str, tuple mit unveränderlichen Elementen) sind hashbar.
- Der Hash von `str` ist zwischen Prozessen (Standard) randomisiert (Hash-Seed) zur Abwehr von DoS-Angriffen durch gezielte Kollisionen.

Zusammenfassung Eine gute Hashfunktion ist schnell, verteilt Schlüssel gleichmäßig und minimiert so Kollisionen. Kollisionsbehandlung (Chaining oder offene Adressierung) plus kontrollierter Lastfaktor sichern amortisierte $O(1)$-Operationen für Insert, Lookup und Delete.

Laufzeitanalyse von Dictionaries

- Erfolgreiche Suche: amortisiert $O(1)$
- Einfügen: amortisiert $O(1)$
- Löschen: amortisiert $O(1)$
- Worst Case (pathologisch viele Kollisionen): $O(n)$

Konsistenzanforderungen

- Hash und Gleichheit müssen konsistent sein: $k_1 = k_2 \Rightarrow h(k_1) = h(k_2)$.
- Schlüssel dürfen sich nach Einfügen nicht (logisch) ändern.

Python-Dicts

Python-Dictionaries und -Sets nutzen eine optimierte Form offener Adressierung mit perturbierter Sondenfolge und speichern (kompakt) Hash und Schlüssel/Wert zusammen. Löschungen hinterlassen spezielle Marker, periodisches Resizing räumt auf. Iterationsreihenfolge (Einfügereihenfolge stabil) ist implementierungs- bedingt, aber praktisch nutzbar.

```
# Erzeugen
d = {}                                  # leeres Dict
person = {"name": "Ada", "age": 37}

# Zugriff / Einfügen / Überschreiben
person["city"] = "London"               # Einfügen
person["age"] = 38                      # Überschreiben
print(person["name"])                   # Direktzugriff

# Sicherer Zugriff
print(person.get("email"))               # -> None
print(person.get("email", "kein Eintrag"))

# Existenztest
if "age" in person:
    ...

# Entfernen
del person["city"]                      # KeyError falls fehlt
age = person.pop("age")                 # entfernt + liefert Wert
val = person.pop("email", None)         # Default falls Key fehlt
k, v = person.popitem()                 # entfernt letztes (LIFO, 3.7+)

# Iteration
for k in person: ...                    # Keys
for k, v in person.items(): ...
for v in person.values(): ...
```

5.5 Sets (Mengen)

- Ein *Set* entspricht einer mathematischen Menge.
- Gleiche Elemente können in einem Set nur einmal enthalten sein.

- In einem Set kann man (im Gegensatz zur `ArrayList`) nicht mittels Index auf bestimmte Elemente zugreifen.
- Operationen auf Sets:
  - Hinzufügen eines Elementes
  - Entfernen eines Elementes
  - Überprüfen ob ein Element im Set enthalten ist

Die konkrete Umsetzung der oben definierten Anforderungen an die Datenstruktur Set kann auf verschiedene Arten erfolgen. Eine sehr gebräuchliche und effiziente Methode ist die Verwendung von Hashtabellen, was zu sogenannten HashSets führt. Baum-basierte Sets werden im anschließenden Abschnitt 5.5.2 behandelt.

5.5.1 HashSet

Die Grundidee hinter HashSets ist die Verwendung einer Hashfunktion, um die Elemente in der Menge zu organisieren. Jedes Element wird durch seinen Hashwert in eine bestimmte Position (Bucket) der Hashtabelle eingefügt. Dies ermöglicht einen schnellen Zugriff auf die Elemente, da die Suche, das Hinzufügen und das Entfernen im Durchschnitt in konstanter Zeit $O(1)$ erfolgen können. Hierbei wird der Vorteil gegenüber einer etwaigen naiven Umsetzung mittels Listen ersichtlich: bei Listen würde jeder Zugriff $O(n)$ Zeit benötigen, da im schlechtesten Fall die gesamte Liste durchsucht werden muss.

Python

In Python heißt der Typ `set`. Intern ist das ein Hash-basiertes Set (wie HashSet in Java).

Wichtige Operationen:

- `s = 1, 2, 3` → Erzeugung durch Literal
- `s = set([1, 2, 3])` → Erzeugung aus Liste
- `s.add(x)` → fügt ein Element hinzu (tut nichts, wenn schon vorhanden).
- `s.remove(x)` → entfernt ein Element, KeyError, falls nicht vorhanden.
- `s.discard(x)` → entfernt ein Element, ohne Fehler, falls nicht vorhanden.
- `s.pop()` → entfernt & gibt ein beliebiges (meist erstes) Element zurück.
- `s.clear()` → leert das Set.

Eigenschaften:

Die Elemente sind im Set einzigartig. Es gibt keine garantierte Reihenfolge (bis Python 3.6 war die Iterationsreihenfolge zufällig, seit Python 3.7 ist sie stabil und entspricht der Einfüge-Reihenfolge – ähnlich wie LinkedHashSet in Java, aber das ist nur ein Implementierungsdetail, kein Ordnungsversprechen). Zugriff, Einfügen und Entfernen sind im Schnitt O(1).

5.5.2 TreeSet

In vielen Programmiersprachen gibt es zusätzlich zu HashSets auch TreeSets. Diese verwenden intern balancierte Suchbäume um die Daten effizient zu verwalten. TreeSets bieten eine sortierte Sicht auf die Elemente und ermöglichen schnelle Such-, Einfüge- und Löschoperationen in $O(\log n)$.

Zu beachten ist, dass Elemente zur Verwendung in TreeSets vergleichbar sein müssen, um in den Suchbaum an der richtigen Stelle eingefügt werden zu können. Während Hashtabellen bei Überschreitung der Auslastung neu aufgebaut werden müssen, bleibt die Struktur von TreeSets stabil. Allerdings müssen die Suchbäume regelmäßig ausbalanciert werden, um die logarithmische Zeitkomplexität zu gewährleisten.

Während die Operationen auf HashSets im Durchschnitt in $O(1)$ Zeit erfolgen, benötigen die entsprechenden Operationen auf TreeSets $O(\log n)$ Zeit. Der wesentliche Vorteil von TreeSets liegt in der garantierten Sortierung der Elemente, was bei HashSets nicht der Fall ist. Aufgrund des sehr langsamen Anwachsens der Logarithmusfunktion bieten TreeSets aber dennoch in vielen Situationen eine gute Leistung. Insbesondere wenn es schwierig ist eine gute Hashfunktion aufzustellen, oder eben Sortierungen benötigt werden, stellen TreeSets die bessere Wahl dar. Anderenfalls sind HashSets meist die bessere Wahl.

Python

In Python gibt es standardmäßig kein TreeSet, jedoch kann man die Funktionalität mit der Bibliothek `sortedcontainers` nachbilden.

6 Algorithmen auf Graphen

6.1 Breiten- und Tiefensuche

Wir betrachten Algorithmen zur Traversierung von Bäumen; sie lassen sich auch auf allgemeine Graphen anwenden. Die beiden wichtigsten Verfahren sind die Breitensuche (Breadth-First Search, BFS) und die Tiefensuche (Depth-First Search, DFS).

BFS (Breitensuche) In jedem Schritt werden zunächst alle Nachbarn eines Knotens besucht, bevor man in die nächste Ebene weitergeht.

DFS (Tiefensuche) Man folgt einem Pfad so tief wie möglich, bevor man zu einer Verzweigung zurückkehrt.

Begriffe

Ein Knoten wird **entdeckt**, wenn er das erste Mal besucht wird, und **fertiggestellt**, wenn er das letzte Mal verlassen wird. Für manche Anwendungen ist es wichtig festzuhalten, wann dies geschieht. Dazu führen wir einen Zähler τ mit, der bei jedem Ereignis erhöht wird: $\tau_d(v)$ beim Entdecken, $\tau_f(v)$ beim Fertigstellen.

Beispiel Breitensuche

Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden (ëntdeckt") wurde. Unten ist der Verlauf der Breitensuche (Ausgangspunkt Knoten 1) in ausgewählten Schritten dargestellt. Grüne Knoten sind entdeckt (in die Queue aufgenommen), blaue fertiggestellt (alle Nachbarn verarbeitet), rote Kanten wurden gerade zur Entdeckung genutzt.

Die BFS besucht die Knoten (von 1 ausgehend) schichtweise:

$$1;\ 2,3;\ 4,5,6,7,8;9,10,11,12,13,14,15$$

Breitensuche: Datenstruktur (Queue)

In nahezu allen Programmiersprachen existiert eine Datenstruktur namens Queue (Warteschlange). Elemente werden hinten eingereiht; das älteste Element wird zuerst entnommen

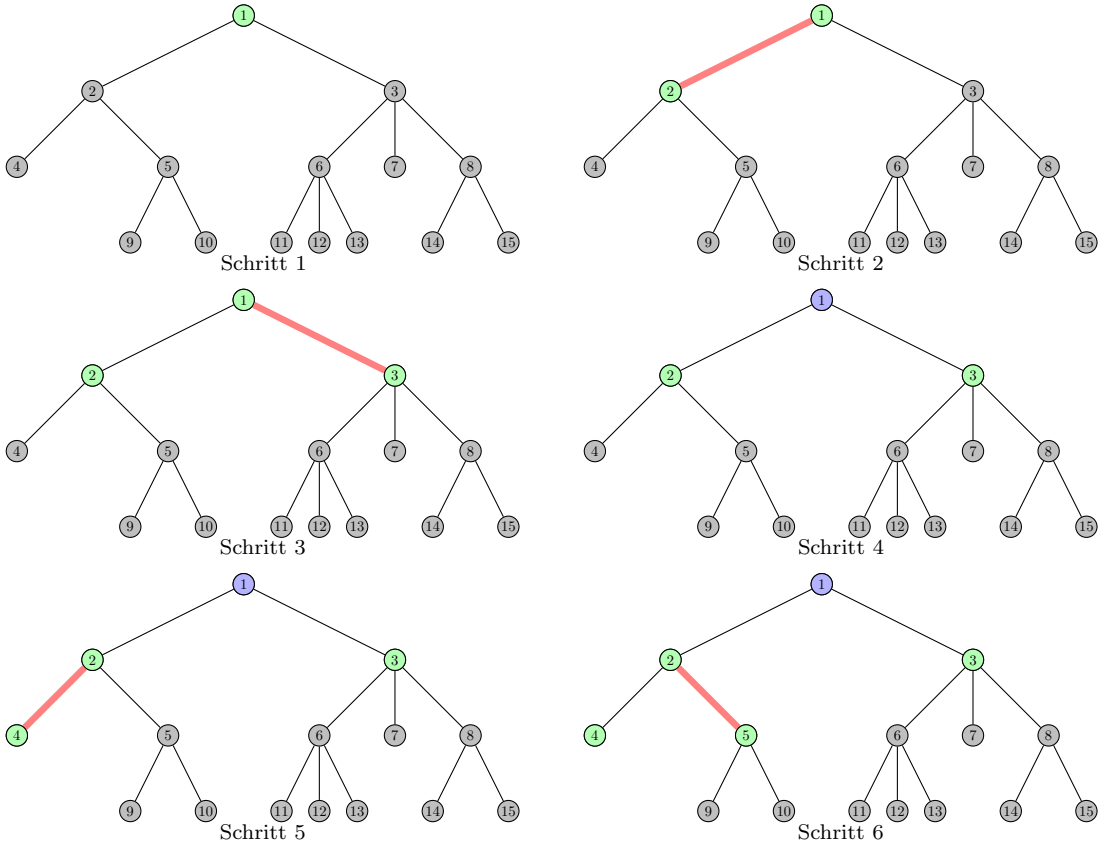


Abbildung 6.1: Breitensuche: Schritte 1–6

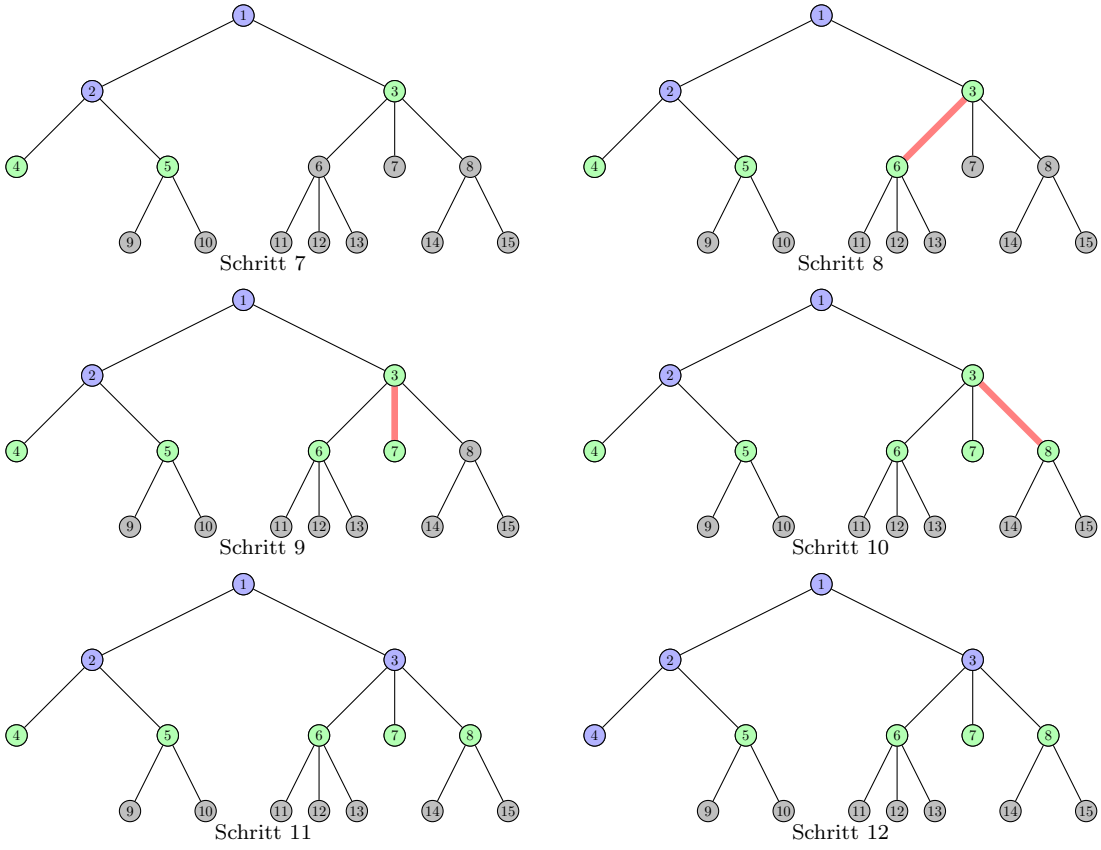


Abbildung 6.2: Breitensuche: Schritte 7–12

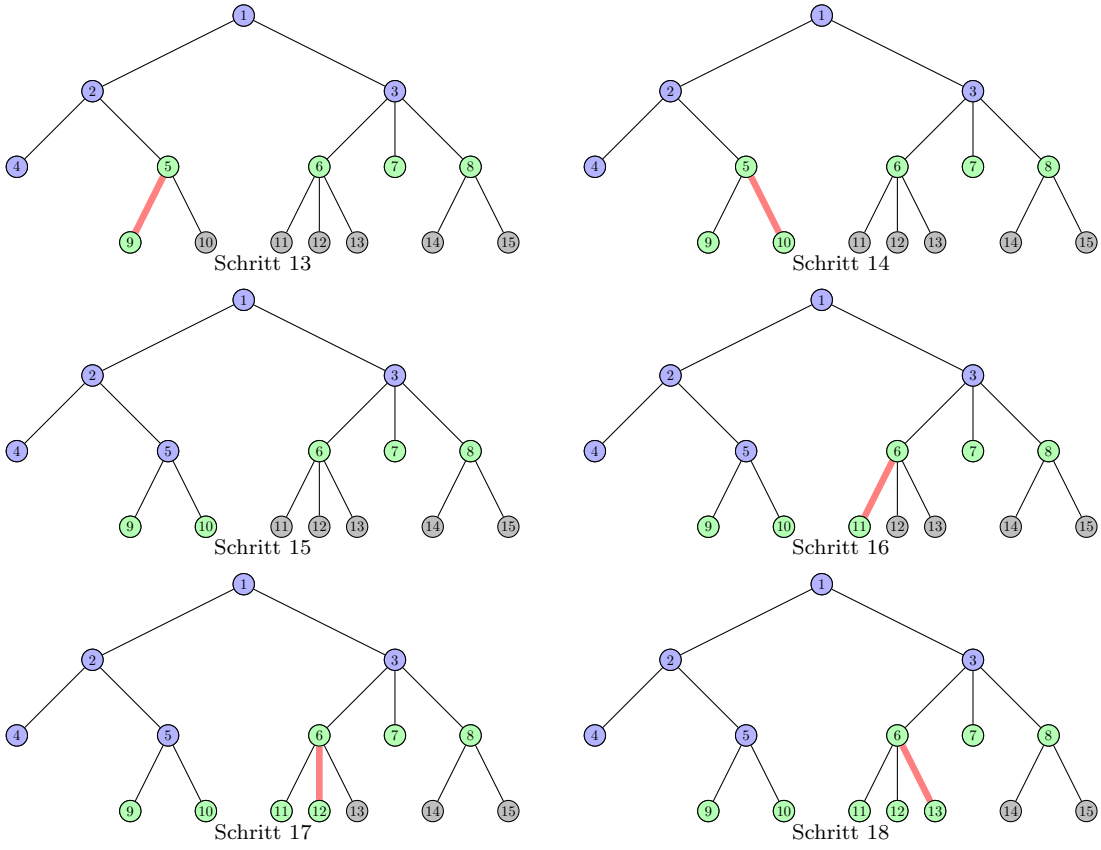


Abbildung 6.3: Breitensuche: Schritte 13–18

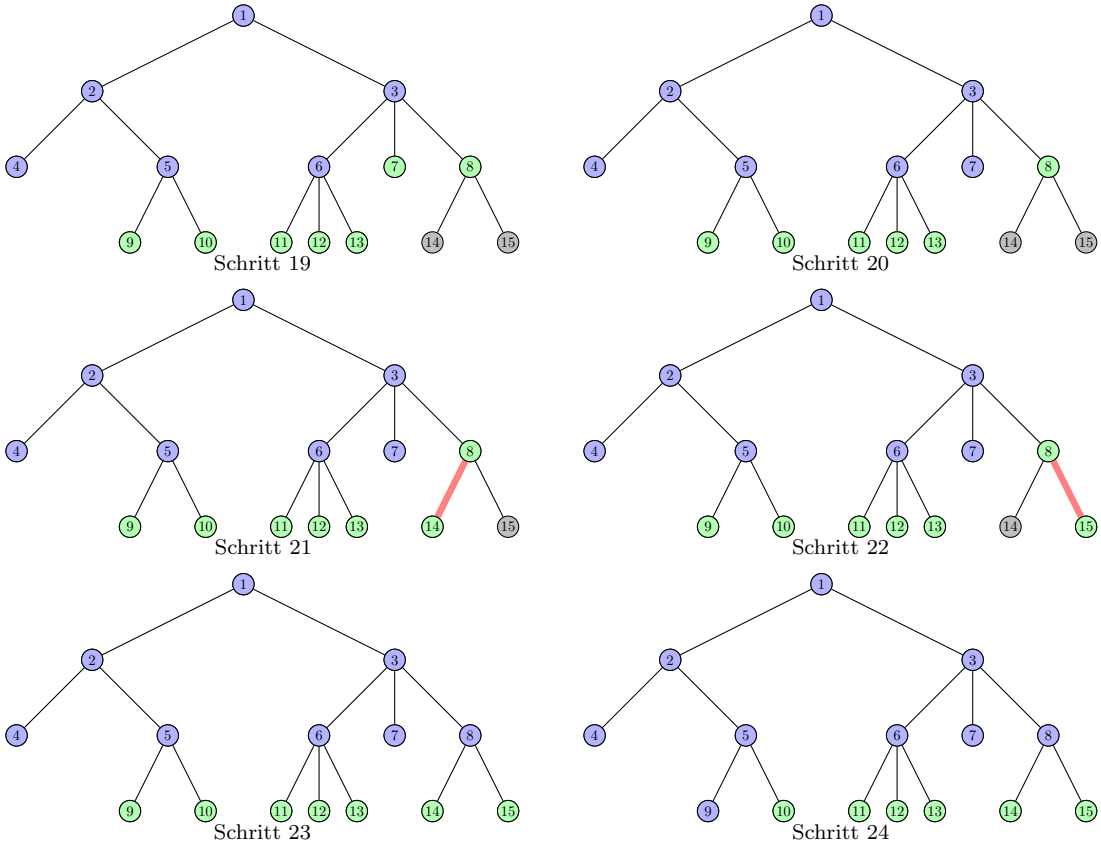


Abbildung 6.4: Breitensuche: Schritte 19–24

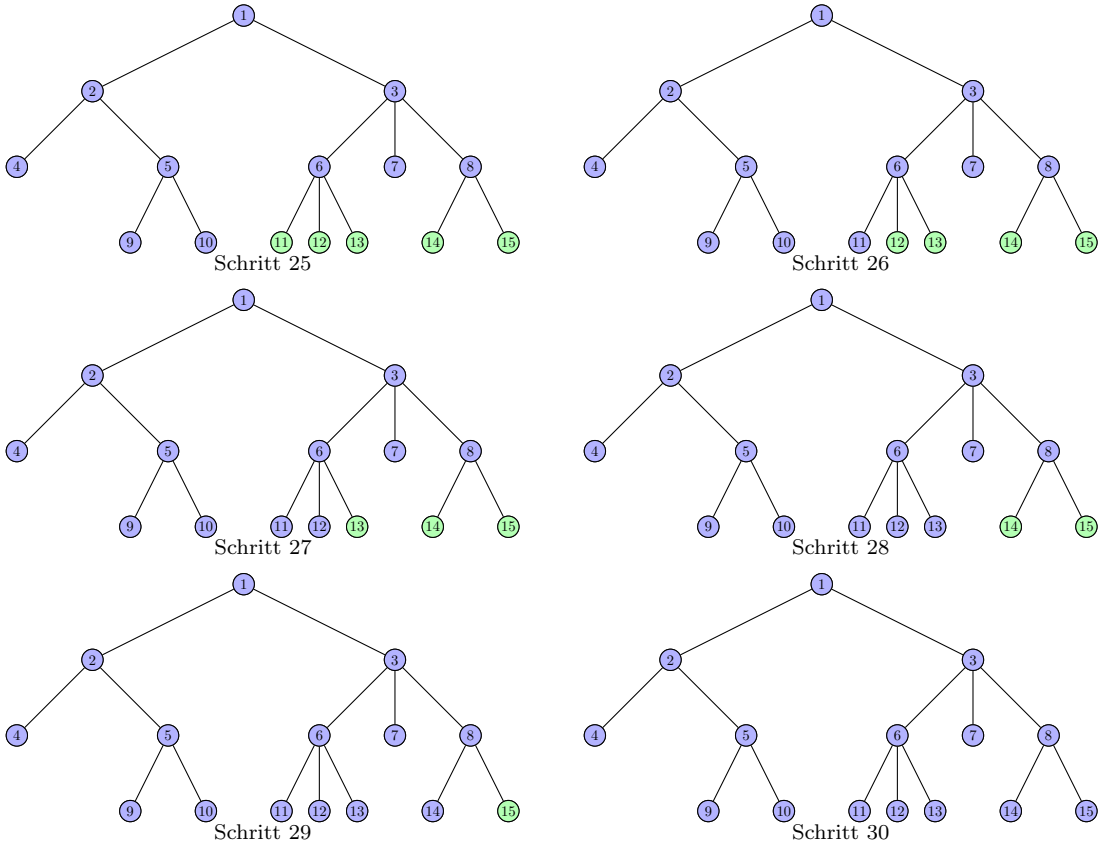


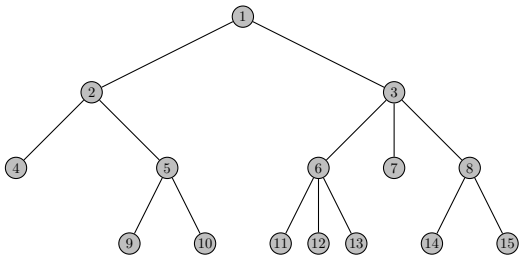
Abbildung 6.5: Breitensuche: Schritte 25–30

(FIFO).

Breitensuche: Algorithmus

1. Füge Startknoten (Wurzel) in Queue ein.
2. Entnimm vorn stehenden Knoten.
   - Falls gesuchter Knoten: Abbruch.
   - Sonst: füge alle bisher unbesuchten[1] Nachbarn hinten an.
3. Ist die Queue leer, wurden alle erreichbaren Knoten besucht (Abbruch), sonst gehe zu (2).

Beispiel Tiefensuche



Eine mögliche DFS-Besuchsreihenfolge (präfix) ausgehend von 1 ist z.B.:

$$1, 2, 4, 5, 9, 10, 3, 6, 11, 12, 13, 7, 8, 14, 15$$

Verlauf der Tiefensuche (ausgewählte Schritte). Grüne Knoten = entdeckt (aktiv auf dem Rekursions-/Stackpfad), blaue Knoten = fertiggestellt; rote Kante = gerade verfolgte Baumkante bei der Entdeckung. Gezeigt werden die ersten 9 Schritte sowie die letzten 2 Abschlüsse.

Tiefensuche: Datenstruktur (Stack)

In nahezu allen Programmiersprachen existiert eine Datenstruktur namens Stack (Stapel). Elemente werden oben abgelegt; das zuletzt eingelegte Element wird zuerst entnommen (LIFO).

[1] nicht entdeckt, nicht abgeschlossen

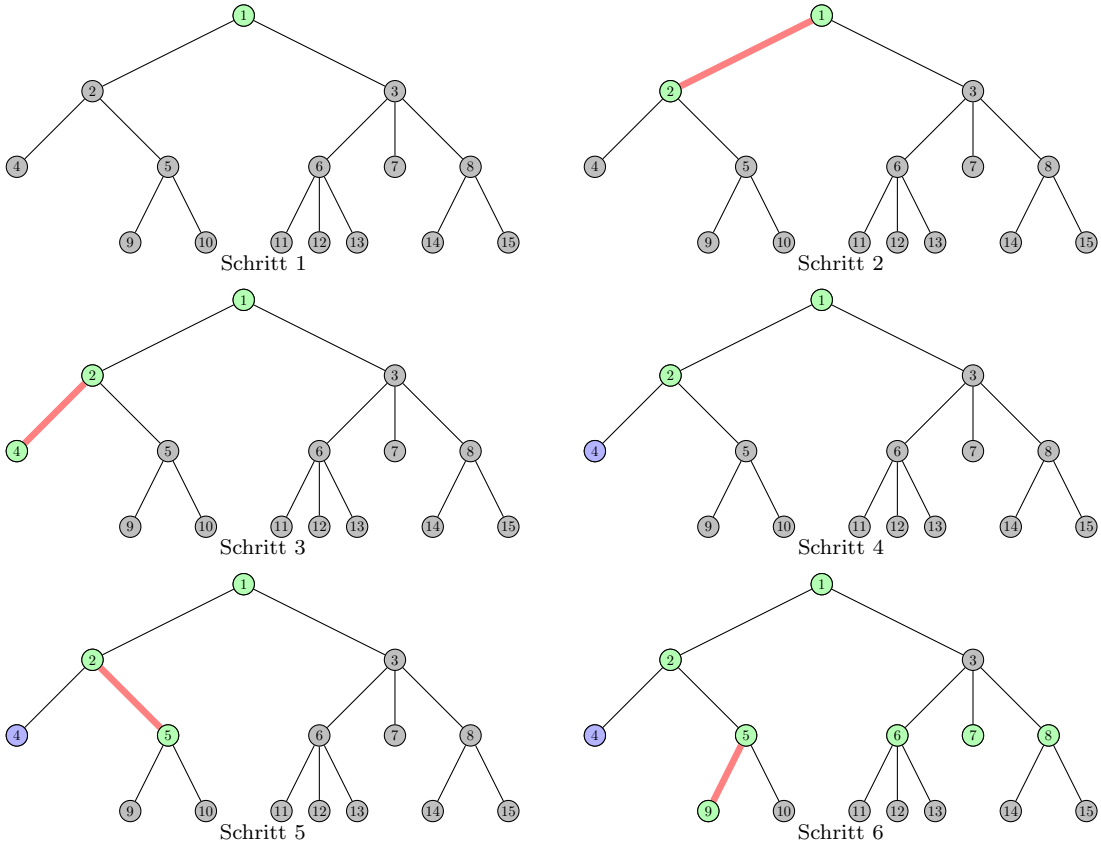


Abbildung 6.6: Tiefensuche: Schritte 1–6

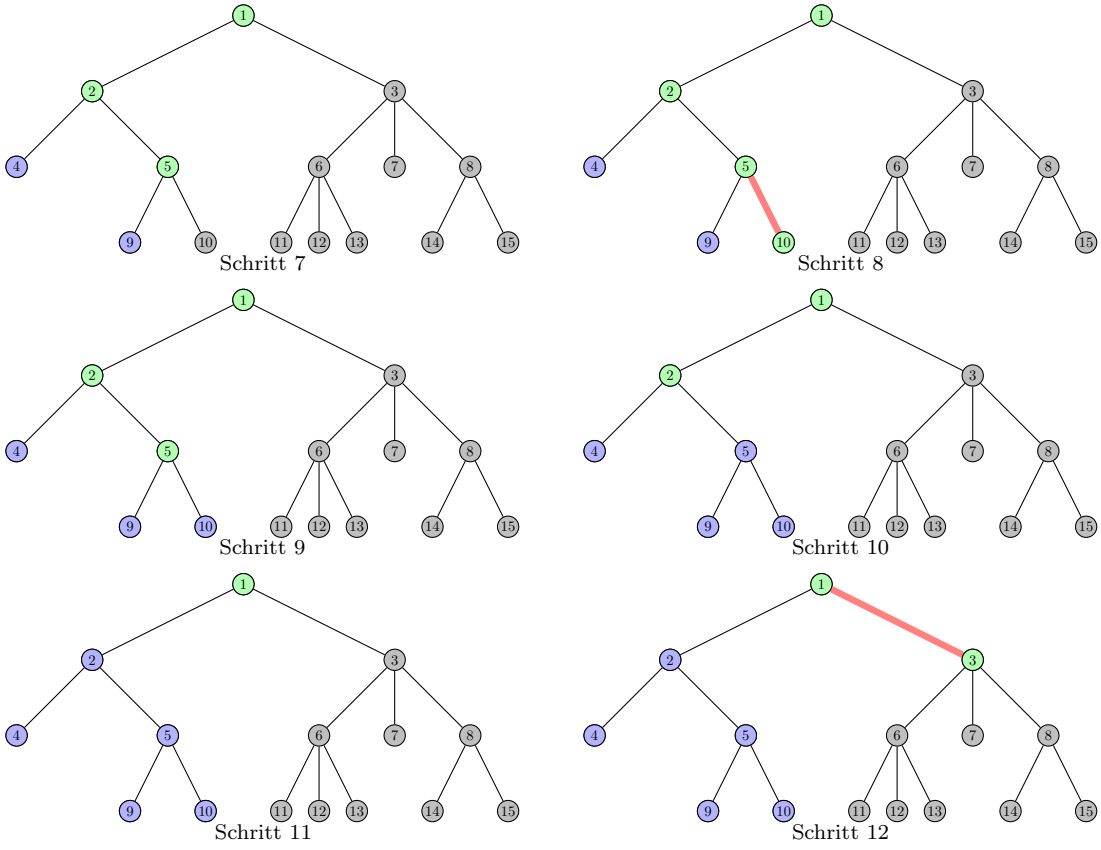


Abbildung 6.7: Tiefensuche: Schritte 7–12

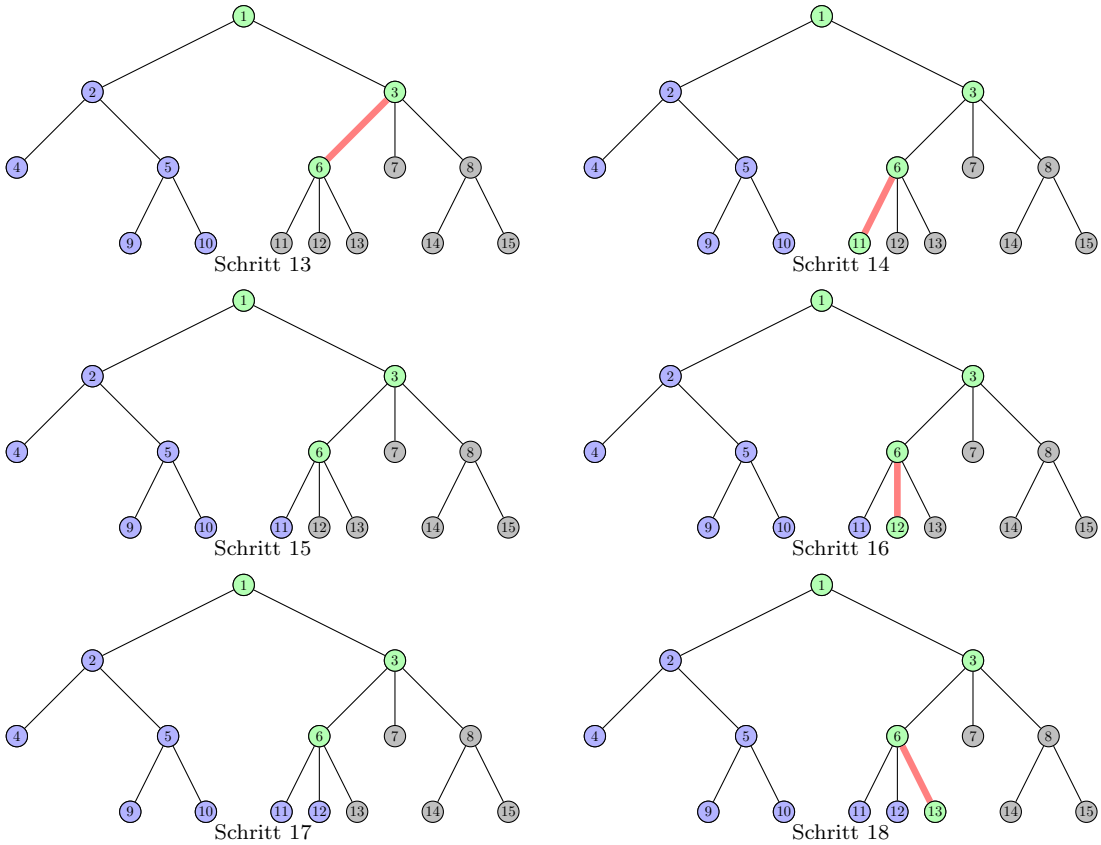


Abbildung 6.8: Tiefensuche: Schritte 13–18

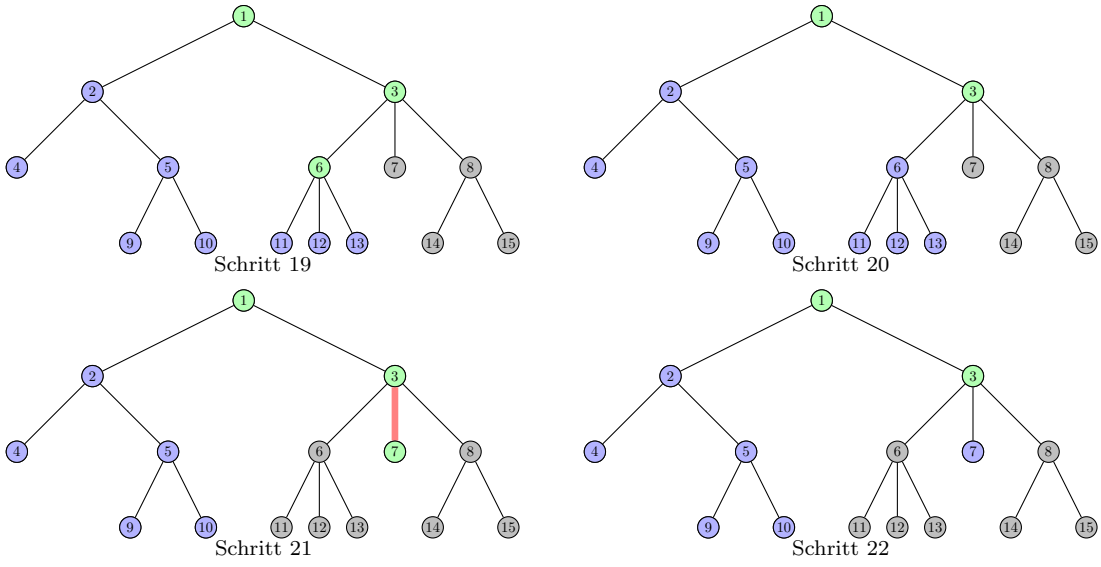


Abbildung 6.9: Tiefensuche: Schritte 19–22

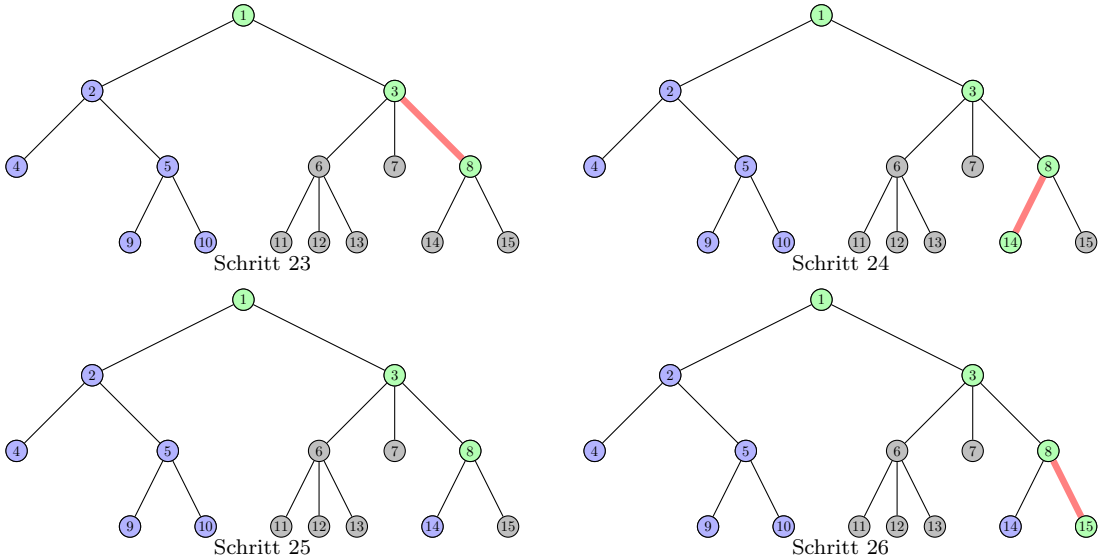


Abbildung 6.10: Tiefensuche: Schritte 23–26

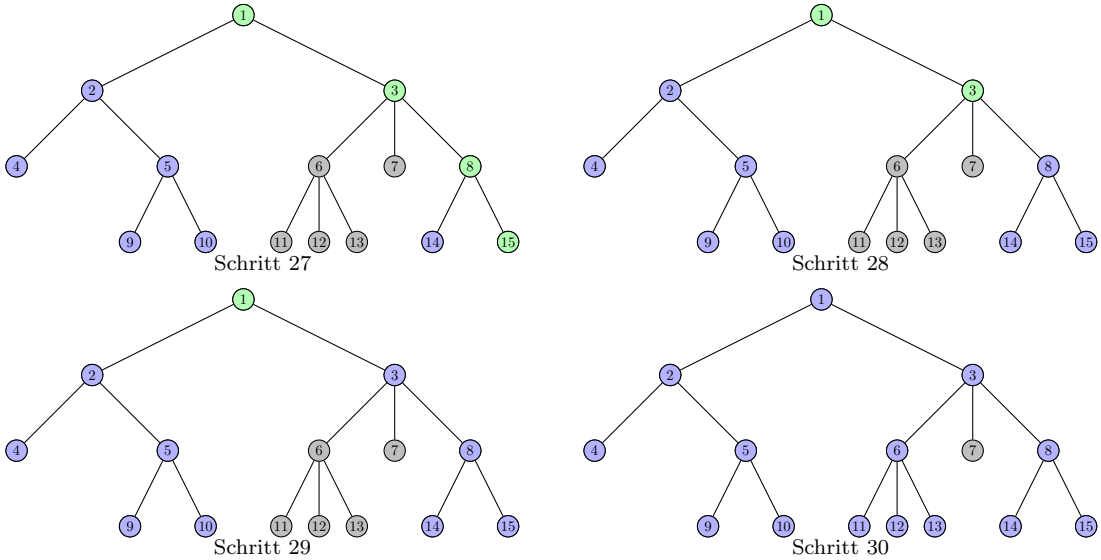


Abbildung 6.11: Tiefensuche: Schritte 27–30

Tiefensuche: Algorithmus

Algorithmus 17: Tiefensuche (iterativ)

```
Function DFS(G = (V, E), Startknoten v, gesuchter Knoten s)
    Result: Knoten s ∈ V, falls erreichbar
    Stack S;
    S.push(v);
    while S not empty do
        v = S.pop();
        if v = s then
            return v;
        if v noch nicht besucht then
            Markiere v als besucht;
            for alle Kanten (v, u) ∈ E(G) do
                if u noch nicht besucht then
                    S.push(u);
```

Anmerkungen

- Beide Verfahren funktionieren in allgemeinen Graphen; bereits besuchte Knoten werden nicht erneut verarbeitet.
- Anwendungen BFS: 2-Färbbarkeit, kürzeste Wege (ungewichtete Graphen), Erkennen von Komponenten.
- Anwendungen DFS: Test auf Kreisfreiheit, topologische Sortierung, starke Zusammenhangskomponenten.

6.2 Algorithmen für kürzeste Wege

Kürzeste-Wege-Probleme fragen nach minimalen Pfadkosten zwischen Knoten eines Graphen und modellieren u.a. Navigation, Netzwerk-Routing, Produktions- und Zeitplanung, Ressourcen- oder Risikominimierung. Die Pfadlänge kann die reine Kantenanzahl (bei ungewichtete Graphen) oder die Summe von Gewichten (Kosten, Distanzen, Zeiten) sein – teilweise auch mit negativen Werten. Anwendungen reichen von GPS und Internet-Routing (OSPF, BGP-Varianten) über Logistik und Projektplanung bis zur Bioinformatik und KI-Suchverfahren. Im Folgenden skizzieren wir Varianten und zentrale Algorithmen und vertiefen anschließend den Algorithmus von Dijkstra.

6.2.1 Problemvarianten und Algorithmen

Kürzeste Wege treten in vielen Variationen auf. Grundsätzlich unterscheiden wir:

- **Ungwichtete Graphen** (oder alle Gewichte gleich): Anzahl der Kanten bestimmt die Distanz. Lösung: *Breadth-First Search (BFS)* ab Startknoten.
- **Gewichtete Graphen** mit nichtnegativen Kantenkosten: klassische Variante; Distanzen sind Summen von Gewichten. Hauptalgorithmus: *Dijkstra* (siehe nächsten Abschnitt).
- **Gewichtete Graphen mit negativen Gewichten**: Dijkstra nicht mehr korrekt. Benötigt Verfahren, das auch negative Relaxationen zulässt (z.B. *Bellman–Ford*).
- **Negative Zyklen**: Existiert ein von der Quelle erreichbarer Kreis mit Gesamtkosten < 0, so ist keine wohldefinierte endliche kürzeste Distanz zu allen beteiligten Knoten vorhanden ("$-\infty$"). Algorithmen müssen dies erkennen (Bellman–Ford, Varianten von Floyd–Warshall).

Problemformulierung. Häufig betrachtete Varianten:

- *Single-Pair Shortest Path (SPSP)*: kürzester Weg zwischen gegebenen s und t.
- *Single-Source Shortest Paths (SSSP)*: alle Ziele ab Start s (liefert auch SPSP als Spezialfall).
- *All-Pairs Shortest Paths (APSP)*: Distanzen für alle Paare (u, v). Kann durch wiederholte SSSP oder spezieller über Matrix-/Dynamik-Verfahren gelöst werden.

Bellman-Gleichung. Die optimalen Distanzen $d(v)$ (für SSSP ohne negative Zyklen) erfüllen das Optimierungsprinzip

$$d(s) = 0, \qquad d(v) = \min_{(u,v)\in E} \{d(u) + w(u, v)\} \quad (v \neq s),$$

also eine Fixpunkt-Gleichung, die durch sukzessive Relaxation (BFS, Dijkstra, Bellman–Ford) angenähert und schließlich erreicht wird.

Überblick über zentrale Algorithmen.

- **BFS** (SSSP in $O(n + m)$ für ungewichtete / einheitlich gewichtete Graphen).
- **Dijkstra** (SSSP für nichtnegative Gewichte; mit geeigneten Prioritätsstrukturen $O(m \log n)$).

- **Bellman–Ford** (SSSP mit negativen Gewichten, erkennt negative Zyklen; $O(nm)$).
- **Floyd–Warshall** (APSP mittels dynamischer Programmierung; $O(n^3)$; findet negative Zyklen über Diagonale).
- **Matrixbasierte APSP** (Wiederholtes "Min-Plus"-Matrix-Produkt / exponentiation by squaring; theoretisch durch schnelle Matrixmultiplikation asymptotisch verbesserbar, praktisch selten relevant).

Im Folgenden fokussieren wir auf den Algorithmus von Dijkstra als repräsentatives effizientes Verfahren für Graphen mit nichtnegativen Gewichten.

6.2.2 Algorithmus von Dijkstra

Der Algorithmus von Dijkstra berechnet kürzeste Wege in einem Graphen mit nichtnegativen Gewichten $w_{uv} \geq 0$. Ähnlich wie BFS expandiert er schrittweise die Menge abgeschlossener Knoten, wählt aber jeweils den aktuell kleinsten vorläufigen Abstand δ_v aus.

Zwischenergebnisse. Für jeden Knoten v wird ein Wert δ_v (bester bisher gefundener Abstand) sowie ein Vorgängerzeiger gespeichert. Iteration: (1) Wähle Knoten k mit minimalem δ_k unter den nicht abgeschlossenen. (2) Markiere k abgeschlossen. (3) Relaxiere alle ausgehenden Kanten (k, v): Falls $\delta_v > \delta_k + w_{kv}$, setze $\delta_v := \delta_k + w_{kv}$ und Vorgänger$(v) := k$.

Algorithmus 18: Dijkstra$(G = (V, E), w, s)$

```
Data: Graph mit w_uv ≥ 0
for v ∈ V do
    δ_v ← ∞
δ_s ← 0; Prioritätswarteschlange Q (alle Knoten) ; while Q ≠ ∅ do
    u ← ExtractMin(Q) ; markiere u fertiggestellt; for (u, v) ∈ E mit v nicht
     fertiggestellt do
        if δ_v > δ_u + w_uv then
            δ_v ← δ_u + w_uv ; Vorgänger(v)← u;
```

Anmerkung. Eine einfache Array-Implementierung für Q führt zu $O(n^2 + m)$, Heaps zu $O((n + m) \log n)$, Fibonacci-Heaps amortisiert $O(n \log n + m)$.

Beispiel (schrittweise). Wir suchen den kürzesten Weg von A nach H. Jede Abbildung zeigt den Zustand nach einem Schritt des Algorithmus: blau = fertiggestellt (endgültiger

Abstand, Vorgängerkante als blaue Pfeilkante), grün = aktuell relaxierte Kanten dieses Schrittes, orange = vorläufiger Abstand (noch nicht fertiggestellt), ∞ = noch nicht erreicht. Reihenfolge der Fertigstellung: $A(0), B(1), E(3), C(5), D(6), H(7), F(8), G(11)$; kürzester Weg $A \to H$: $A \to B \to E \to D \to H$.

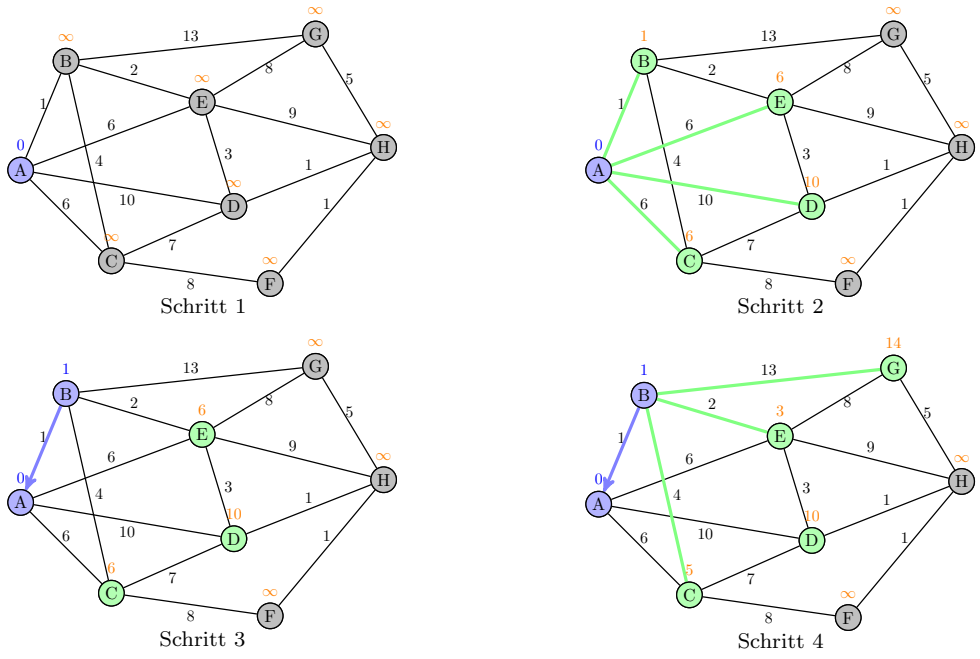


Abbildung 6.12: Dijkstra: Schritte 1–4

1 **Ziel: kürzester Weg $A \to H$.** Startknoten A wird fertiggestellt; alle anderen Knoten noch ∞.

2 Relaxation aller Kanten von A: B, C, D, E werden entdeckt. Werte: $\delta_A = 0,\ \delta_B = 1,\ \delta_C = 6,\ \delta_E = 6,\ \delta_D = 10$.

3 Knoten mit kleinstem vorläufigen Abstand wird fertiggestellt: B (1). Blaue Vorgängerkante (A, B).

4 Relaxation von B: neue Werte $\delta_E = 3,\ \delta_C = 5,\ \delta_G = 14$ (nur C, E verbessert). Knoten C, E, D, G vorläufig.

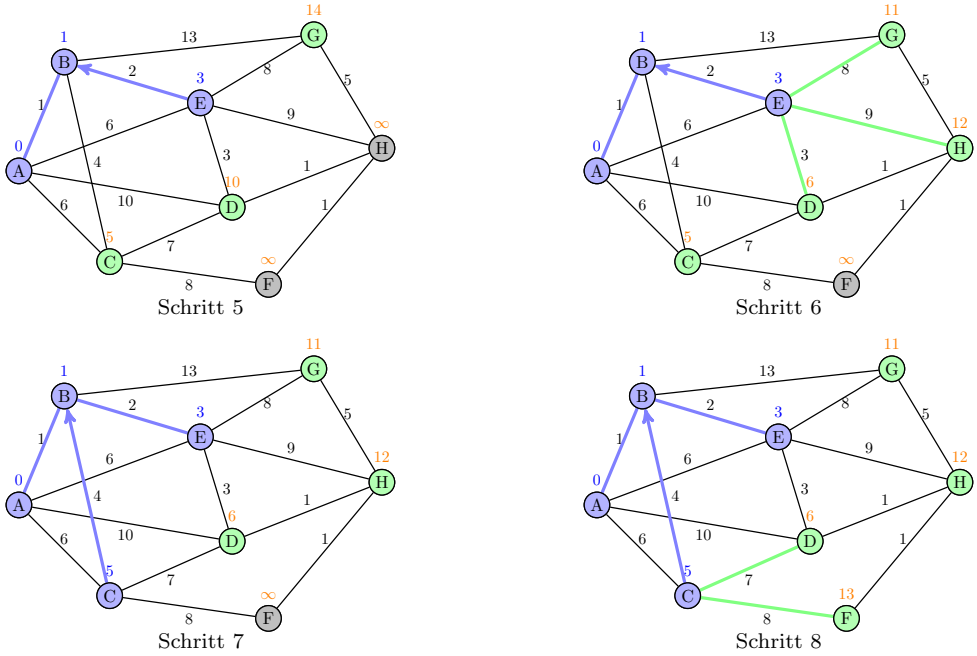


Abbildung 6.13: Dijkstra: Schritte 5–8

5 E hat nun den kleinsten Wert (3) und wird fertiggestellt; Vorgänger (B, E).
6 Relaxation von E: Aktualisierung $\delta_D = 6$ (statt 10), $\delta_H = 12$, $\delta_G = 11$.
7 Kleinster vorläufiger Wert jetzt $C = 5$: C wird fertiggestellt; Vorgänger (B, C).
8 Relaxation von C: $\delta_F = 13$, δ_D bleibt 6 (kein Update, da $5 + 7 > 6$).

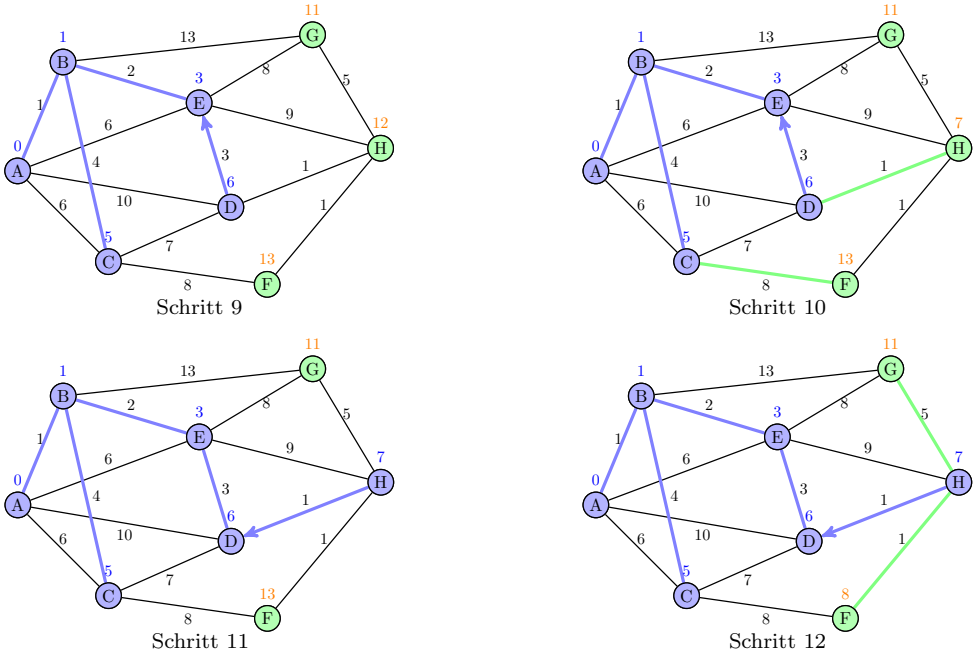


Abbildung 6.14: Dijkstra: Schritte 9–12

9 D (6) wird fertiggestellt; Vorgänger (E, D).
10 Relaxation von D: δ_H verbessert sich auf 7 (statt 12); δ_F unverändert.
11 H (7) wird fertiggestellt; Vorgänger (D, H).
12 Relaxation von H: $\delta_F = 8$ (Verbesserung), δ_G bleibt 11.

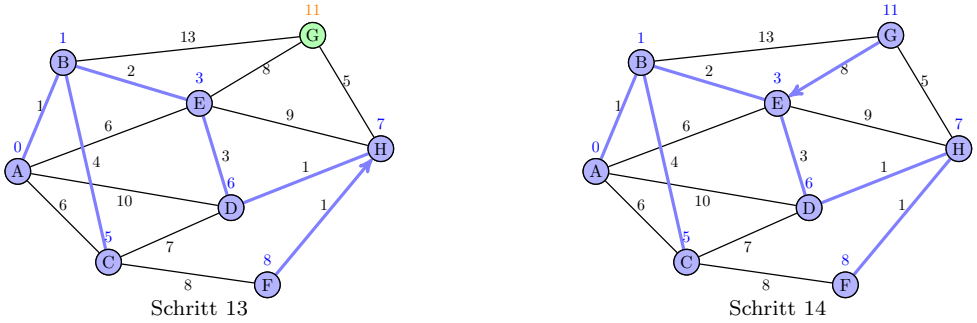


Abbildung 6.15: Dijkstra: Schritte 13–14

13 F (8) wird fertiggestellt; Vorgänger (H, F).
14 Letzter verbleibender Knoten G (11) wird fertiggestellt; Vorgänger (E, G). Alle Abstände endgültig.

Reihenfolge der Fertigstellung und Distanzen:

Schritt	endgültiger Knoten	δ
0	A	0
1	B	1
2	E	3
3	C	5
4	D	6
5	H	7
6	F	8
7	G	11

Laufzeitanalyse

Die Laufzeit des Algorithmus von Dijkstra hängt von der Implementierung der Prioritätswarteschlange ab. Die naive Umsetzung mittels Liste führt zu einer Laufzeit von $O(n^2)$. Die Umsetzung mittels Heap ermöglicht den Knoten mit minimalem δ-Wert in $O(\log n)$ zu ermitteln, gleiches gilt für decreaseKey. Dies führt dann zu einer verbesserten Laufzeit von $O((n+m)\log n) = O(m \log n)$. Bei sehr dünn besetzten Graphen entspricht dies quasi $O(n \log n)$. Durch Fibonacci-Heaps[2] kann die Laufzeit weiter auf $O(n \log n + m)$ verbessert werden. Dieses Ergebnis ist jedoch eher von theoretischer Bedeutung.

Operation		Queue Implementierung		
Name	Anzahl	Liste	Heap	Fibonacci Heap[3]
decreaseKey [18]	m	$O(1)$	$O(\log n)$	$O(1)$
getMin [18]	n	$O(n)$	$O(\log n)$	$O(\log n)$
create [18]	1	$O(n)$	$O(n)$	$O(n)$
Gesamt		$O(n^2 + m)$ $= O(n^2)$	$O((n+m)\log n)$	$O(n \log n + m)$

Tabelle 6.1: Laufzeitanalyse des Algorithmus von Dijkstra in Abhängigkeit der Queue-Implementierung. In der Spalte ganz links ist in eckigen Klammern auf die Zeile der jeweiligen Operation im Pseudocode verwiesen.

6.3 Minimale Spannbäume

Definition 6.1 (Spannbaum)**:** Ein *Spannbaum* T zu einem Graphen $G = (V, E)$ ist ein (auf-)spannender Teilgraph von G der ein Baum ist.

[2] Fibonacci-Heaps sind eine spezielle Datenstruktur, die eine amortisierte Laufzeit von $O(1)$ für die Operationen insert und decreaseKey bietet. Eine genauere Betrachtung ist nicht Gegenstand dieses Skriptums.

Wir betrachten nun ungerichtete, schlichte und zusammenhängende Graphen $G = (V, E)$ mit einer Gewichtsfunktion $w : E \to \mathbb{R}^+$ auf den Kanten $e \in E(G)$:

Definition 6.2 (Minimaler Spannbaum)**:** Ein *Minimaler Spannbaum* T von $G = (V, E)$ mit $w : E \to \mathbb{R}^+$ für alle $e \in E(G)$ ist ein zusammenhängender Teilgraph mit $|E(T)| = |V(G)| - 1$ der alle Knoten enthält, und für den gilt:

$$\sum_{e \in E(T)} w(e) \text{ ist minimal}$$

Ein Minimaler Spannbaum (Minimum Spanning Tree (MST)) ist also ein Baum mit minimalen Kantengewichten (=Kantenkosten).

6.3.1 Algorithmus von Kruskal

Der Algorithmus von Kruskal ist ein *Greedy-Algorithmus* zur Berechnung eines minimalen Spannbaums in gewichteten ungerichteten Graphen. Greedy-Algorithmen haben die Eigenschaft, in jedem Schritt die lokal optimale Wahl zu treffen, um eine Gesamtlösung zu finden. Im Allgemeinen führt eine derartige Vorgehensweise nicht zur insgesamt optimalen Lösung. Im konkreten Fall liefert jedoch der Algorithmus von Kruskal einen minimalen Spannbaum (d.h. die tatsächlich optimale Lösung).

Die Grundidee des Algorithmus ist, zunächst alle Kanten $e \in E(G)$ aufsteigend nach $w(e)$ zu sortieren. Dann wird der Algorithmus in einer Schleife jede Kante der Reihe nach betrachten und entscheiden, ob sie in den minimalen Spannbaum aufgenommen werden soll oder nicht. Dies geschieht, wenn das Einfügern der Kante keinen Kreis erzeugen würde. Am Ende entsteht ein Baum mit minimalem Gewicht.

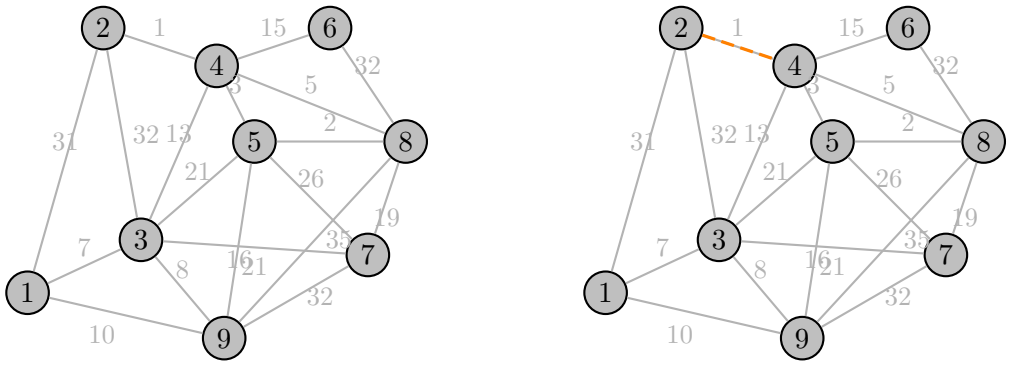
Algorithmus 19: Algorithmus von Kruskal

Function *Kruskal(Graph $G = (V, E)$)*
Result: Minimum Spanning Tree T
Ordne alle Kanten $e \in E(G)$ aufsteigend nach $w(e)$;
$\Rightarrow L = (e_1, e_2, \ldots, e_n)$ mit $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_n)$
$V(T) = V(G)$;
$E(T) = \emptyset$;
for $e \in L$ **do**
  if $T = (V, E(T) \cup \{e\})$ *ist kreisfrei* **then**
    $E(T) = E(T) \cup \{e\}$;

Die Kreisfreiheit kann mit Breiten- oder Tiefensuche berechnet werden. Nach Hinzufügen von $n - 1$ Kanten wird der Algorithmus beendet (jede weitere hinzugefügte Kante würde nun einen Kreis erzeugen).

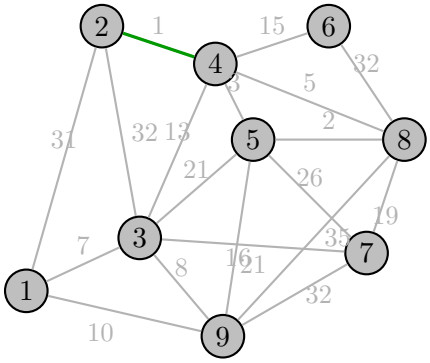
Beispiel 6.1: Ausführung des Algorithmus von Kruskal auf einem Beispielgraphen. Die aufsteigend sortierten Kanten sind: $w([2,4]) = 1, w([5,8]) = 2, w([4,5]) = 3, w([4,8]) = 5, w([1,3]) = 7, w([3,9]) = 8, w([1,9]) = 10, w([3,4]) = 13, w([4,6]) = 15, w([5,9]) = 16, w([7,8]) = 19, w([3,5]) = 21, w([3,7]) = 21, w([5,7]) = 26, \ldots$.

Eine im aktuellen Schritt betrachtete Kante wird orange markiert. Wird die Kante tatsächlich gewählt, wird sie sodann grün dargestellt, anderenfalls rot.

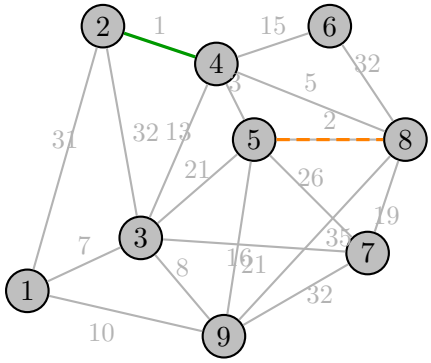


(a) (1) Ausgangszustand (alle Kanten grau). (b) (2) Kandidat $(2, 4)$ mit Gewicht 1.

Abbildung 6.16: Kruskal: Schritte 1–2.

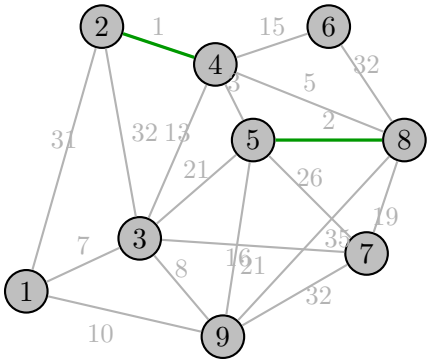


(a) (3) $(2,4)$ aufgenommen.

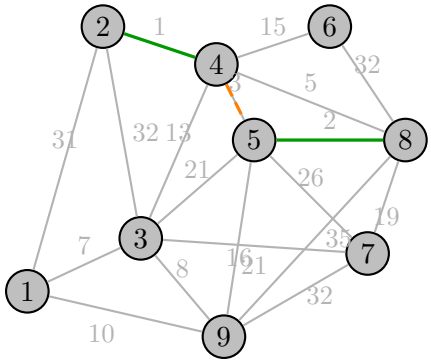


(b) (4) Kandidat $(5,8)$ (2).

Abbildung 6.17: Kruskal: Schritte 3–4.

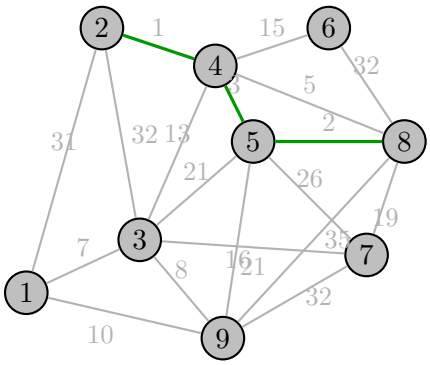


(a) (5) $(5,8)$ aufgenommen.

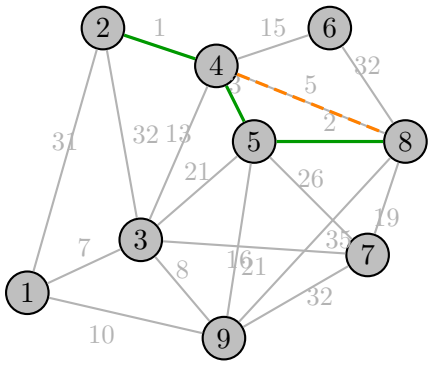


(b) (6) Kandidat $(4,5)$ (3).

Abbildung 6.18: Kruskal: Schritte 5–6.

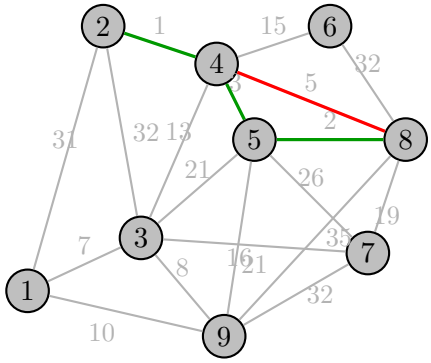


(a) (7) $(4,5)$ aufgenommen.

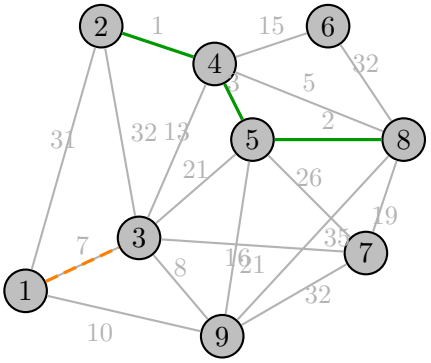


(b) (8) Kandidat $(4,8)$ (5).

Abbildung 6.19: Kruskal: Schritte 7–8.

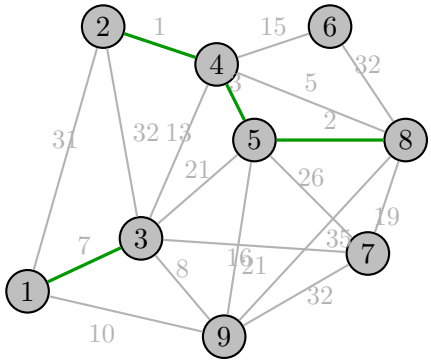


(a) (9) $(4, 8)$ verworfen (Kreis).

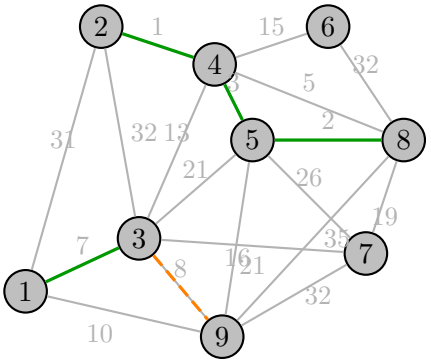


(b) (10) Kandidat $(1, 3)$ (7).

Abbildung 6.20: Kruskal: Schritte 9–10.

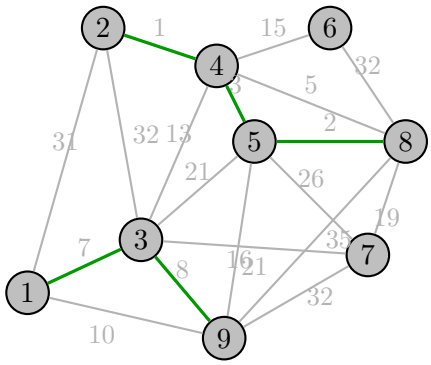


(a) (11) $(1, 3)$ aufgenommen.

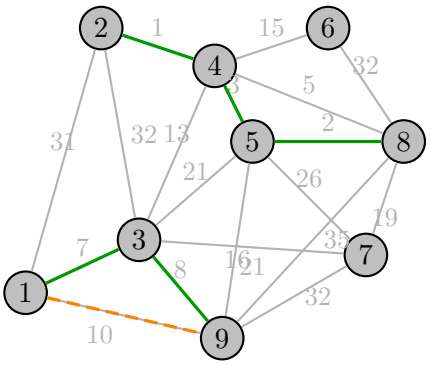


(b) (12) Kandidat $(3, 9)$ (8).

Abbildung 6.21: Kruskal: Schritte 11–12.

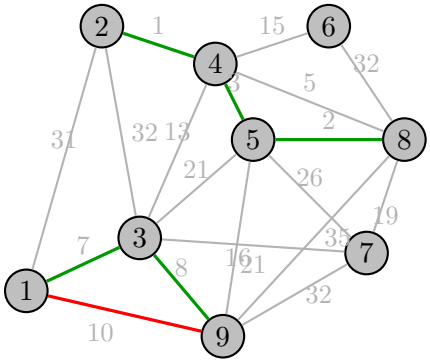


(a) (13) $(3, 9)$ aufgenommen.

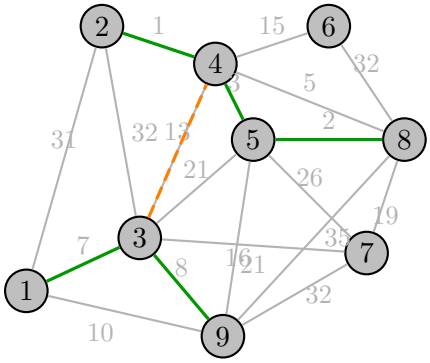


(b) (14) Kandidat $(1, 9)$ (10).

Abbildung 6.22: Kruskal: Schritte 13–14.

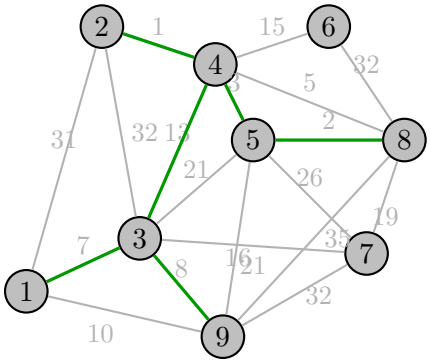


(a) (15) $(1, 9)$ verworfen (Kreis).

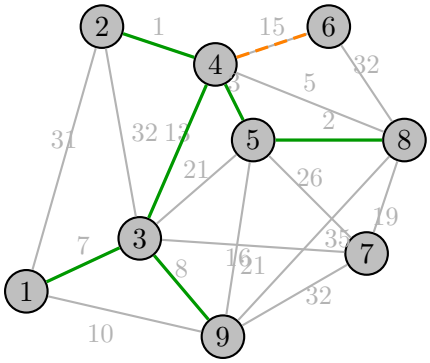


(b) (16) Kandidat $(3, 4)$ (13).

Abbildung 6.23: Kruskal: Schritte 15–16.

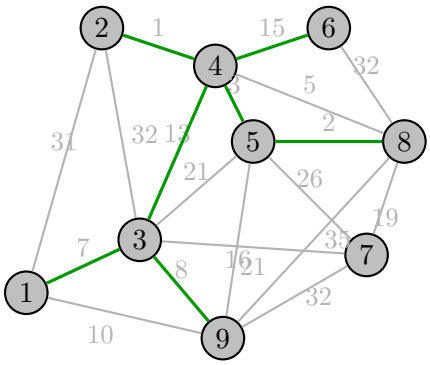


(a) (17) $(3, 4)$ aufgenommen.

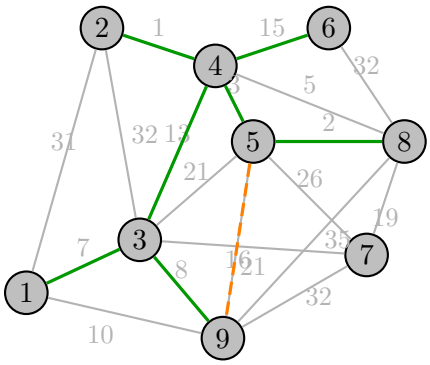


(b) (18) Kandidat $(4, 6)$ (15).

Abbildung 6.24: Kruskal: Schritte 17–18.

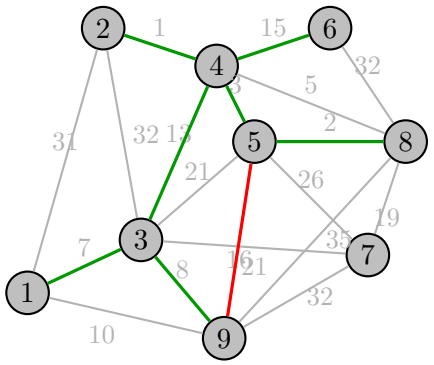


(a) (19) $(4, 6)$ aufgenommen.

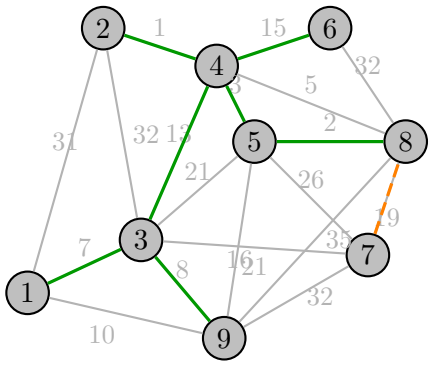


(b) (20) Kandidat $(5, 9)$ (16).

Abbildung 6.25: Kruskal: Schritte 19–20.



(a) (21) $(5, 9)$ verworfen (Kreis).



(b) (22) Kandidat $(7, 8)$ (19).

Abbildung 6.26: Kruskal: Schritte 21–22.

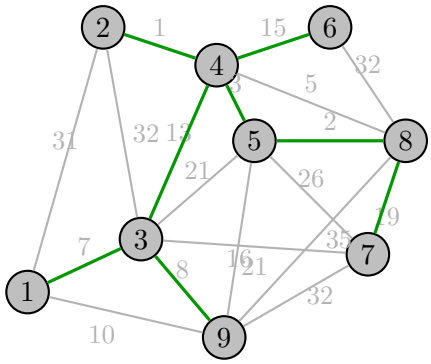


Abbildung 6.27: (23) $(7, 8)$ aufgenommen – MST vollständig (8 Kanten).

Zur effizienten Implementierung des Algorithmus von Kruskal wird häufig die Datenstruktur *Union-Find* verwendet, die es erlaubt, die Zugehörigkeit eines Knotens zu einer Komponente in nahezu konstanter Zeit zu bestimmen und zwei Komponenten in nahezu konstanter Zeit zu vereinigen. Dadurch kann die Kreisfreiheit effizient überprüft werden. Für nähere Informationen zur Union-Find-Datenstruktur sei jedoch auf die Literatur verwiesen.

6.3.2 Algorithmus von Prim

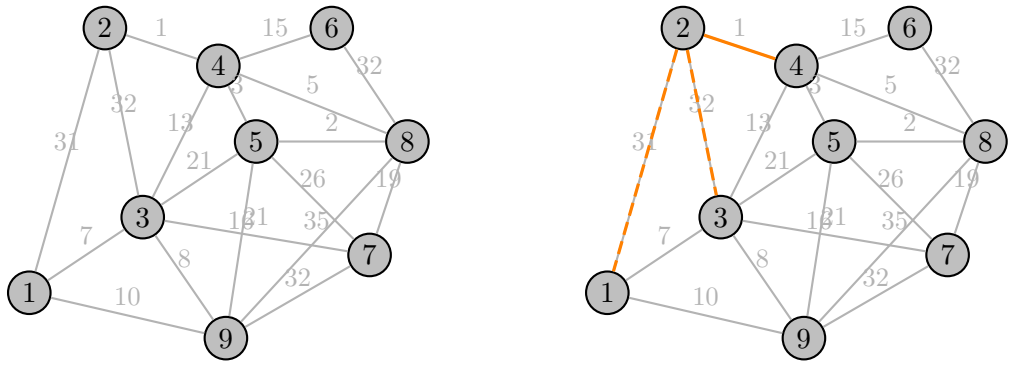
Der Algorithmus von Prim konstruiert ebenfalls einen MST, geht dabei allerdings anders vor. Es wird mit einem beliebigen Knoten begonnen, der als Teil des MST markiert wird. In jedem Schritt wird die Kante mit dem kleinsten Gewicht zum bestehenden Teilbaum

hinzugefügt, wobei jeweils alle Kanten betrachtet werden, die von einem Knoten des MST-Teilbaumes zu einem noch nicht in den Teilbaum aufgenommenen Knoten verlaufen.

Algorithmus 20: Algorithmus von Prim

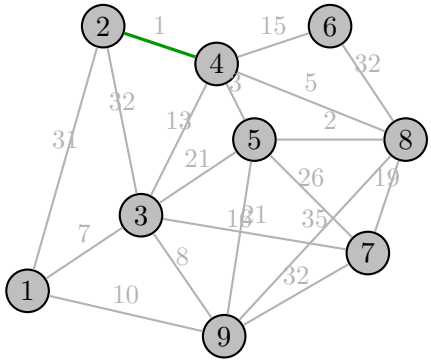
Function *Prim(Graph G)*
- **Result:** Minimum Spanning Tree T
- $E(T) = \emptyset$;
- $V(T) =$ ein zufaellig gewaehlter Startknoten;
- **while** $V(T) \subset V(G)$ **do**
  - Sei E' die Menge aller Kanten zwischen Knoten
  - aus $V(T)$ und $V(G) \backslash V(T)$
  - $e' = \text{argmin}_{e \in E'} w(e)$; // Kante mit kleinstem Gewicht
  - $E(T) = E(T) \cup e'$;
  - Füge neuen Knoten von e' zu $V(T)$ hinzu;

Beispiel 6.2: Wir betrachten die Konstruktion des MSTs mit dem Algorithmus von Prim auf einem gegebenen Graphen Schritt für Schritt. Es werden wieder die in jedem Schritt betrachteten Kanten orange strichliert dargestellt, wobei die Kante mit (von diesen Kanten) geringstem Kantengewicht durchgezogen dargestellt wird. Die grünen Kanten stellen die aktuell in den MST-Teilbaum aufgenommenen Kanten dar.

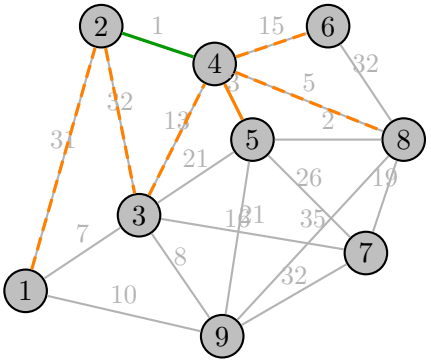


(a) (1) Ausgangszustand. (b) (2) Kandidaten, (2,4) minimal.

Abbildung 6.28: Prim: Schritte 1–2.

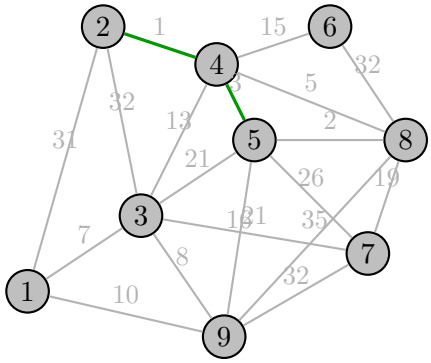


(a) (3) (2,4) aufgenommen.

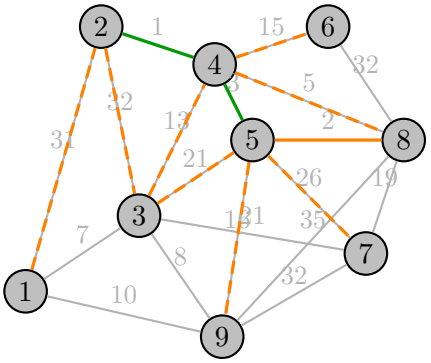


(b) (4) Neue Kandidaten, (4,5) minimal.

Abbildung 6.29: Prim: Schritte 3–4.

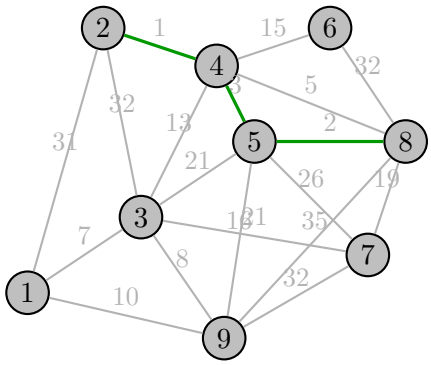


(a) (5) (4,5) aufgenommen.

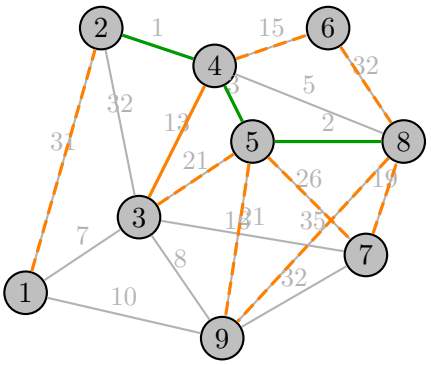


(b) (6) (5,8) minimal.

Abbildung 6.30: Prim: Schritte 5–6.

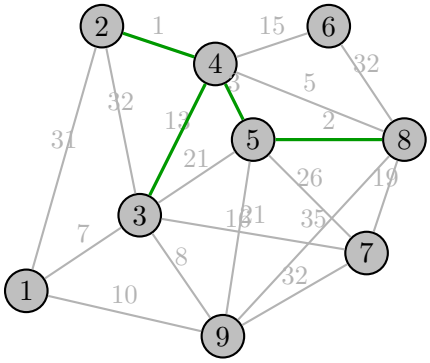


(a) (7) (5,8) aufgenommen.

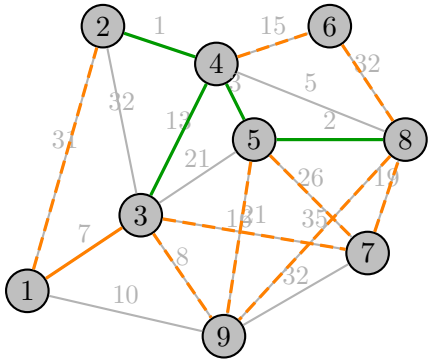


(b) (8) (3,4) minimal.

Abbildung 6.31: Prim: Schritte 7–8.

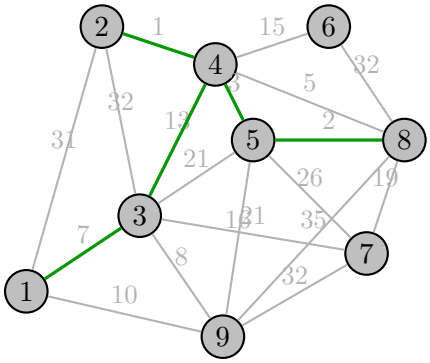


(a) (9) (3,4) aufgenommen.

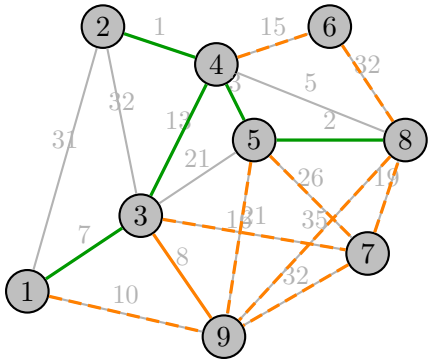


(b) (10) (1,3) minimal.

Abbildung 6.32: Prim: Schritte 9–10.

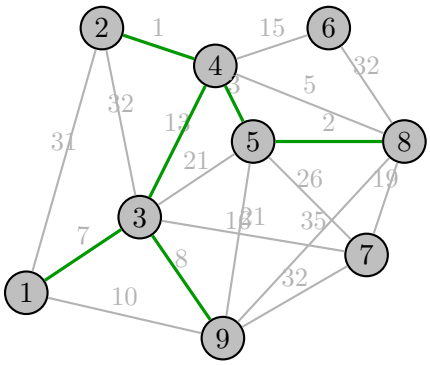


(a) (11) (1,3) aufgenommen.

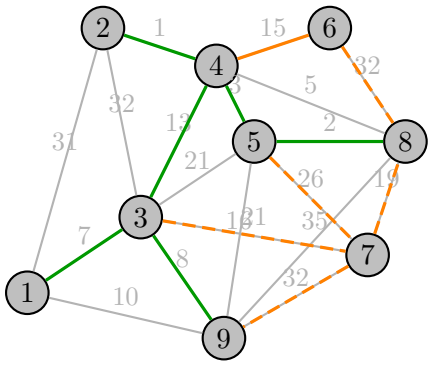


(b) (12) (3,9) minimal.

Abbildung 6.33: Prim: Schritte 11–12.

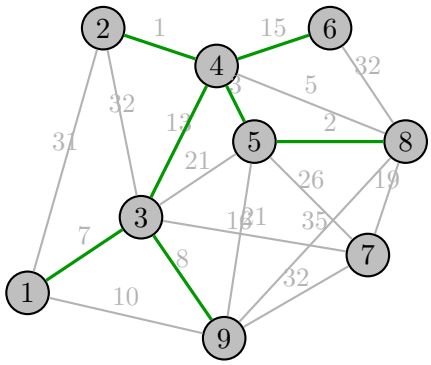


(a) (13) (3,9) aufgenommen.

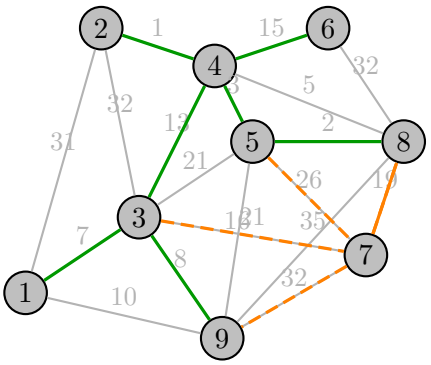


(b) (14) (4,6) minimal.

Abbildung 6.34: Prim: Schritte 13–14.



(a) (15) (4,6) aufgenommen.



(b) (16) (7,8) minimal.

Abbildung 6.35: Prim: Schritte 15–16.

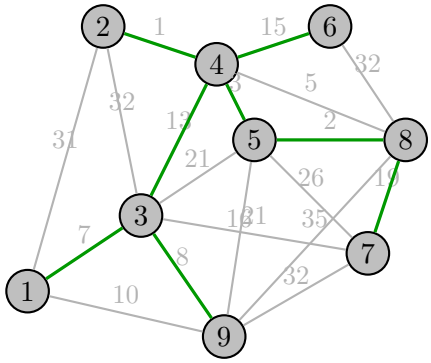


Abbildung 6.36: (17) (7,8) aufgenommen – MST vollständig.

6.3.3 Vergleich

- Beide Algorithmen (Kruskal und Prim) finden den MST in gewichteten Graphen.
- Bei *dicht*[4] besetzten Graphen ist der Algorithmus von Prim jedoch effizienter.
- Bei *dünn*[5] besetzten Graphen ist Kruskal besser.
- Beim Algorithmus von Kruskal müssen die Kanten vorab sortiert werden, was bei vielen Kanten einen erheblichen Aufwand darstellt (sogar den Hauptaufwand des gesamten Verfahrens).

[4] $|E| \in O(|V|^2)$, d.h. die Anzahl der Kanten liegt in der Größenordnung der Anzahl der Kanten eines vollständigen Graphen.

[5] $|E| \in O(|V|)$, d.h. die Anzahl der Kanten liegt in der Größenordnung der Anzahl der Knoten; der Graph enthält also relativ wenige Kanten.

- Bei vergleichsweise wenigen Kanten überwiegen jedoch die Vorteile der einfacheren darauffolgenden Schritte.

Der MST kann also *effizient* (mit einem Polynomialzeit-Algorithmus) gefunden werden. Allerdings stellt er nur das "einfachste" Baum-Problem dar. In praktischen Anwendungen (wie beispielsweise dem Design von Kommunikationsnetzwerken, Infrastruktur) treten oft wesentlich komplexere Problemstellungen auf, die dann Algorithmen wie in Kapitel 8 beschrieben, erfordern.

6.4 Chinese Postman Problem

In Abschnitt 2.3.2 wurden (geschlossene) Eulersche Linien mit ihrer Bedeutung für die Abfallentsorgung, Schneeräumung und Briefzustellung betrachtet. Die Modellierung ist allerdings nicht direkt anwendbar, da reale Straßengraphen nicht unbedingt Eulersch sind. Ebenso spielt die Länge der einzelnen Straßen eine Rolle!

Dies führt zu der bekannten Erweiterung des Problems, die unter dem Namen *Chinese Postman Problem* bekannt ist.[6]

Definition 6.3 (Chinese Postman Problem)**:** Gegeben sei ein gewichteter Graph (nicht notwendigerweise Eulersch). Gesucht wird ein kürzester Zyklus[7] im Graphen, der jede Kante *mindestens* einmal enthält.

Der Algorithmus verwendet das Konzept der *Paarung* (*Matching*), d.h. 2-elementigen Teilmengen ("Paaren") von Knoten.

Definition 6.4 (Paarung/Matching)**:** Gegeben sei ein Graph $G = (V, E)$. Eine Menge $M \subseteq E$ heißt *Matching* (oder Paarung), wenn kein Knoten aus V zu mehr als einer Kanten aus M inzident ist.

- Paarungen heißen **vollständig**, wenn alle Knoten gepaart sind.

[6] Der Name Chinese Postman Problem stammt von der ursprünglichen Fragestellung, die die optimale Route für einen Briefträger in einem Stadtgebiet betraf und wurde vom chinesischen Mathematiker Mei Ko Kwan (*1934) formuliert.

[7] Zyklus, hier im Sinne eines geschlossenen Kantenzuges.

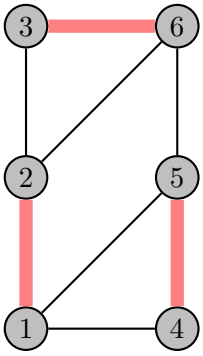


Abbildung 6.37: Beispiel eines vollständigen Matchings (rote Kanten) in einem ungewichteten Graphen.

- In gewichteten Graphen sind meist Paarungen minimalen oder maximalen Gewichts von Interesse, wobei
$$w(M) = \sum_{e \in M} w(e)$$

Ein zentraler Schritt bei der Lösung des Chinese Postman Problem auf einem nicht Eulerschen Graphen ist das Auffinden einer geeigneten Paarung (Matching) der Knoten mit ungeradem Grad, um durch das Äuffüllen"(Duplizieren) bestimmter Kanten einen Eulerschen Graphen zu erzeugen.

Definition 6.5 (Perfekte und minimale Paarung)**:** Sei $G = (V, E)$ ein Graph. Ein *Matching* ist eine Kantenmenge $M \subseteq E$, so dass kein Knoten aus V in mehr als einer Kante von M vorkommt. Ein Matching heißt *vollständig* (perfekt), wenn alle Knoten von V darin vorkommen. In gewichteten Graphen interessiert häufig ein Matching minimalen (oder maximalen) Gewichts mit

$$w(M) = \sum_{e \in M} w(e).$$

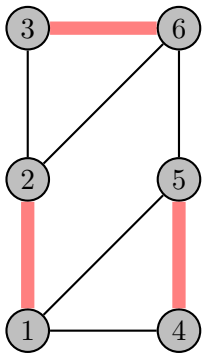


Abbildung 6.38: Beispiel eines vollständigen Matchings (rote Kanten) in einem ungewichteten Graphen.

Algorithmus

1. Bestimme die Menge O der Knoten mit ungeradem Grad. (Ein Graph ist genau dann Eulersch, wenn $O = \emptyset$.)
2. Baue den vollständigen Graphen K auf O; Kantenkosten: Länge eines kürzesten Weges der beiden Knoten im ursprünglichen Graphen.
3. Finde ein kostenminimales perfektes Matching M in K.
4. Dupliziere für jede Matching-Kante die Kanten des entsprechenden kürzesten Weges im ursprünglichen Graphen.
5. Der resultierende Multigraph besitzt nur noch Knoten geraden Grades und ist somit Eulersch; ein Euler-Zyklus liefert eine optimale Lösung des Chinese Postman Problem.

Im Folgenden illustrieren wir diese Schritte an einem Beispiel.

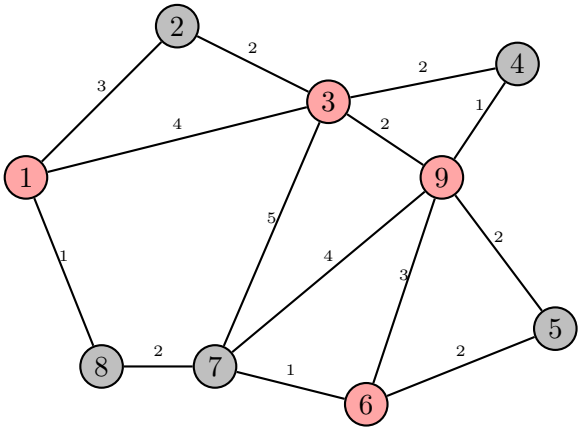


Abbildung 6.39: Ausgangsgraph: rot markierte (hier: hervorgehobene) Knoten besitzen ungeraden Grad. Der Graph ist nicht Eulersch.

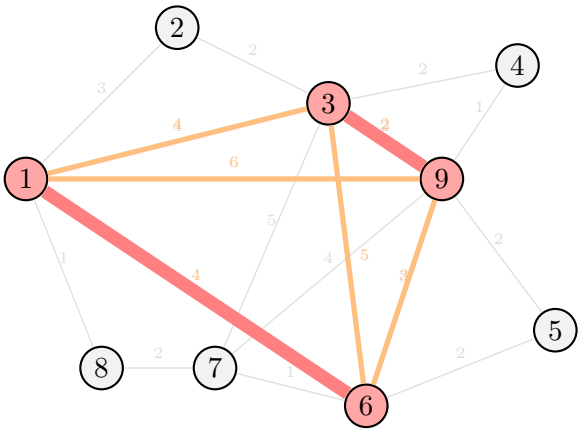


Abbildung 6.40: Vollständiger Graph auf den ungeraden Knoten mit Kantengewichten (orange). Hervorgehoben: ein kostenminimal ausgewähltes Matching.

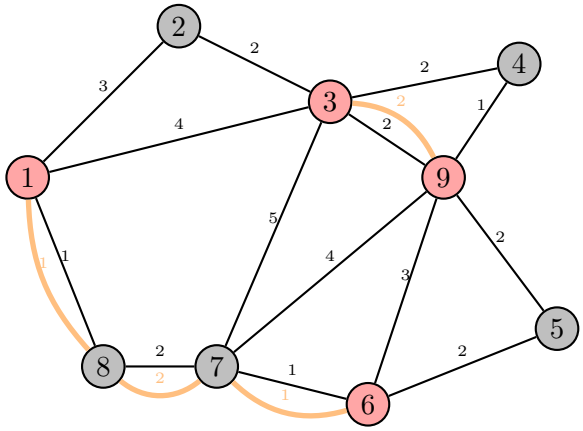


Abbildung 6.41: Duplizierte Kanten entlang der kürzesten Wege der Matching-Knoten: der resultierende Multigraph ist Eulersch.

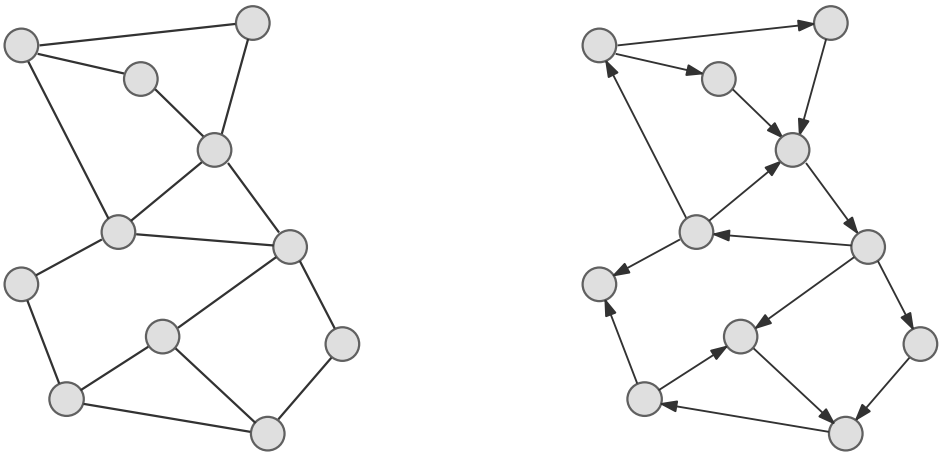
Damit ist der Graph eulersch und ein Euler-Zyklus liefert eine optimale Lösung für das Chinese Postman Problem auf dem Ausgangsgraphen.

6.5 Starke Zusammenhangskomponente

Definition 6.6 (Zusammenhangskomponenten)**:** In einem ungerichteten Graphen G ist eine *Zusammenhangskomponente* ein maximaler, zusammenhängender Teilgraph. Zwei Knoten u und v sind in der selben Zusammenhangskomponente genau dann, wenn $u \rightsquigarrow v$.[8]

Wie kann der Begriff der *Zusammenhangskomponente* auf gerichtete Graphen übertragen werden? Siehe dazu Abbildung 6.42. Die Antwort auf diese Frage ist in Abbildung 6.43 skizziert, und wird in den folgenden Definitionen näher erläutert.

[8] $u \rightsquigarrow v$ bedeutet, dass v von u aus erreichbar ist. Im ungerichteten Graphen impliziert dies $v \rightsquigarrow u$.



(a) Beispiel 1 (b) Beispiel 2

Abbildung 6.42: Beispielgraphen zu Zusammenhang.

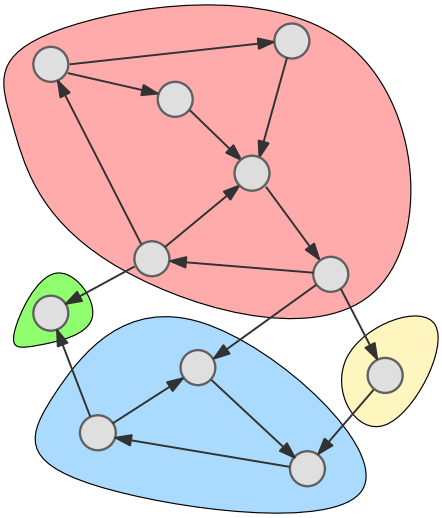


Abbildung 6.43: Starke Zusammenhangskomponenten im Beispielgraphen.

Definition 6.7 (Schwacher Zusammenhang)**:** Man spricht von *schwachem Zusammenhang* in einem gerichteten Graphen, wenn der zugrunde liegende ungerichtete Graph ("Schatten") zusammenhängend ist.

Definition 6.8 (Starke Zusammenhangskomponente)**:** Eine *Starke Zusammenhangskomponente (SZK)* ist eine maximale Teilmenge an Knoten $U \subseteq V$ mit der Eigenschaft, dass für alle Paare $u, v \in U$ sowohl $u \rightsquigarrow v$ als auch $v \rightsquigarrow u$ gilt.

6.5.1 Algorithmus

Es existiert ein verblüffend einfacher Algorithmus zur Berechnung der SZK, wobei jedoch das Funktionsprinzip nicht ganz offensichtlich ist. Wir stellen zunächst den Algorithmus vor, und behandeln anschließend in Abschnitt 6.5.2 die Korrektheit.

Definition 6.9 (Transponierter Graph G^T)**:** Sei $G^T = (V, A^T)$ der Graph der aus dem gerichteten Graphen $G = (V, A)$ entsteht indem alle gerichteten Kanten umgedreht werden, also $A^T = \{(j, i) \mid (i, j) \in A\}$.

Der Algorithmus zur Berechnung der SZK verwendet intern die Tiefensuche (DFS). Dabei ist zu beachten, dass ein Aufruf der Tiefensuche nicht unbedingt alle Knoten besucht. Unter DFS(G) verstehen wir also den gegebenenfalls wiederholten Aufruf der Tiefensuche, solange bis alle Knoten im Graphen besucht sind. Natürlich gilt, dass bereits besuchte Knoten nicht erneut besucht werden. Speichert man in jedem Schritt den Verweis auf den Vorgängerknoten als gerichtete Kante, so entsteht ein *Tiefensuch-Wald*[9].

Algorithmus von Kosaraju-Sharir.

1. Aufruf von DFS(G) und Bestimmung der Fertigstellungszeiten $\tau_f(v)$.
2. Berechne G^T.
3. Aufruf von DFS(G^T) für Knoten $v \in V$ in absteigender Reihenfolge der Werte $\tau_f(v)$.
4. Die in den einzelnen DFS-Aufrufen in Schritt (3) gefundenen Knotenmengen sind die starken Zusammenhangskomponenten.

Beispiel 6.3: Wir demonstrieren das Vorgehen des Algorithmus nun anhand des Graphen in Abbildung 6.44.

[9] Ein Wald ist ein Graph, dessen zusammenhängende Komponenten Bäume sind.

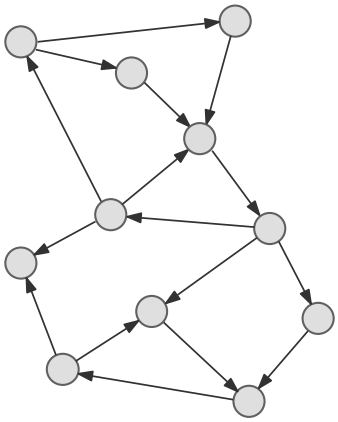


Abbildung 6.44: Beispielgraph zur Bestimmung der starken Zusammenhangskomponenten.

Im ersten Schritt wird die Tiefensuche ausgeführt, und damit die Fertigstellungszeiten $\tau_f(v)$ bestimmt. In diesem Beispiel erfolgt der Aufruf für den Knoten links oben im Graphen. Die Fertigstellungszeiten sind in Abbildung 6.45(a) (als Beschriftung in den Knoten) dargestellt. Im nächsten Schritt werden alle Kanten umgedreht (G^T), siehe Abbildung 6.45(b).

(a) Fertigstellungszeiten τ_f

(b) Transponierter Graph G^T

Abbildung 6.45: Links der Graph G mit den Fertigstellungszeiten als Beschriftung in den Knoten angegeben. Der Erste Aufruf erfolgte für den Knoten links oben. Dadurch erhält er schließlich die Fertigstellungszeit 11. Rechts ist der transponierte Graph G^T dargestellt.

Nun wird die Tiefensuche auf G^T in absteigender Reihenfolge der Fertigstellungszeiten aufgerufen. Man sieht sofort, dass beim Aufruf für den Knoten mit $\tau_f(v) = 11$ die unteren Knoten des Graphen nicht erreicht werden können. Alle in diesem ersten Aufruf erreichbaren Knoten bilden die erste SZK. Der Knoten mit $\tau_f(v) = 5$ ist nicht erreichbar. Für diesen Knoten wird die Tiefensuche erneut aufgerufen, und er bildet alleine eine weitere SZK, da von ihm aus keine weiteren unbesuchten Knoten erreichbar sind. Nach zwei weiteren DFS-Aufrufen sind die vier SZKn ermittelt, siehe Abbildung 6.46.

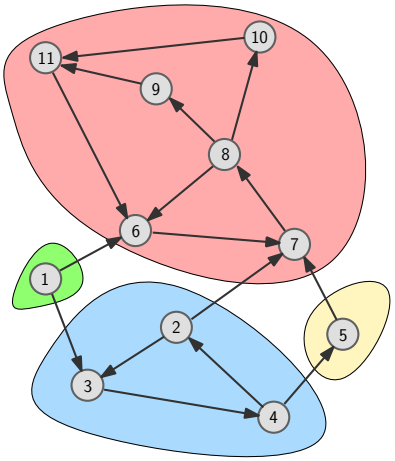


Abbildung 6.46: Gefundene starke Zusammenhangskomponenten.

6.5.2 Korrektheitsbeweis

Da die Korrektheit dieses Algorithmus weniger intuitiv ist, geben wir hier einen formalen Beweis an.

Lemma 6.1: *Sei $G' = (V', E')$ der Graph der daraus entsteht indem in G jede starke Zusammenhangskomponente (SZK) in einen einzelnen Knoten kontrahiert wird. G' ist ein gerichteter azyklischer Graph (DAG)*[10].

Der Graph G' ist in Abbildung 6.47 dargestellt.

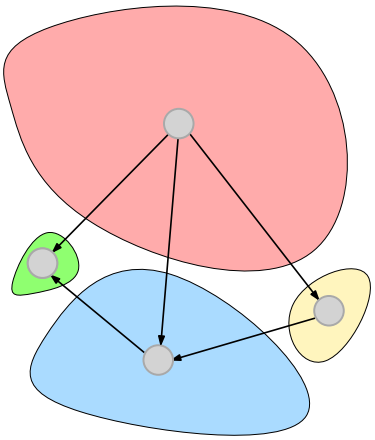
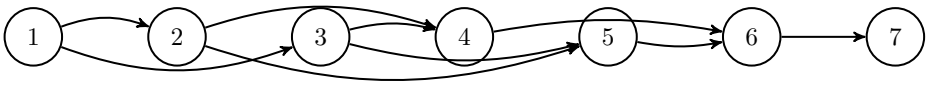


Abbildung 6.47: Kontraktion der SZKn ergibt einen DAG.

Beweis. Angenommen es gibt einen Kreis $C_1, C_2, \ldots, C_n$ mit $C_i \in V'$. Dann existiert für jedes k eine gerichtete Kante von einem Knoten in C_k zu einem in C_{k+1} (sowie von C_n nach C_1). Damit existiert ein Pfad zwischen allen Knoten aller C_i in beide Richtungen und alle gehören zu einer SZK – Widerspruch zur Kontraktion. □

Eigenschaften von DAGs: Ein DAG enthält keinen gerichteten Kreis; seine Knoten lassen sich linear so anordnen, dass alle Kanten nach rechts zeigen (topologische Sortierung). Jeder DAG besitzt mindestens einen Knoten v mit $d^+(v) = 0$ und mindestens einen Knoten mit $d^-(v) = 0$.

[10] DAG: Directed Acyclic Graph



Topologische Sortierung: $1, 2, 3, 4, 5, 6, 7$

Abbildung 6.48: Beispiel einer topologischen Sortierung: Alle Kanten zeigen in eine Richtung.

Lemma 6.2: *Sei C eine SZK von G ohne auslaufende Kanten. Ein DFS-Aufruf für einen Knoten $v \in C$ besucht genau alle Knoten $u \in C$.*

Beweis. Sei v beliebig in C. Für jedes $u \in C$ gilt $v \rightsquigarrow u$. Da keine auslaufende Kante existiert, ist kein weiterer Knoten erreichbar. □

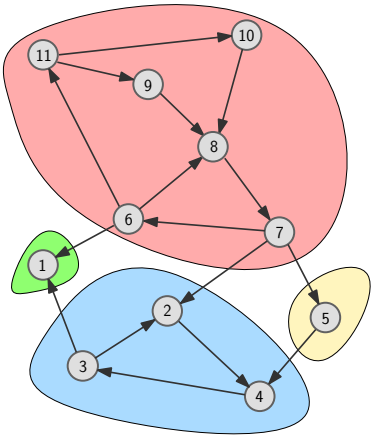


Abbildung 6.49: Illustration zu Lemma 6.2.

Lemma 6.3: *Seien C_1 und C_2 zwei SZK von G und $a = (i, j)$ eine gerichtete Kante mit $i \in C_1$, $j \in C_2$. Sei v^* der erste in C_1 von DFS besuchte Knoten. Dann gilt $\tau_f(v^*) > \tau_f(v_k)$ für alle $v_k \in C_2$.*

Beweis. Wir unterscheiden zwei Fälle (siehe auch Abbildung 6.50).

Fall 1: DFS startet in einem Knoten $v \in C_1$ bevor ein Knoten aus C_2 aufgerufen wird. Dann erreicht DFS von v aus alle Knoten in C_1 und (über a) auch alle in C_2. Der erste abgeschlossene Knoten in C_1 erhält daher eine Fertigstellungszeit größer als alle Knoten in C_2.

Fall 2: DFS startet zuerst in einem Knoten aus C_2. Es gibt keinen Weg von C_2 zurück nach C_1, sonst wären beide Komponenten vereint. Somit werden alle Knoten in C_2 abgeschlossen bevor ein Knoten in C_1 entdeckt wird. Damit ist $\tau_f(v^*)$ wiederum größer als jede Fertigstellungszeit in C_2.

In beiden Fällen folgt die Behauptung. □

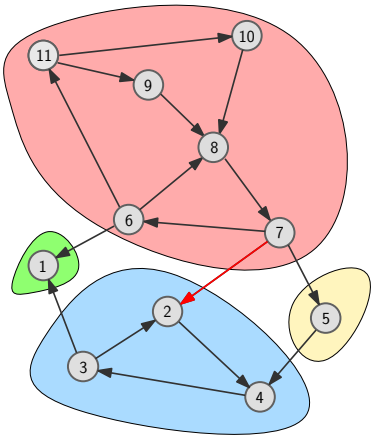


Abbildung 6.50: Situation zweier Komponenten C_1, C_2 mit Kante $C_1 \to C_2$.

Lemma 6.4: *Ein Knoten v mit $\max(\tau_f(v))$ in G gehört zu einer SZK ohne einlaufende Kanten.*

Beweis. Folgt direkt aus Lemma 6.3. □

Lemma 6.5: *Die Knotenmengen der SZKn von G entsprechen jenen in G^T.*

Beweis. Für zwei Knoten u, v in derselben SZK in G gilt $u \rightsquigarrow v$ und $v \rightsquigarrow u$. Durch Umkehr aller Kanten gilt dies ebenso in G^T, also sind u und v auch dort in derselben SZK (und umgekehrt). □

Theorem 6.1: *Die zusammenhängenden Komponenten des Tiefensuch-Waldes in G^T (Schritt 3 des Algorithmus) entsprechen den SZK in G.*

Beweis. Teil 1: Nach Lemma 6.4 wird beim ersten geeigneten Aufruf eine SZK ohne auslaufende Kanten (in G^T) vollständig besucht (Lemma 6.2).

Teil 2: Sei v Start eines späteren DFS-Aufrufs und C_v seine SZK. Angenommen es würde ein noch unbesuchter Knoten $w \notin C_v$ (mit $v \rightsquigarrow w$) erreicht. Dann gäbe es in G einen Pfad von w zurück zu einem Knoten in C_v, sonst wären v und w nicht im selben Aufruf erreichbar. Damit hätte aber ein Knoten aus der SZK von w eine größere Fertigstellungszeit als v (Widerspruch zu der Reihenfolge). Also werden ausschließlich Knoten von C_v besucht. Zusammen mit Lemma 6.5 folgt die Behauptung. □

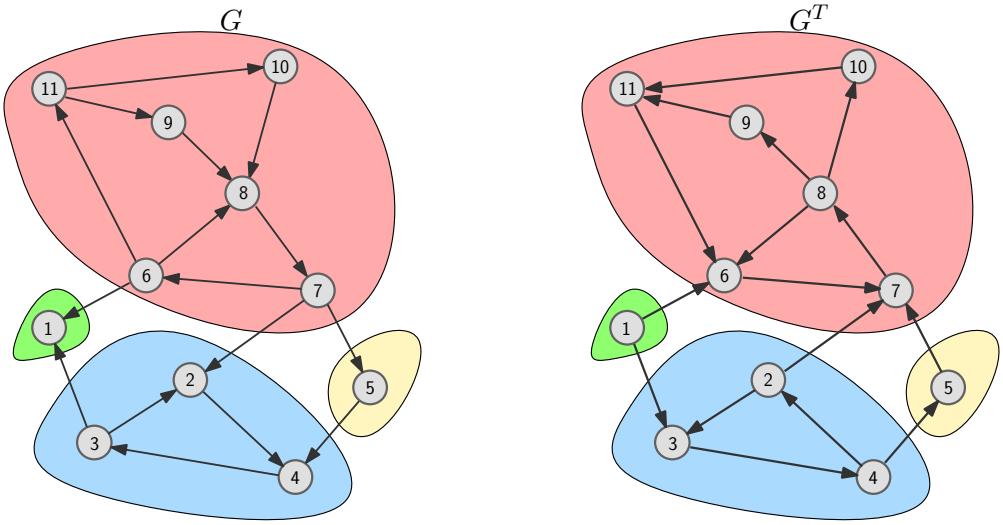


Abbildung 6.51: Relation der SZKn zwischen G und G^T.

Anmerkung. Führt eine Kante in G^T von C_v nach C_w, so mussten im Originalgraphen G alle Knoten in C_w nach jenen in C_v fertiggestellt werden (Kante verläuft in G von C_w nach C_v). Daher erscheinen sie zuvor in der absteigenden Sortierung der τ_f-Werte.

7 Komplexitätstheorie

Warum Komplexitätstheorie?

Die Komplexitätstheorie ermöglicht es, gegebene Probleme in Bezug auf ihre Ressourcenanforderungen zu klassifizieren und zu vergleichen. *Lohnt es sich, weiter nach einem viel schnelleren Algorithmus zu suchen? Wie viel Ressourcen (Zeit / Speicher) braucht ein Problem mindestens?* Komplexitätstheorie liefert eine Landkarte der Schwierigkeit von Problemen.

Ressourcen und Kostenmodelle

Typischerweise sind die **Laufzeit** (Anzahl elementarer Schritte) und der **Speicherplatz** in Abhängigkeit der Eingabelänge n relevant. Wir betrachten asymptotisches Verhalten für großes n.

Entscheidungsprobleme und Sprachen

Zur theoretischen Analyse reduziert man Optimierungs-/Suchprobleme oft auf **Ja/Nein**-Fragen (Entscheidungsprobleme). Beispiel: SAT: "Hat die Formel eine erfüllende Belegung?". Es besteht ein enger Zusammenhang zwischen Entscheidungsproblemen und den zugehörigen Optimierungsproblemen.

Weiters besteht ein enger Zusammenhang zum Gebiet der formalen Sprachen. Zu diesen existieren (formale) Automaten, die deartige Sprachen verarbeiten können. Hierzu seien genannt: Endlicher Automat, Keller-Automat, Turing-Maschine. Diesen Zusammenhang genauer zu betrachten, sprengt jedoch den Rahmen dieses Skriptums bei weitem.

7.1 Komplexitätsklassen

Die Klasse P: Menge aller Entscheidungsprobleme, die ein *deterministischer* Algorithmus in polynomieller Zeit löst (z.B. Sortieren, Kürzeste Wege mit nicht-negativen Gewichten, bipartite Matching). Praktisch: gilt als effizient lösbar.

Die Klasse NP: "Nondeterministic-Polynomial": Probleme, bei denen man eine **gegebene Lösung** (Zertifikat) in polynomieller Zeit prüfen kann. Wichtig: "NP" heißt *nicht* "nicht polynomial" – viele NP-Probleme könnten theoretisch in P liegen. Unbekannt: $P = NP$?

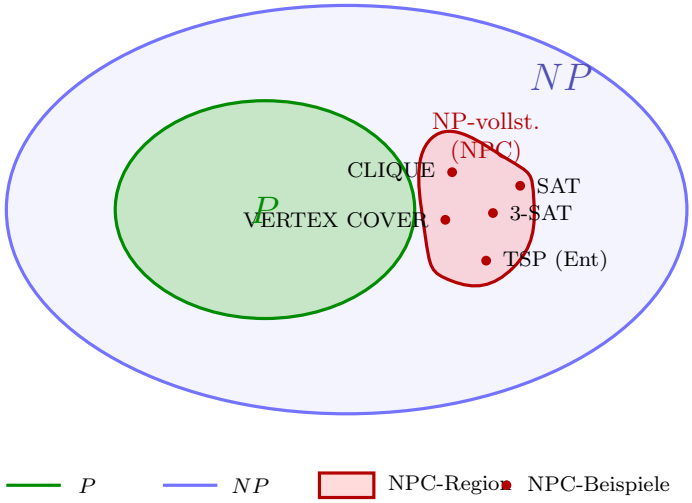


Abbildung 7.1: Beziehung der Klassen: $P \subseteq NP$; NP-vollständige Probleme (NPC) sind die schwersten in NP. Ob einige (dann alle) NPC-Probleme in P liegen, ist genau die offene Frage $P = NP$. Gezeigt wird das Szenario $P \neq NP$ (kein Überlapp).

Eine nach wie vor offene Frage im Bereich der theoretischen Informatik ist, ob $P = NP$. Allgemein wird angenommen, dass $P \neq NP$ gilt. Anderenfalls wären viele der heute schwierigsten Probleme (NP-vollständige Probleme wie SAT, Travelling Salesman, Graph-Färbung usw.) plötzlich effizient lösbar. Praktisch hieße das: Alle Optimierungsprobleme, die bisher nur mit enormem Rechenaufwand oder Näherungsverfahren lösbar sind, könnten in polynomialer Zeit berechnet werden. Aufgrund zu vieler fehlgeschlagener Versuche, $P = NP$ zu beweisen, wird dies von vielen Experten als unwahrscheinlich angesehen.

7.2 NP-schwere und NP-Vollständigkeit

- **NP-schwer**: Mindestens so schwer wie jedes Problem in NP (alle NP-Probleme lassen sich darauf reduzieren).
- **NP-vollständig** (NPC): In NP *und* NP-schwer.

Intuition: NP-vollständige Probleme sind "universelle harte Knoten" der Klasse NP.

Reduktionen (Kernidee): Um zu zeigen, dass neues Problem X schwer ist: Wähle bekanntes NP-vollständiges Problem Y, konstruiere *polynomielle* Abbildung $Y \to X$. Wenn X effizient wäre, wäre auch Y effizient. Damit: X ist mindestens so schwer wie Y.

7.3 Klassische NP-vollständige Probleme (Beispiele)

Beispiele (alle Entscheidungsvarianten):

- SAT (Erfüllbarkeit boolescher Formel)
- 3-SAT (Klauseln mit je 3 Literalen)
- CLIQUE (enthält der Graph eine Clique Größe k?)
- VERTEX COVER (gibt es eine Überdeckungsmenge Größe k?)
- HAMILTONIAN CYCLE / PATH
- PARTITION / SUBSET-SUM
- TSP (Travelling Salesman Problem)

Was bedeutet NP-vollständig in der Praxis? Für das Problem gibt es sehr wahrscheinlich keinen polynomiellen Algorithmus. Für kleine Instanzen (z.B. $n < 100$) sind exakte Verfahren (Backtracking, Branch-and-Bound, SAT-Solver) oft praktikabel. Für größere Instanzen helfen oft Approximationen, Parameterisierung oder Heuristiken.

Weitere Klassen: PSPACE (polynomieller Speicher), EXPTIME (exponentielle Zeit).

7.3.1 Überblick P vs. NP

Klasse	Idee
P	effizient berechenbar
NP	Lösung schnell prüfbar
NPC	härteste Probleme in NP
NP-schwer	mindestens so schwer wie NP, evtl. nicht in NP
Co-NP	Komplemente zu NP-Sprachen
PSPACE	beschränkt durch polynomiellen Speicher

7.4 Typische Beweisstrategie für NP-Vollständigkeit

1. Zeige: Problem X liegt in NP (Zertifikat + Verifizierer in Polyzeit).
2. Wähle bekanntes NP-vollständiges Problem Y.
3. Konstruiere polynomielle Reduktion $Y \to X$.
4. Schluss: X ist NP-vollständig.

7.5 Zusammenfassung

Komplexitätstheorie erklärt Grenzen des Machbaren. Sie hilft zu entscheiden, ob wir (a) nach besserem exakten Algorithmus suchen, (b) approximieren, (c) heuristisch vorgehen oder (d) Problem einschränken. Kernmerksatz: *NP-vollständig* heißt sehr wahrscheinlich: "Keine allgemeine schnelle exakte Lösung – such Alternativen."

8 Optimierungsalgorithmen

8.1 Optimierungsprobleme

Viele algorithmische Fragestellungen verlangen nicht nur *irgendeine* zulässige Lösung ("feasible solution"), sondern eine *beste* gemäß eines vorgegebenen Gütemaßes. Solche Aufgaben nennt man **Optimierungsprobleme**. Typische Beispiele sind das Kürzeste-Wege-Problem (minimiere Distanz), das Rucksackproblem (maximiere Wert bei Gewichtsgrenze) oder Tourenplanungsprobleme wie das Traveling Salesperson Problem (TSP: minimiere Gesamtlänge einer Rundtour).

Grundbausteine. Ein (diskretes) Optimierungsproblem lässt sich abstrakt durch folgende Komponenten modellieren:

- **Instanzmenge** I: Jede Instanz $x \in I$ beschreibt die konkreten Eingabedaten (Graph, Gewichte, Kapazitäten, ...).
- **Lösungsraum** $S(x)$: Menge aller *zulässigen* (feasible) Lösungen für Instanz x (z.B. alle einfachen s-t-Wege, alle Teilmengen von Gegenständen, alle Permutationen der Städte).
- **Zielfunktion** (Kosten- oder Nutzenfunktion) $f_x : S(x) \to \mathbb{R}$, die jeder zulässigen Lösung einen Wert zuordnet.
- **Optimierungsrichtung**: Minimierung oder Maximierung.

Das Ziel ist: Finde für gegebene Instanz x eine Lösung $s^* \in S(x)$ mit

$$f_x(s^*) = \begin{cases} \min\limits_{s \in S(x)} f_x(s) & \text{(Minimierung)} \\ \max\limits_{s \in S(x)} f_x(s) & \text{(Maximierung).} \end{cases}$$

Formale Standard-Schreibweise. Häufig schreibt man kompakt:

$$\min\{f_x(s) : s \in S(x)\}, \quad \text{bzw.} \quad \max\{f_x(s) : s \in S(x)\}.$$

Die Nebenbedingungen (Restriktionen) sind dabei implizit in der Definition von $S(x)$ kodiert.

Beispiel (Rucksackproblem / Knapsack). Gegeben Gegenstände $1, \dots, n$ mit Gewichten w_i und Werten v_i sowie Kapazität W. Gesucht: Teilmenge $S \subseteq \{1, \dots, n\}$ mit

$$\sum_{i \in S} w_i \leq W \quad \text{und maximalem Wert} \quad \sum_{i \in S} v_i.$$

Hier ist $S(x)$ die Menge aller Teilmengen, die Kapazitätsbeschränkung modelliert die Zulässigkeit, und die Zielfunktion ist $f_x(S) = \sum_{i \in S} v_i$ (Maximierung).

Beispiel (Kürzeste Wege). Instanz: gerichteter Graph $G = (V, E)$ mit nicht-negativen Kantenlängen $\ell : E \to \mathbb{R}_{\geq 0}$ und zwei Knoten s, t. Lösungen: einfache Wege P von s nach t. Zielfunktion: $f_x(P) = \sum_{e \in P} \ell(e)$, Minimierung. Algorithmen wie Dijkstra liefern ein optimales P effizient für nicht-negative Längen.

Optimierung vs. Entscheidungsproblem. Für viele Optimierungsprobleme betrachtet man die *Entscheidungsvariante*, z.B.:

SHORTEST-PATH-DEC: Gibt es einen s-t-Weg der Länge $\leq K$?

Die Entscheidungsform ist wichtig in der Komplexitätstheorie (NP-Vollständigkeit). Ein *Optimierer* kann oft durch wiederholtes Lösen der Entscheidungsvariante (z.B. binäre Suche auf K) konstruiert werden, falls die Struktur das zulässt.

Qualität und Approximation. Ist exaktes Lösen schwer (z.B. NP-schwer), untersucht man Näherungsverfahren. Für eine Minimierungsaufgabe nennt man eine zulässige Lösung s *α-approximativ*, wenn

$$f_x(s) \leq \alpha \cdot f_x(s^*) \quad (\alpha \geq 1).$$

Analog bei Maximierung: $f_x(s) \geq f_x(s^*)/\alpha$. (Mehr dazu in späteren Abschnitten.)

Kontinuierliche Optimierung. Neben diskreten (kombinatorischen) Problemen existieren stetige Varianten (z.B. Minimierung glatter Funktionen $f : \mathbb{R}^n \to \mathbb{R}$ unter Gleichungs-/Ungleichungsrestriktionen). Dort kommen Ableitungen, Gradienten, Konvexität ins Spiel. In diesem Skript fokussieren wir primär diskrete Beispiele; grundlegende Begriffe (Konvexität, Gradient) können bei Bedarf kurz eingeführt werden.

Ausgearbeitetes Beispiel 1 (Kürzeste Wege – detailliert). Wir gehen das bekannte Kürzeste-Wege-Problem noch einmal sehr systematisch durch und benennen die Bausteine – so kann dieses Schema später auf andere Probleme übertragen werden.

Instanz: Gerichteter Graph $G = (V, E)$, Startknoten $s \in V$, Zielknoten $t \in V$, nicht-negative Kantengewichte $\ell : E \to \mathbb{R}_{\geq 0}$.
Lösungen: Alle *einfachen* Wege $P = (v_0 = s, v_1, \ldots, v_k = t)$ mit $(v_{i-1}, v_i) \in E$. Warum dürfen wir uns auf einfache Wege beschränken? Weil ein Zyklus mit nicht-negativen Längen die Gesamtdistanz nur vergrößert – man kann ihn also weglassen ohne schlechter zu werden.
Zielfunktion (Minimierung):

$$f_x(P) = \sum_{i=1}^{k} \ell(v_{i-1}, v_i).$$

Alternative Formulierung als Integer-Programm (Flow-Modell): Wir definieren eine binäre Variable x_e für jede Kante $e \in E$, die 1 ist, falls die Kante im Weg liegt.
Optimales Ziel: $\text{dist}(s, t) = \min_{P \in S(x)} f_x(P)$ – dies ist der kürzeste Abstand zwischen s und t.
Entscheidungsvariante: Existiert ein s-t-Weg mit Gesamtlänge $\leq K$? (Parameter K ist eine vorgegebene Schranke.)
Rekonstruktionsprinzip (Algorithmischer Kern): Dijkstras Algorithmus berechnet für jeden Knoten v einen Wert $d[v]$ (bisher bekannte kürzeste Distanz) und einen Vorgänger $pred[v]$. Sobald t fest verarbeitet ist, laufen wir rückwärts über die Vorgängerzeiger und erhalten so einen optimalen Weg.

$$\begin{aligned}
&\min \sum_{e \in E} \ell(e) x_e \\
&\text{s.t.} \sum_{(u,v) \in E} x_{(u,v)} - \sum_{(v,w) \in E} x_{(v,w)} = \begin{cases} 1, & v = s \\ -1, & v = t \\ 0, & \text{sonst} \end{cases} && \forall v \in V \\
&\quad x_e \in \{0, 1\} && \forall e \in E
\end{aligned}$$

Die Flussbilanz stellt sicher, dass genau ein Pfad von s nach t entsteht. Mit nicht-negativen Gewichten ist die LP-Relaxation (nur $0 \leq x_e \leq 1$) bereits *integral*, d.h. jede optimale Basislösung ist automatisch ganzzahlig (Eigenschaft von Einheits-Flussproblemen).
Kleines Zahlenbeispiel: $V = \{A, B, C, D, E\}$, $s = A$, $t = E$.

Kante	Gewicht
$A \to B$	2
$A \to C$	5
$B \to C$	1
$B \to D$	2
$C \to D$	2
$C \to E$	6
$D \to E$	1

Einige Wege (mit Summen): $A \to B \to D \to E$ (2+2+1=5), $A \to B \to C \to D \to E$ (2+1+2+1=6), $A \to C \to D \to E$ (5+2+1=8), $A \to B \to C \to E$ (2+1+6=9). Die beste (kürzeste) Variante ist somit $A \to B \to D \to E$ mit Kosten 5.

Ausgearbeitetes Beispiel 2 (0/1-Knapsack – detailliert). Auch hier wieder Bausteine und mehrere Lösungsansätze.

Instanz: n Gegenstände mit Gewichten $w_1, \ldots, w_n \in \mathbb{N}$, Werten $v_1, \ldots, v_n \in \mathbb{R}_{>0}$ und Kapazität $W \in \mathbb{N}$.
Lösungen: Teilmengen $S \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in S} w_i \leq W$.
Zielfunktion (Maximierung): $f_x(S) = \sum_{i \in S} v_i$ – wir wollen den Gesamtwert maximieren.
Entscheidungsvariante: Gibt es eine Teilmenge S mit $\sum_{i \in S} w_i \leq W$ und Wert $\geq K$?
ILP-Modell:

$$\begin{aligned} \max \ & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} \ & \sum_{i=1}^{n} w_i x_i \leq W \\ & x_i \in \{0,1\} \qquad\qquad i = 1, \ldots, n \end{aligned}$$

Variable $x_i = 1$ bedeutet: Gegenstand wird eingepackt.

Dynamische Programmierung (DP): Definiere $DP[i][c]$ = maximaler Gesamtwert mit den ersten i Gegenständen und Kapazität c. Rekurrenz (Fallunterscheidung: Gegenstand i weglassen oder nehmen, falls er hineinpasst):

$$DP[i][c] = \max\big(DP[i{-}1][c],\ v_i{+}DP[i{-}1][c{-}w_i]\big) \quad \text{falls } w_i \leq c, \text{ sonst } DP[i][c] = DP[i{-}1][c].$$

Komplexität $O(nW)$ – pseudo-polynomiell, weil sie linear in W (der Zahlenwert) ist.
Kleines Zahlenbeispiel:

Gegenstand	Gewicht w_i	Wert v_i
1	3	4
2	4	5
3	2	3
4	5	8
5	1	2

Kapazität $W = 8$

Wir testen einige gültige Kombinationen (Gesamtgewicht ≤ 8):

- $\{4, 3, 5\}$: Gewicht 5+2+1=8, Wert 8+3+2=13
- $\{2, 3, 5\}$: Gewicht 4+2+1=7, Wert 5+3+2=10
- $\{1, 3, 5\}$: Gewicht 3+2+1=6, Wert 4+3+2=9
- $\{1, 2\}$: Gewicht 3+4=7, Wert 4+5=9
- $\{1, 4\}$ ist zu schwer (3+5=8 geht noch) mit Wert 12 (auch möglich) – hier also Kandidat mit Wert 12

Beste gefundene Kombination: $\{4, 3, 5\}$ mit Wert 13 (optimal).

Greedy Hinweis: Sortiert man nach Wert/Gewicht (Profit-Dichte), erhält man Reihenfolge (4:1.6), (5:2), (3:1.5), (1:1.33), (2:1.25). Greedy würde so (5,4,3) oder (4,5,3) wählen (je nach Sortierungstie-break) und kommt hier zufällig zum Optimum – aber das klappt nicht immer (Greedy ist für 0/1-Knapsack generell kein optimaler Algorithmus).

Zusammenfassung. Ein Optimierungsproblem ist vollständig beschrieben durch (Instanz, Lösungsraum, Zielfunktion, Richtung). Algorithmisches Ziel ist eine effiziente Bestimmung (oder Annäherung) einer optimalen Lösung sowie Analyse von Laufzeit, Korrektheit und ggf. Approximationsgarantien.

8.2 Approximationsalgorithmen

Bei NP-schweren Problemen haben exakte Algorithmen (unter $P \neq NP$) eine exponentielle Laufzeit, und sind deshalb für größere Probleminstanzen nicht praktikabel. Approximationsalgorithmen liefern schnell *gute* (qualitätsgarantierte) Lösungen. Für eine Minimierungsaufgabe heißt Algorithmus A ein α-*Approximationsalgorithmus* ($\alpha \geq 1$),

wenn für jede Instanz x gilt: $f_A(x) \leq \alpha \cdot OPT(x)$. (Bei Maximierung: $f_A(x) \geq OPT(x)/\alpha$.) Konstante α unabhängig von $|x|$.

Beispiel (0/1-Knapsack) statt Technik-Katalog. Wir betrachten das klassische 0/1-Knapsack-Problem: Gewichte $w_i > 0$, Werte $v_i > 0$, Kapazität W. Gesucht ist eine Teilmenge maximalen Gesamtwertes mit Gesamtgewicht $\leq W$.

Eine sehr einfache Approximationsidee ("Take-Largest-Value-or-Greedy-Fraktion") liefert schon einen Faktor 2.

Algorithmus (Wert/gewicht-Greedy + Vergleich mit bestem Einzelgegenstand):

1. Sortiere Gegenstände nach absteigender Dichte v_i/w_i.
2. Packe in dieser Reihenfolge, solange der nächste hineinpasst.
3. Merke dir zusätzlich den Gegenstand mit *größtem Einzelwert* v_{max}.
4. Gib die bessere der beiden Lösungen zurück: (a) die gefüllte Greedy-Liste oder (b) nur den Gegenstand mit Wert v_{max}.

Beweis der 2-Approximation (Skizze): Sei OPT der optimale Wert. Betrachte die erste Position, an der Greedy einen Gegenstand j nicht mehr einpacken kann. Dann ist die bisher gepackte Menge G eine zulässige Lösung. Die optimale Lösung besteht aus vollständig genommenen Gegenständen; vergleiche deren Werte mit der (nach Dichte) mindestens so effizienten Folge von Greedy vor Abbruch. Man zeigt

$$OPT \leq v(G) + v_j \leq v(G) + v_{max} \leq 2\max\{v(G), v_{max}\}.$$

Damit ist der ausgegebene Wert $\geq OPT/2$.

Kleines Beispiel:

Gegenstand	A	B	C	D
w_i	4	5	2	3
v_i	20	18	7	8
v_i/w_i	5.0	3.6	3.5	2.67

Kapazität $W = 9$. Sortiert nach Dichte: A (5.0), B (3.6), C (3.5), D (2.67). Greedy packt A ($w = 4$), kann B nicht mehr (würde 9¿W? $4 + 5 = 9$ passt exakt, also nimmt B; alternativ wähle Beispiel so, dass Bruch entsteht – wir modifizieren $w_B = 6$): Nehmen wir $w_B = 6$ (sonst kein Bruch). Dann passt B nicht mehr ($4 + 6 = 10 > 9$), packt danach C ($4 + 2 = 6$) und D ($6 + 3 = 9$). Greedy-Lösung: A,C,D mit Wert $20 + 7 + 8 = 35$. Bester

Einzelgegenstand hat Wert 20 (A). Optimal wäre hier tatsächlich A,C,D=35, also Faktor 1.

Der Algorithmus ist sehr einfach, garantiert aber nie schlechter als Faktor 2.

Verhältnis interpretieren. Optimum = 100, Lösung = 130 (Minimierung) bedeutet Verhältnis 1.3. Bei Maximierung würde man $100/130 \approx 0.77$ vergleichen oder sagen: 1.3-Approximationsfaktor.

Für manche Probleme lässt sich beweisen, dass kein Polynomialzeit-Algorithmus ein besseres Verhältnis als eine bestimmte Schranke erreichen kann – es sei denn $P = NP$. Zum Beispiel lässt sich *Set Covering* nicht besser als $(1 - \varepsilon) \ln n$ approximieren.

In der praktischen Anwendung bieten Approximationsalgorithmen eine Sicherheitsgarantie, nicht mehr als einen bestimmten Faktor vom Optimum entfernt zu sein. Dies ist besonders wichtig in Situationen, in denen exakte Lösungen zu teuer oder zeitaufwendig sind. Dennoch lassen sich in der Praxis oft bessere Lösungen durch *Heuristiksche Optimierungsverfahren* erzielen. Diese sind Gegenstand des nächsten Abschnittes.

8.3 Heuristische Optimierungsverfahren

Heuristiken verzichten auf strenge Gütegarantien zugunsten einfacher Implementierung, Geschwindigkeit oder besserer durchschnittlicher Lösungen. Sie sind zentral bei sehr großen oder komplexen Instanzen (Routing, Produktionsplanung, Bioinformatik). Nachfolgend kompakte, aber etwas ausführlichere Kernverfahren.

Die Herausforderung bei der Lösung von Problemen mit stark *multimodalen* Lösungsräumen besteht darin, *lokalen Optima* zu entkommen.

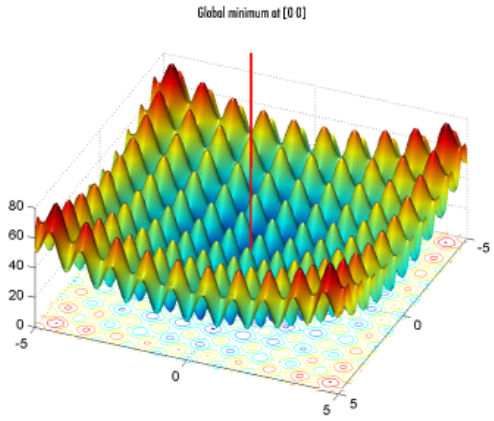


Abbildung 8.1: Stark multimodale Zielfunktion.
Quelle: `https://cssanalytics.files.wordpress.com/2013/08/rastrigin-function.png`

8.3.1 Konstruktionsheuristiken

Konstruktionsheuristiken erzeugen einmalig eine Lösung (z.B. Nearest Neighbor beim TSP, First-Fit / Best-Fit bei Bin Packing). Anschließende lokale Verbesserung ist oft sinnvoll. Die Schwierigkeit besteht mitunter darin überhaupt gültige Lösungen erzeugen zu können!

8.3.2 Lokale Suche (LS)

Kernidee: (1) Erzeuge eine Startlösung s (z.B. durch einfache Konstruktion). (2) Betrachte eine *Nachbarschaft* $N(s)$ aller Lösungen, die durch eine kleine Änderung (Move) aus s entstehen. (3) Wähle eine Verbesserung $s' \in N(s)$ (besserer Zielfunktionswert) und setze $s \leftarrow s'$. Wiederhole bis keine Verbesserung mehr möglich ist: dann steckt man in einem *lokalen Optimum*. Beispiele für Moves: 2-Opt / 3-Opt Kantenvertauschungen beim TSP, Insert/Swap/Relocate beim Routing. Herausforderung: Aus lokalen Optima wieder herausfinden (Diversifikation).

Definition 8.1 (Nachbarschaft)**:** Die Nachbarschaft $\mathcal{N}(x)$ einer Lösung x sind all jene Lösungen $x' \in S$ (Lösungsraum S) die aus x durch eine bestimmte Anzahl an Änderungsschritten hervorgehen. $\Rightarrow$ *ähnliche Lösungen*

- **Lokale Suche** durchsucht iterativ Nachbarschaft $\mathcal{N}(x)$, um bessere Lösungen x' zu finden (Zielfunktion $f(x') < f(x)$).

- Vergleichbar mit Gradienten-Abstiegs-Verfahren

Algorithmus 21: Lokale Suche

Input: Lösung $x \in \mathcal{S}$
repeat
  wähle $x' \in \mathcal{N}(x)$
  if $f(x') \leq f(x)$ **then**
    $x \leftarrow x'$
until ...;
return x

Strategien zur Wahl von x':

- **Random Neighbor:** zufällige Nachbarlösung
- **Next Improvement:** nächst-bessere Nachbarlösung
- **Best Improvement:** beste aller Nachbarlösungen

Lokale Suche findet lokales Minimum bezüglich der Startlösung und der Nachbarschaft $\mathcal{N}$.

Multi-Start Local Search: mehrere unterschiedliche Startlösungen generieren, und Lokale Suche ausführen.

8.3.3 Variable Nachbarschaftssuche (VNS)

- Unterschiedliche Nachbarschaften: $\mathcal{N}_1(x), \mathcal{N}_2(x), \ldots, \mathcal{N}_{k_{\max}}(x)$

Algorithmus 22: Variable Nachbarschaftssuche

Input: Lösung $x \in \mathcal{S}$
$k \leftarrow 1$
repeat
  choose $x' \in \mathcal{N}_k(x)$
  if $f(x') < f(x)$ **then**
    $x \leftarrow x'$
    $k \leftarrow 1$
  else
    $k \leftarrow k + 1$
until $k = k_{\max}$;

8.3.4 Simulated Annealing (SA)

Simulated Annealing ist inspiriert durch Mechanismen bei der Metall-Abkühlung: Solange die Temperatur hoch ist, darf das System auch "schlechte" Schritte machen und entkommt so lokalen Optima. Sinkt die Temperatur, wird das Verfahren konservativer.

Kernschritt: Wähle zufälligen Nachbarn x'. Ist er besser (f(x')¡f(x)), akzeptiere ihn. Ist er schlechter, akzeptiere ihn mit Wahrscheinlichkeit $\exp(-\Delta/T)$, wobei $\Delta = f(x') - f(x) > 0$.

Cooling Schedule: Typisch geometrisch $T \leftarrow \alpha T$ mit $0.8 \leq \alpha < 1$. Zu schnelles Abkühlen kann in einem schlechten lokalen Optimum enden; zu langsames Abkühlen kostet Zeit.

- Grundidee: Auch schlechtere Lösungen werden mit bestimmter Wahrscheinlichkeit akzeptiert.
- Nachbarlösung $x' \in \mathcal{N}(x)$ mit $f(x') > f(x)$ wird auch angenommen wenn

$$r < e^{-|f(x')-f(x)|/T},$$

  mit Zufallszahl $r \in [0, 1)$.
- Schrittweises Absenken der Temperatur T
- Sonst wie *Lokale Suche*

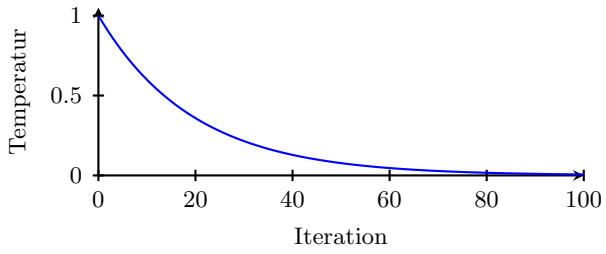


Abbildung 8.2: Typischer geometrischer Temperaturverlauf $T_0 = 1$, $T_k = 0.95^k$.

8.3.5 Tabu-Suche (TS)

Vermeidet das ständige Hin- und Herspringen zwischen wenigen Lösungen durch ein *kurzfristiges Gedächtnis* (Tabu-Liste). Diese merkt sich kürzlich verwendete Moves (oder Attribute) und verbietet deren unmittelbare Rücknahme.

Mechanismen:

- *Tabu-Liste:* Enthält z.B. die zuletzt geswappten Knoten. Länge bestimmt Balance zwischen Diversifikation und Flexibilität.
- *Aspirationskriterium:* Ignoriere Tabu, falls der Move eine neue globale Bestlösung liefert.
- *Langzeit-Memory:* Statistiken (Frequenzen), um selten besuchte Bereiche gezielt anzusteuern.

Iteration: Wähle das beste *zulässige* (nicht tabuierte) Nachbarlösungskandidaten und aktualisiere Listen.

8.3.6 Genetische Algorithmen (GA)

Genetische Algorithmen (GA) sind der prominenteste Vertreter aus der Klasse der *evolutionären Algorithmen*, einer Klasse von Optimierungsverfahren die sich stark an Mechanismen der natürlichen Evolution orientiert.

Sie arbeiten mit einer *Population* mehrerer Lösungen statt nur einer. Die Population besteht aus Individuen (Kandidaten-Lösungen), die im Lauf der *Generationen* (Iterationen) verbessert werden. Ziel ist, eine gewisse Diversität im "Gen-Pool" der Population zu erhalten, um nicht vorschnell zu konvergieren (d.h. alle Individuen der Population sind sehr ähnlich).

Die wesentlichen Elemente eines GAs sind:

Chromosom / Kodierung: Kodierung einer Lösung, oft als "Bit-String". Beim TSP: Permutation aller Städte.

Fitness: Bewertet Qualität (bei Minimierung oft: Fitness = 1 / Kosten oder eine Rang-basiert normalisierte Größe).

Selektion: Bevorzugt gute Individuen als Eltern. Häufig: Fitness-proportionale Selektion (Roulette-Wheel-Selection) (Zufall gewichtet nach Fitness). Selektionswahrscheinlichkeit für Individuum i bei Populationsgröße n:

$$p_S(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)},$$

wobei $\forall j : f(j) > 0$.
Alternativ: *Tournament-Selection*: die Kandidatenlösungen werden direkt verglichen; sie treten ähnlich wie bei einem Turnier gegeneinander an, und der beste wird

ausgewählt..

Crossover (Rekombination): Kombiniert zwei Eltern, um Merkmale zu mischen (Order Crossover für Permutationen, Ein-/Zweipunkt für Binärkodierung).

Mutation: Kleine zufällige Änderung (Bit flip, Swap, Inversion eines Teilsegments) erhält Diversität und verhindert vorzeitiges Zusammenfallen.

Replacement / Elitismus: Auswahl der nächsten Generation; Elitismus behält die beste bisher gefundene Lösung sicher bei.

Algorithmus 23: Generischer GA

```
Erzeuge initiale Population P
evaluate(P)
t ← 0
while t < max. generations do
    P' ← recombine(P(t))
    P'' ← mutate(P')
    evaluate(P'')
    P(t + 1) ← select(P'' ∪ P(t))
    t ← t + 1
Ausgabe der insgesamt besten Lösung
```

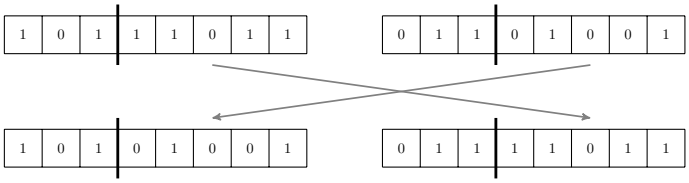


Abbildung 8.3: One-Point Crossover zweier Eltern (oben) zu zwei Kindern (unten).

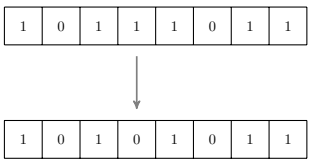


Abbildung 8.4: Einfache Mutation durch Flip eines Bits

- Roulette Wheel Selection = Fitness-Proportionale Selektion

- Selektionswahrscheinlichkeit für Individuum i bei Populationsgröße n:

$$p_S(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)},$$

wobei $\forall j : f(j) > 0$.

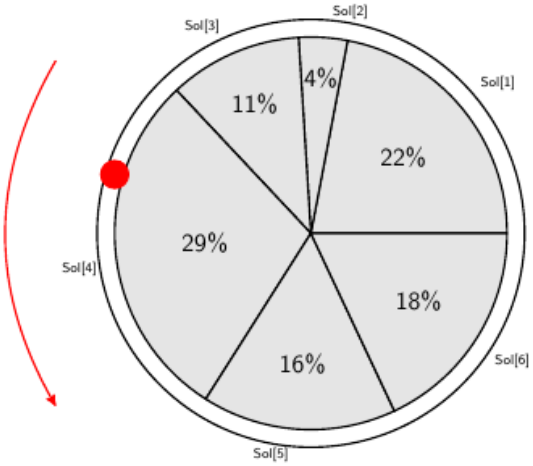


Abbildung 8.5: Roulette Wheel Selection: Die Wahrscheinlichkeit, dass eine Lösung ausgewählt wird, ist proportional zu ihrer Fitness.

8.3.7 Ant Colony Optimization

Das Verfahren *Ant Colony Optimization* (ACO), ist inspiriert durch das Verhalten von Ameisenkolonien bei der Futtersuche. Viele künstliche Ameisen bauen gleichzeitig Lösungen auf. Gute Entscheidungen hinterlassen eine stärkere "Pheromonspur", wodurch spätere Ameisen bevorzugt ebenfalls gute Komponenten wählen.

Wahrscheinlichkeitsregel: Beim Erweiterungsschritt wählt Ameise eine Kante (i, j) mit Wahrscheinlichkeit proportional zu $\tau_{ij}^{\alpha}\eta_{ij}^{\beta}$ (τ = Pheromon, η = lokale Heuristik wie 1/Distanz).

Für alle Kanten $[i, j]$ im Konstruktionsgraph:

- **Lokale Information:** η_{ij} (z.B. Greedy)
- **Pheromon-Werte:** τ_{ij}

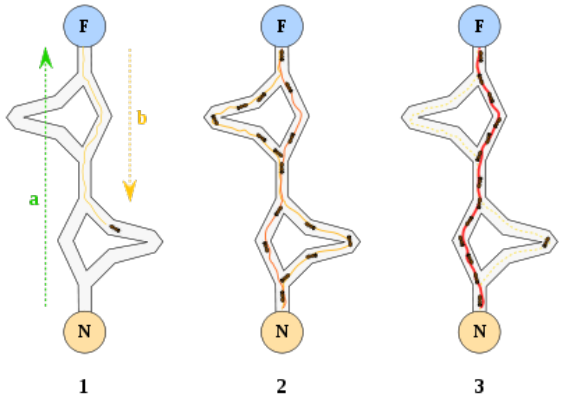


Abbildung 8.6: Futtersuche von Ameisen im Experiment
Quelle: `Quelle: https://biology.stackexchange.com/questions/56654/how-do-ants-follow-ea`

Abbildung 8.7: Auswahl der nächsten Kante.

Aktualisierung:

1. Verdunstung: $\tau_{ij} \leftarrow (1-\rho)\tau_{ij}$ für alle Kanten (verhindert Dominanz veralteter Informationen).
2. Verstärkung: Erhöhe τ_{ij} auf Kanten guter Touren (z.B. bester der aktuellen Iteration oder global bester).

So entsteht ein Feedback-Loop aus Exploration und Ausnutzung.

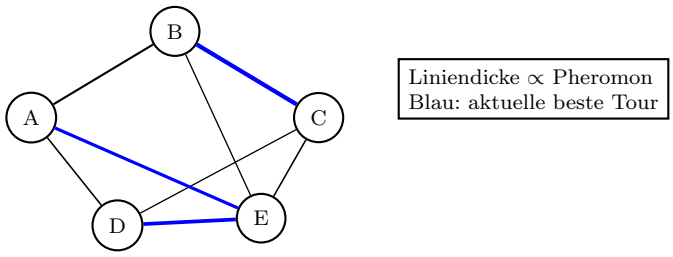


Abbildung 8.8: ACO: Pheromonintensitäten auf Kanten (dickere Linien = höhere τ).

8.3.8 Hybride Metaheuristiken

Verknüpfen Stärken: Ein exakter Löser verarbeitet kleine Teilprobleme oder liefert Bounds; eine Metaheuristik sorgt für globale Suche (z.B. Large Neighborhood Search: entferne Teil der Lösung (Ruin) und konstruiere ihn verbessert wieder (Recreate)).

8.3.9 Bewertung von Heuristiken.

Gute empirische Studien verwenden:

- Feste Zufalls-*Seeds* für Reproduzierbarkeit.
- Mehrere Instanzklassen (klein/mittel/groß; strukturiert vs. zufällig).
- Auswertung: Mittelwert, Bestes, Schlechtestes, Standardabweichung.
- Qualitäts-Zeit-Kurven (wie schnell nähert man sich guter Qualität?).
- Vergleich zu einfachen Baselines (Greedy, lokale Suche) – sonst ist Fortschritt schwer messbar.

Es gibt kein universell bestes Verfahren ("No Free Lunch").

8.4 Branch-and-Bound

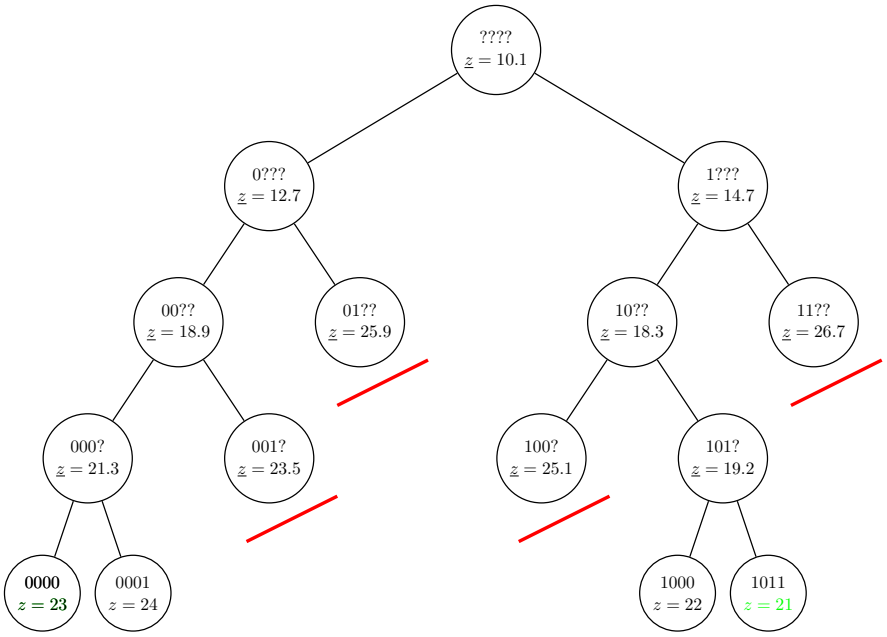


Abbildung 8.9: Beispiel: Branch-and-Bound Baum

8.5 Ganzzahlige lineare Programmierung

Viele Optimierungsprobleme können durch lineare (Un-)Gleichungen formuliert werden. Ein solches Lineares Programm (LP) in kanonischer Form lautet:

$$\begin{aligned} \min \quad & \vec{x}^T \cdot \vec{c} \\ \text{s.t.} \quad & A\vec{x} \geq \vec{b} \\ & \vec{x} \geq \vec{0} \end{aligned}$$

mit $\vec{x} \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ und $\vec{b} \in \mathbb{R}^m$.

Minimierungsprobleme können einfach in Maximierungsprobleme umgewandelt werden (und umgekehrt). Gleichungen können mit sog. Schlupfvariablen als Ungleichungen dargestellt werden.

Lösungsprinzipien.

- **Simplex**: Traversiert Ecken (Basislösungen) des zulässigen Polyeders; oft sehr schnell, Worst-Case exponentiell.
- **Innere-Punkte-Verfahren**: Polynomial, bewegen sich durch das Innere (Barrier Methods). Gut für große, dichte LPs.
- **Branch-and-Bound**: Baumexploration; Relaxations-Bounds schneiden Teilbäume ab.
- **Cutting Planes**: Hinzufügen gültiger Ungleichungen (Schnitte) zur Verschärfung der Relaxation (Gomory, Cover Cuts, Lift-and-Project, ...).
- **Branch-and-Cut**: Kombination von Branch-and-Bound und dynamisch generierten Cuts (moderne MILP-Solver-Standard).

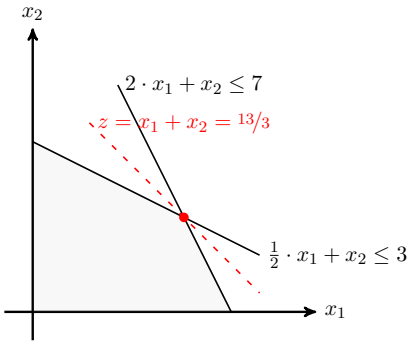


Abbildung 8.10: Gültiger Bereich (Polygon) und Zielfunktion im $\mathbb{R}^2$

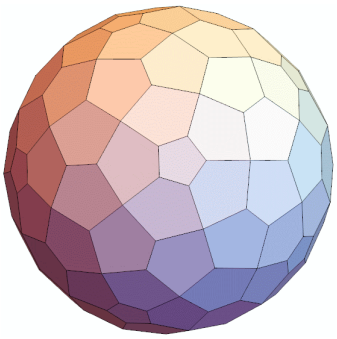


Abbildung 8.11: Beispiel: $\mathbb{R}^3$ Polyeder

Integer Linear Program (ILP): (Ganzzahliges Lineares Programm)

$$\min \quad \vec{x}^T \cdot \vec{c} \tag{8.1}$$

$$\text{s.t.} \quad A\vec{x} \geq \vec{b} \tag{8.2}$$

$$\vec{x} \geq \vec{0} \tag{8.3}$$

mit $\vec{x} \in \mathbb{Z}^n$, $A \in \mathbb{Z}^{m \times n}$ und $\vec{b} \in \mathbb{Z}^m$.

Im Gegensatz zu LP gilt für ILP:

Theorem 8.1 (Komplexität von ILP)**:** *Ganzzahlige Lineare Programmierung ist NP-schwierig.*

Mixed Integer Programming (MIP): Mischform zwischen LP und ILP

- Lösung des LPs enthält fraktionale Werte
- ILP-Lösung kann i.A. nicht aus LP-Lösung abgeleitet werden.

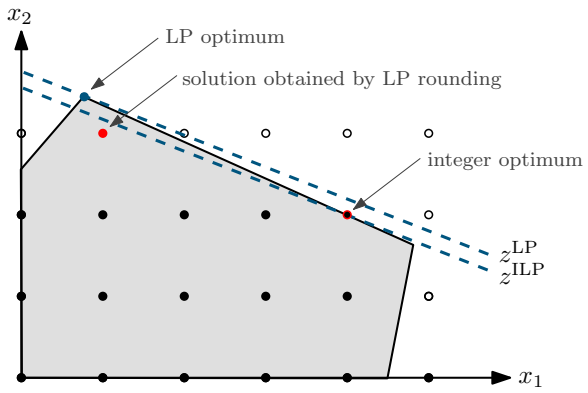


Abbildung 8.12: Rundung von LP-Lösungen auf ganzzahlige Werte

- Eine ILP/MIP-Formulierung kann nicht direkt mit dem Simplex-Algorithmus gelöst werden.
- Dennoch ist Lösung der *LP-Relaxierung* hilfreich.
- LP-Relaxierung: Lösen des ILPs ohne Ganzzahligkeitsbedingungen (also des zugrundeliegenden LPs)
- Betrachten Minimierungsproblem...

- LP-Relaxierung liefert **untere Schranke** z^{LP} des Zielfunktionswertes der optimalen Lösung
- Sei z^{inc} der bisher beste gefundene Zielfunktionswert einer gültigen Lösung (incumbent solution) $\Rightarrow$ **obere Schranke**
- LP-basiertes B&B:
  - Berechnung der LP-Relaxierung in jedem Knoten des B&B-Baumes.
  - Verwendung der Schanken zum Beschneiden der Teilbäume.
  - Verzweigung (branching) mit $x_i^l \leq \lfloor x_i \rfloor$ und $x_i^r \geq \lceil x_i \rceil$

- Eine Zeile im Gleichungssystem definiert eine Hyperebene im Polyeder.
- Idee: Suche Ungleichungen die in aktueller Lösung verletzt sind
- Hinzufügen einer Ungleichung (während B&B):
  - Separierung einer Schnittebene

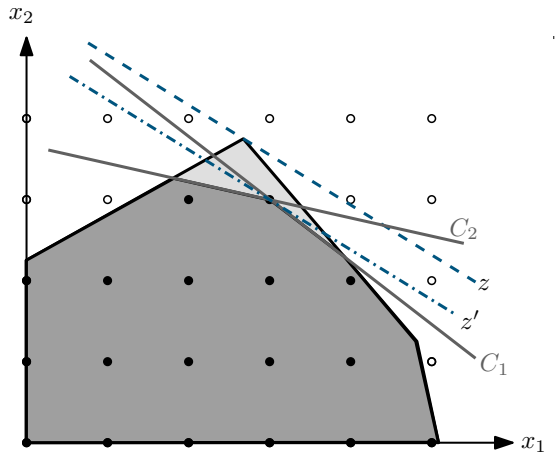


Abbildung 8.13: Die Schnittebenen (Cutting-Planes) C_1 und C_2 verkleinern das Polyeder. Der ursprüngliche Zielfunktionswert z ist dadurch nicht mehr gültig, z' entspricht in diesem Fall einer ganzzahligen Lösung.

- Kombination von Schnittebenenverfahren mit LP-basiertem B&B: **Branch-and-Cut**
- Standard-Schnittebenen: Verkleinern Polyeder, sodass es näher an ganzzahligen Werten liegt.

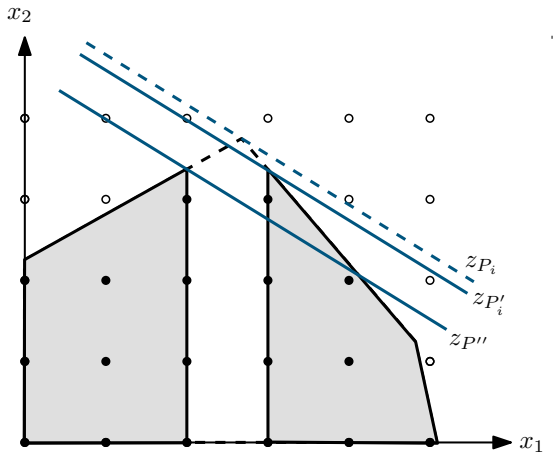


Abbildung 8.14: Kombination von Schnittebenenverfahren mit Branch-and-Bound

- Viele Problem lassen sich mit $O(n^k)$ vielen Ungleichungen formulieren
  - Jedoch oft sehr schlechte Beschreibung des Polyeders
  - Großer B&B-Baum
- Oft Formulierungen mit $O(k^n)$ vielen Ungleichungen *wesentlich* besser.
  - Können jedoch nicht (praktikabel) hinzugefügt werden.
  - $\Rightarrow$ Während B&B **nur bei tatsächlicher Verletzung hinzufügen!**

Beispiel TSP:

$$
\begin{aligned}
z := \min \quad & \sum_{e \in E} x_e \cdot c_e \\
\text{s.t.} \quad & \sum_{i \in V(k)} x_{ik} = 2 \qquad \text{für alle } k \in V
\end{aligned}
$$

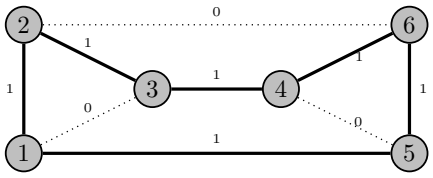


Abbildung 8.15: Beispiel: Lösung mit einer geschlossenen Tour.

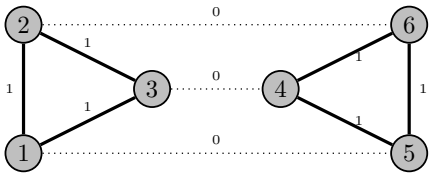


Abbildung 8.16: Beispiel: Lösung mit zwei Subtouren.

- Subtour-Elimination:

$$\sum_{e \in \delta(S)} x_e \leq \|S\| - 1 \text{ für alle } S \subset V, \|S\| \geq 3$$

- Exponentiell viele Ungleichungen, werden nur bei Verletzung gefunden und hinzugefügt
- Auffinden der Cuts:
  - Algorithmus zum Finden eines minimalen Schnittes im Graphen (Max-Flow/Min-Cut)
  - Cut-Separierung oft anhand graphentheoretischer Algorithmen

Damit ist das Ganzzahliges Polyeder aber noch nicht perfekt beschrieben. Zur Lösung des ILP müssen weitere Schnittebenen separiert werden (Branch-and-Cut).

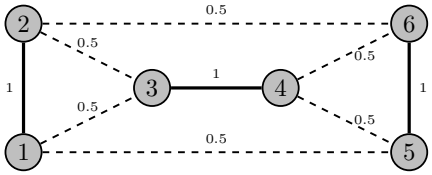


Abbildung 8.17: Beispiel: Fraktionale Lösung nach Cut-Separierung.

Modellierungsmuster.

- **Binäre Variablen** für Auswahl / Aktivierung (Facility open, Kante im Weg, Gegenstand gewählt).
- **Flüsse** zur Pfadmodellierung (Kürzeste Wege, Netzwerkdesign).
- **Big-M** zur Verknüpfung (Implikationen); Wahl eines engen M kritisch (numerische Stabilität).
- **Symmetrie vermeiden**: sonst viele äquivalente Lösungen (z.B. durch Ordnungseinschränkungen).

Bound-Gap und Stop-Kriterien. Solver berichten typischerweise bestes gefundenes Integer-Ziel z und Bound B (LP-Relaxation). Relative Lücke $(z - B)/(|z| + \varepsilon)$ steuert Abbruch.

Einsatz. ILP lohnt bei moderater Problemgröße oder starker Struktur (Schnitte effektiv). Für sehr große Instanzen: Dekomposition (Benders, Dantzig-Wolfe) oder heuristische Verfahren.

LP vs. ILP (Geometrische Sicht). Das LP betrachtet das kontinuierliche Polyeder $P = \{x \in \mathbb{R}^n : Ax \leq b\}$. Das ILP verlangt zusätzlich Ganzzahligkeit. Liegen alle Ecken (extremen Punkte) von P ohnehin in $\mathbb{Z}^n$, nennt man P *integral* – dann sind LP- und ILP-Optimum gleich. Andernfalls ist die beste LP-Lösung nur eine untere Schranke (bei Minimierung). Die Differenz misst man über das *Integrality Gap*. Beispiel Vertex Cover (siehe unten): LP-Wert 1.5, echter Wert 2, Gap = 4/3.

Polyhedrale Formulierungen. Idealer Traum: Ein System von Ungleichungen, dessen Lösungsmenge genau die konvexe Hülle aller ganzzahligen Lösungen ist (dann Gap = 0). In der Praxis nähert man sich diesem Ziel: Man fügt stärkere Ungleichungen (*Facetten*) hinzu oder erweitert das Modell um zusätzliche Variablen (*Extended Formulation*), sodass die Projektion wieder die ursprünglichen Lösungen beschreibt. Jede Verbesserung, die den LP-Bound anhebt (Minimierung) oder senkt (Maximierung), reduziert potentiell die Größe des Branch-and-Bound-Baums.

Beispiel (Vertex Cover: Integrality Gap und Rounding). Für das Dreieck K_3 mit Variablen x_1, x_2, x_3 und Nebenbedingungen $x_1 + x_2 \geq 1$, $x_2 + x_3 \geq 1$, $x_1 + x_3 \geq 1$ ist eine ganzzahlige optimale Lösung z.B. $(1, 1, 0)$ mit Wert 2. Die LP-Relaxation erlaubt $(1/2, 1/2, 1/2)$ mit Wert 1.5. Verhältnis $2/1.5 = 4/3$. Würde man einfach jede Variable mit $x_i \geq 1/2$ auf 1 setzen, bekäme man alle drei Knoten (Wert 3) – sogar schlechter als optimal. Ein besseres (klassisches) 2-Approximation-Verfahren: Wiederholt eine beliebige unbedeckte Kante wählen und beide Endpunkte ins Cover legen. Am Ende ist jede Kante abgedeckt und es lässt sich zeigen, dass man höchstens doppelt so viele Knoten wie im Optimum nimmt.

Branch-and-Bound (BB). BB organisiert die Suche nach optimalen ganzzahligen Lösungen als Baum.

Ablauf in Schritten:

1. Löse LP-Relaxation am Wurzelknoten (Root). Gibt Wert z_{LP} – eine untere Schranke (Minimierung).
2. Ist die Lösung ganzzahlig? Dann speichere sie als bisher Beste.
3. Sonst wähle eine fraktionale Variable x_j (Strategien: am stärksten fraktional, Strong Branching Tests, etc.).
4. Erzeuge zwei Kinder mit zusätzlichen Restriktionen: $x_j \leq \lfloor v \rfloor$ und $x_j \geq \lceil v \rceil$ (bei Binärvariablen typischerweise $x_j = 0$ oder $x_j = 1$).
5. Wiederhole: Wähle nächsten zu explorierenden Knoten (Depth-First spart Speicher; Best-Bound priorisiert vielversprechende Schranken).
6. *Pruning:* Verwerfe einen Knoten, wenn sein LP-Bound schon schlechter ist als die beste bekannte ganzzahlige Lösung oder das LP unzulässig ist.

Verbesserungen: *Cutting Planes* verschärfen Relaxationen; *Heuristiken* liefern frühe gültige Lösungen (bessere Schranken).

Rounding vs. Exakte Lösung. Runden einer LP-Lösung ist oft ein schneller Weg zu einer zulässigen (wenn auch nicht optimalen) Lösung – ideal als Startpunkt. Risiken: (1) Nebenbedingungen können verletzt werden (z.B. Kapazitätsüberschreitung) und müssen dann repariert werden. (2) Qualitätsverlust bei großem Integrality Gap. Fortgeschrittene Rounding-Techniken (randomisiert, abhängig, Pipage) steuern diesen Verlust. Für ein *Optimalitätszertifikat* bleibt jedoch Branch-and-Cut notwendig.

Polyedrische Stärkung in der Praxis. Typischer Lösungsablauf moderner MILP-Solver:

1. Löse Root-LP.
2. Generiere Schnitte (Cover, Clique, Gomory, Lift-and-Project) solange sie noch nennenswert den Bound verbessern.
3. Branching auf fraktionale Variablen.
4. In Teilknoten erneut (lokale) Schnitte.
5. Heuristiken (z.B. RINS, Local Branching) liefern neue ganzzahlige Kandidaten.
6. Pruning anhand Bounds.

Ziel: Möglichst früh starke Schranken und gute Kandidaten, damit der Suchbaum klein bleibt.

Zusammenfassung (ILP-Kernpunkte).

- LP-Relaxationen liefern schnelle Schranken; Lücke = Integrality Gap.
- Gute Modellierung (starke Ungleichungen, reduzierte Symmetrie) spart Branch-and-Bound-Knoten.
- Rounding liefert schnelle Näherungen / Startlösungen, garantiert aber nicht optimale Qualität.
- Branch-and-Cut vereint systematische Schrankenverbesserung (Cuts) und Baumexploration für exakte Zertifikate.

A Übungsbeispiele zur Graphentheorie

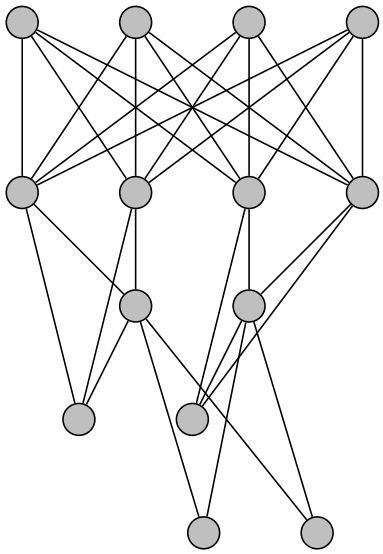
Übung A.1: Konstruktion von Graphen

Konstruieren Sie einen schlichten, zusammenhängenden Graphen mit 14 Knoten, der jeweils

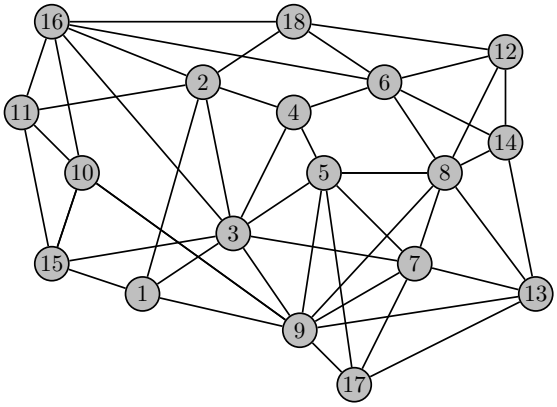
- 4 Knoten vom Grad 6,
- 2 Knoten vom Grad 5,
- 4 Knoten vom Grad 4,
- 2 Knoten vom Grad 3, und
- 2 Knoten vom Grad 2 hat.

Knoten gleichen Grades dürfen nicht adjazent sein, Knoten vom Grad 2 sollen nicht adjazent zu Knoten vom Grad 3 und Grad 4 sein.

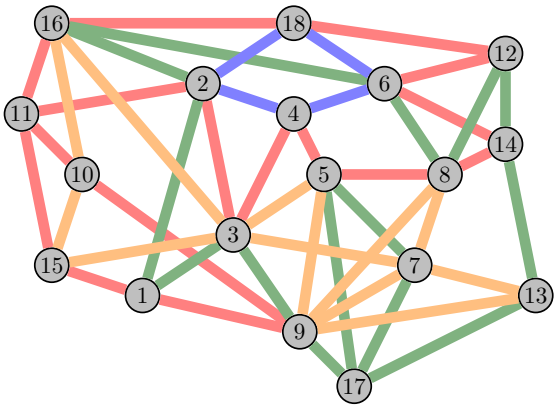
Lösung:



Übung A.2: Euler-Zyklus Geben Sie einen Euler-Zyklus für den folgenden Graphen. Wenn der Graph nicht Eulersch ist, dann ändern Sie eine Kante (hinzufügen, entfernen), sodaß der Graph dann Eulersch ist.



Lösung: Der Graph ist nicht Eulersch, da die Knoten 8 und 13 ungeraden Grad haben. Entfernt man die Kante zwischen diesen beiden Knoten, so existiert ein Euler-Zyklus.



$Z_1 = (1, 15, 11, 16, 18, 12, 6, 14, 8, 5, 4, 3, 2, 11, 10, 9, 1)$

$Z_2 = (1, 2, 16, 6, 8, 12, 14, 13, 17, 7, 5, 17, 9, 3, 1)$

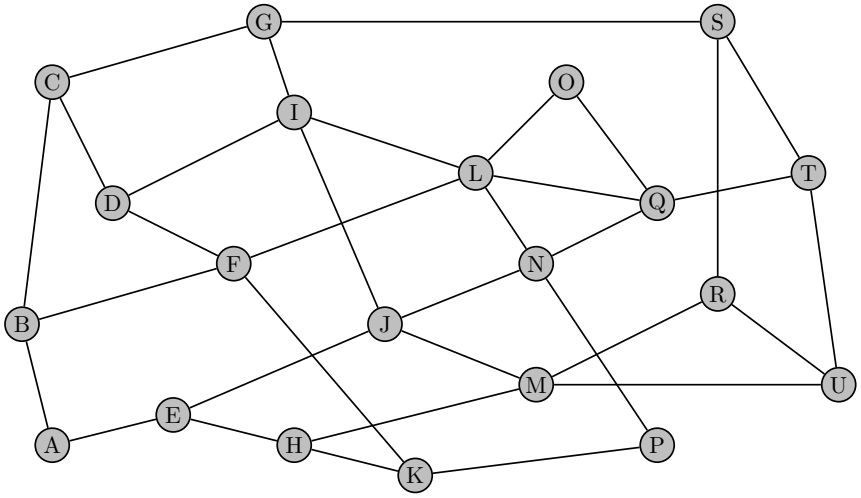
$Z_3 = (18, 6, 4, 2, 18)$

$Z_4 = (16, 3, 5, 9, 13, 7, 8, 9, 7, 3, 15, 10, 16)$

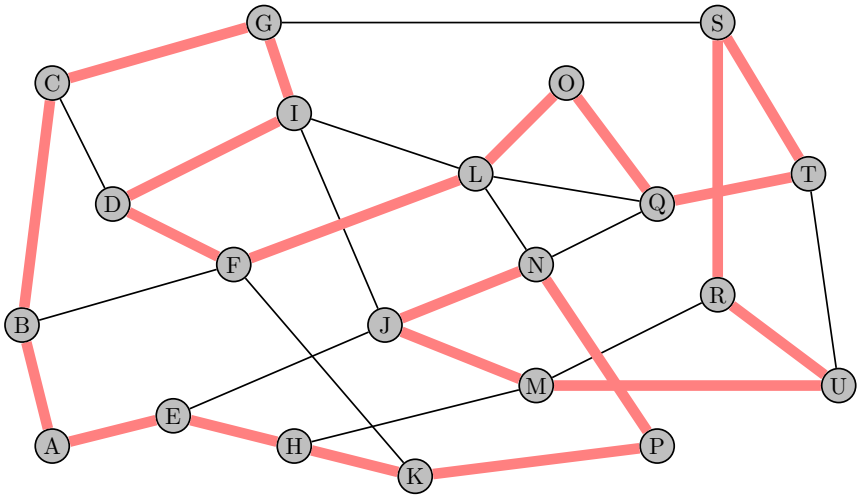
extbfEuler-Zyklus: $Z = (1, 15, 11, 16, 18, 6, 4, 2, 18, 12, 6, 14, 8, 5, 4, 3, 2, 11, 10, 9, 1, 2, 16, 3, 5, 9, 13, 7, 8, 9, 7, 3, 15, 10, 16, 6, 8, 12, 14, 13, 17, 7, 5, 17, 9, 3, 1)$

Übung A.3: Hamilton-Kreis

Geben Sie einen Hamilton-Kreis für den folgenden Graphen an.

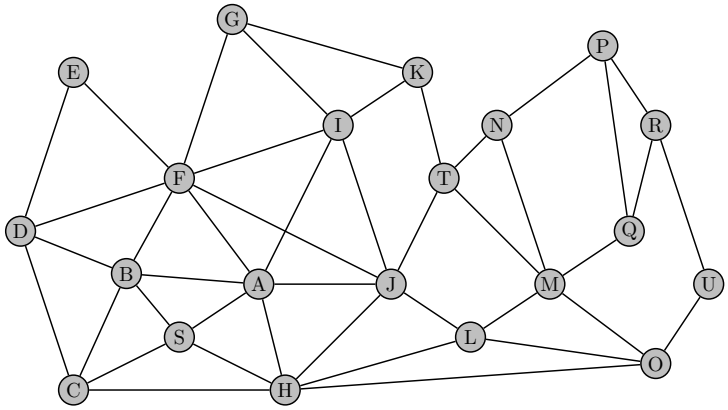


Lösung: Ein möglicher Hamilton-Kreis ist



Übung A.4: Exzentrizitäten, Radius, Durchmesser, Zentrum

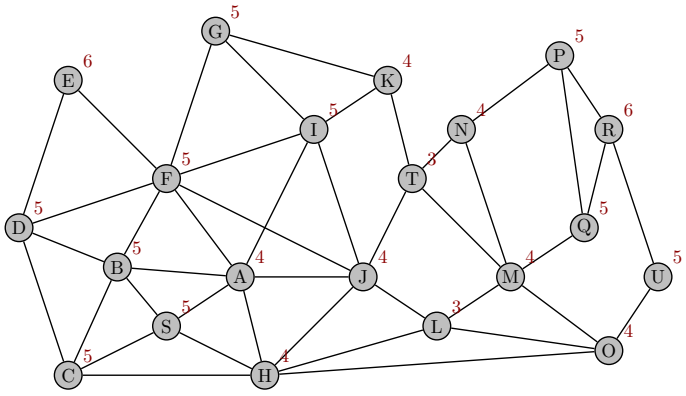
Gegeben sei der Graph G:



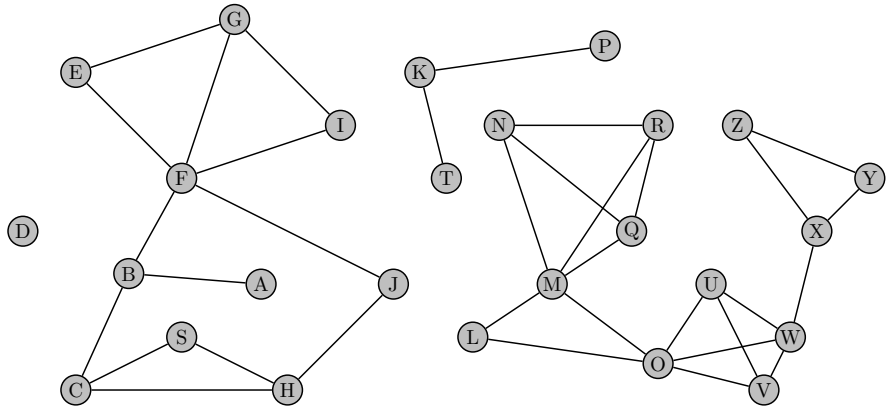
1. Geben Sie die Exzentrizitäten zu allen Knoten an.
2. Bestimmen Sie den Radius $\mathrm{rad}(G)$.
3. Bestimmen Sie den Durchmesser $\mathrm{dm}(G)$.
4. Bestimmen Sie das Zentrum, und schreiben Sie die Knotenmenge $Z(G)$ auf.

Lösung:

1. Lösung im Graph
2. $\mathrm{rad}(G) = 3$.
3. $\mathrm{dm}(G) = 6$.
4. $Z = \{L, T\}$.



Übung A.5: Zusammenhang, Artikulationen, Brücken, Blöcke Gegeben sei der Graph G:



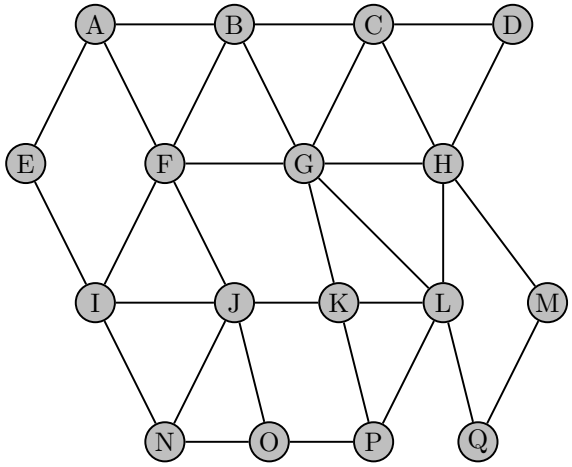
1. Geben Sie alle Zusammenhangskomponenten an.
2. Bestimmen Sie alle Artikulationen.
3. Bestimmen Sie alle Brücken.
4. Bestimmen Sie alle Blöcke, und schreiben Sie diese jeweils als Knotenmenge auf.

Lösung:

1. Zusammenhangskomponenten: $\{A, B, C, E, F, G, H, I, J, S\}$, $\{D\}$, $\{K, P, T\}$, $\{L, M, N, O, Q, R, U, V, W, X, Y, Z\}$
2. Artikulationen: $\{B, F, K, M, O, W, X\}$
3. Brücken: $\{\{A, B\}, \{K, P\}, \{K, T\}, \{W, X\}\}$
4. Blöcke: $\{A, B\}$, $\{B, C, F, H, J, S\}$, $\{F, E, G, I\}$, $\{D\}$, $\{K, P\}$, $\{K, T\}$, $\{L, M, O\}$, $\{M, N, Q, R\}$, $\{O, U, V, W\}$, $\{W, X\}$, $\{X, Y, Z\}$

Übung A.6: Spannbäume

Gegeben sei der folgende Graph G:

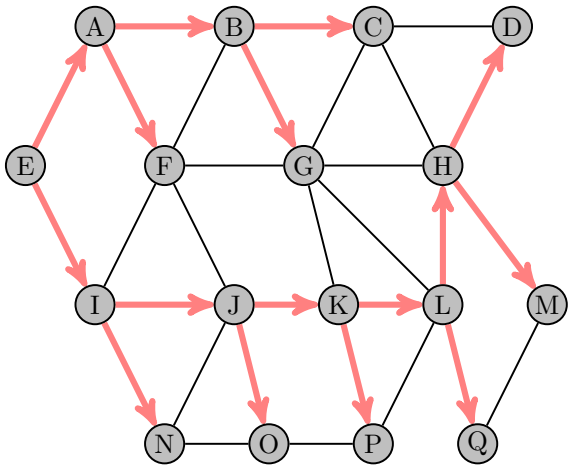


Bestimmen Sie einen Spannbaum mit

1. kleinstem Radius,
2. größtem Durchmesser,
3. der Eigenschaft, dass er ein Binärbaum ist. Unter einem Binärbaum versteht man einen gerichteten Baum, bei dem alle Knoten mit Ausnahme der Wurzel und der Blätter genau einen Vorgänger und zwei Nachfolger haben. Die Blätter haben keine Nachfolger (und sind somit kein Vorgänger eines anderen Knotens). Die Wurzel hat keinen Vorgänger, aber genau zwei Nachfolger.

Lösung:

1. Kleinstes Radius: Ein möglicher Spannbaum mit kleinstem Radius hat die Wurzelknoten B, F, G und K und hat einen Radius von 3.
2. Größter Durchmesser: Ein möglicher Spannbaum mit größtem Durchmesser hat einen Durchmesser von 16.
3. Siehe Graph



Übung A.7: Programmieraufgabe Implementieren Sie ein Programm zur Berechnung wichtiger Eigenschaften von Graphen. Der Graph soll aus einer Datei eingelesen werden, und ist als Adjazenzmatrix[1] gegeben.

Berechnen Sie folgende Eigenschaften des Graphen:

1. Alle Distanzen zwischen den Knoten
2. Zusammenhangskomponenten
3. Exzentrizitäten, Radius, Durchmesser und Zentrum (bei zusammenhängenden Graphen)
4. Artikulationen und Brücken

Vergleichen Sie Ihre Ergebnisse mit jenen auf `https://graphen.theoretische-informatik.at/`. Dort können Sie (ganz unten) auch eine csv-Datei herunterladen, die Daten zu einem Graphen mit 24 Knoten enthält.

Primäre Anforderung ist das Implementieren der Algorithmen, eine grafische, interaktive Darstellung kann bei Interesse ebenso implementiert werden. Achten Sie diesfalls aber auf eine saubere Trennung von algorithmischer Logik und Benutzeroberflächen-Logik.

[1] Eine Adjazenzmatrix ist eine quadratische Matrix, die die Kanten eines Graphen darstellt. Die Elemente der Matrix sind 1, wenn eine Kante zwischen den Knoten existiert, und 0, wenn nicht.

B Übungsbeispiele zu Rekursion

Übung B.1: Programmieraufgabe Gegeben ist ein 2-dimensionales Boolean-Array, befüllt mit *wahr* und *falsch*.

Ermitteln Sie mit einem Programm die Größe des größten zusammenhängenden Gebietes mit Wert wahr.

Beispiel: hier wird falsch mit einem '.' und wahr mit einem 'X' dargestellt.

```
...XXX..XX..
.XXX.XXXX...
X.XX..XX..XX
X...XXX..XXX
XXX....XXX..
XX.XX..X..XX
...X..X.XXX.
.....XXXX...
```

In diesem Beispiel ist die Größe 19.

Füllen Sie ein zweidimensionales Boolean-Array (Größe 300x300) mit Zufallswerten, wobei mit einer Wahrscheinlichkeit von 1/3 der Wert wahr, und mit 2/3 der Wert falsch gesetzt werden soll. Wenden Sie den Algorithmus aus dem vorigen Beispiel an. Ist das Ergebnis plausibel? Was passiert wenn man die Wahrscheinlichkeiten tauscht?

Lösung:

Idee: Tiefensuche (DFS) über alle noch nicht besuchten, auf *wahr* gesetzten Zellen. Für jede solche Startzelle wird die Größe der zusammenhängenden (4-Nachbarschaft: oben, links, unten, rechts) Komponente rekursiv gezählt; das Maximum wird gemerkt. Die Laufzeit ist $\mathcal{O}(n \cdot m)$ für ein Array der Größe $n \times m$, da jede Zelle höchstens einmal besucht wird.

Hinweis: Für ein 300x300-Gitter kann eine einzelne wahre Komponente im Extremfall sehr groß werden. Python hat eine Standard-Recursion-Limit (meist 1000); daher wird dieses (konservativ) angehoben. Alternativ ließe sich die DFS auch iterativ mit einem Stack implementieren.

Python-Implementierung:

```
import random, sys
sys.setrecursionlimit(200000)   # Rekursionstiefe vorsorglich erhöhen

ROWS = 300
COLS = 300
P_TRUE = 1/3   # Wahrscheinlichkeit für 'True' (wahr)

# Gitter initialisieren
grid = [[random.random() < P_TRUE for _ in range(COLS)] for _ in range(ROWS)]
visited = [[False]*COLS for _ in range(ROWS)]

def dfs(r, c):
    """Zählt die Größe der zusammenhängenden True-Komponente ab (r,c)."""
    if r < 0 or r >= ROWS or c < 0 or c >= COLS:  # außerhalb
        return 0
    if visited[r][c] or not grid[r][c]:           # bereits gesehen oder False
        return 0
    visited[r][c] = True
    size = 1
    # 4-Richtungen: oben, links, unten, rechts
    size += dfs(r-1, c)
    size += dfs(r, c-1)
    size += dfs(r+1, c)
    size += dfs(r, c+1)
    return size

max_region = 0
for r in range(ROWS):
    for c in range(COLS):
        if grid[r][c] and not visited[r][c]:
            comp_size = dfs(r, c)
            if comp_size > max_region:
                max_region = comp_size
```

```
print("Größe größtes Gebiet:", max_region)

# Optional: kleine Textdarstellung eines oberen Ausschnitts
for r in range(min(12, ROWS)):
    line = ''.join('X' if grid[r][c] else '.' for c in range(min(60, COLS)))
    print(line)
```

Beobachtungen / Diskussion:

- Bei $P = \frac{1}{3}$ (hier: TRUE mit Wkt. 1/3) zerfällt das Gitter typischerweise in mehrere mittelgroße Komponenten; das Maximum ist (zufällig) deutlich kleiner als $n \cdot m$.
- Erhöht man auf $P = \frac{2}{3}$, so gibt es oft eine dominante ("perkolierende") Komponente, die große Teile oder fast das gesamte Gitter umfasst; der gefundene Maximalwert springt stark nach oben.
- Die Komplexität bleibt linear in der Zellanzahl; der Unterschied liegt nur in der Größe der größten Komponente und damit der maximalen Rekursionstiefe.
- Für robuste Produktion würde man eine iterative Variante (eigener Stack / Queue) nutzen, um nicht vom Recursion-Limit abhängig zu sein.

C Übungsbeispiele zu Sortieralgorithmen

Übung C.1: Sortierabläufe (Beispiel 1) Gegeben sei die Zahlenfolge (12 Elemente):

$$A = (37, 12, 45, 2, 18, 25, 7, 30, 50, 1, 19, 5).$$

Sortieren Sie mit (a) Bubble Sort, (b) Selection Sort, (c) Insertion Sort, (d) Quicksort (Variante wie im Skript: letztes Element als Pivot, Partition wie in Algorithmus PARTITION), (e) Mergesort (Top-Down). Notieren Sie nach jedem Zwischenschritt (äußerer Schleifendurchlauf / Einfügevorgang / Partition / Merge-Ebene) den Zwischenzustand.

(a) Bubble Sort (nach jedem abgeschlossenen Durchlauf – vorzeitiger Abbruch möglich, wenn keine Vertauschung mehr erfolgt)

```
37   12   45   2    18   25   7    30   50   1    19   5

12   37   2    18   25   7    30   45   1    19   5    50
12   2    18   25   7    30   37   1    19   5    45   50
2    12   18   7    25   30   1    19   5    37   45   50
2    12   7    18   25   1    19   5    30   37   45   50
2    7    12   18   1    19   5    25   30   37   45   50
2    7    12   1    18   5    19   25   30   37   45   50
2    7    1    12   5    18   19   25   30   37   45   50
2    1    7    5    12   18   19   25   30   37   45   50
1    2    5    7    12   18   19   25   30   37   45   50
```

(b) Selection Sort (nach jedem Finden/Platzieren des Minimal-Elements im unsortierten Rest)

```
 37   12   45   2    18   25   7    30   50   1    19   5

(37)  12   45   2    18   25   7    30   50  [1]   19   5
 1   (12)  45  [2]   18   25   7    30   50   37   19   5
 1    2   (45)  12   18   25   7    30   50   37   19  [5]
 1    2    5   (12)  18   25  [7]   30   50   37   19   45
 1    2    5    7   (18)  25  [12]  30   50   37   19   45
 1    2    5    7    12  (25)  [18]  30   50   37   19   45
 1    2    5    7    12   18  (25)  30   50   37  [19]  45
 1    2    5    7    12   18   19  (30)  50   37  [25]  45
 1    2    5    7    12   18   19   25  (50)  37  [30]  45
 1    2    5    7    12   18   19   25   30   37  (50) [45]
```

(c) Insertion Sort (nach jedem Einfügen des Elements an Position i in das sortierte Präfix)

```
37   12   45   2    18   25   7    30   50   1    19   5
```

```
 [12]   37|   45    2     18    25    7     30    50    1     19    5
  12    37  [45]|   2     18    25    7     30    50    1     19    5
 [2]    12    37    45|   18    25    7     30    50    1     19    5
  2     12   [18]   37    45|   25    7     30    50    1     19    5
  2     12    18   [25]   37    45|   7     30    50    1     19    5
  2    [7]    12    18    25    37    45|   30    50    1     19    5
  2     7     12    18    25   [30]   37    45|   50    1     19    5
  2     7     12    18    25    30    37    45  [50]|   1     19    5
 [1]    2     7     12    18    25    30    37    45    50|   19    5
  1     2     7     12    18   [19]   25    30    37    45    50|   5
  1     2    [5]    7     12    18    19    25    30    37    45    50|
```

(d) Quicksort Wir zeigen die Zustände

- bei der Wahl des Pivot-Elements (in runden Klammern)
- nach der Partitionierung
- nach endgültiger Platzierung des Pivot-Elementes

Die Partitionierung ist durch die Pfeile gegeben.

```
>  14     10     11     7      9      3      6      4      1      2      8      5   <

>  14     10     11     7      9      3      6      4      1      2      8     (5)  <
  l*2     10     11     7      9      3      6      4      1    r*14     8      5
>  2      10     11     7      9      3      6      4      1     14      8      5   <
   2     l*1     11     7      9      3      6      4    r*10    14      8      5
>  2      1      11     7      9      3      6      4     10     14      8      5   <
   2      1     l*4     7      9      3      6    r*11    10     14      8      5
>  2      1      4      7      9      3      6     11     10     14      8      5   <
   2      1      4     l*3     9    r*7      6     11     10     14      8      5
>  2      1      4      3      9      7      6     11     10     14      8      5   <
   2      1      4      3    l*5      7      6     11     10     14      8    r*9
>  2      1      4      3     [5]     7      6     11     10     14      8      9   <

>  2      1      4     (3)  <  5      7      6     11     10     14      8      9
   2      1     l*3    r*4     5      7      6     11     10     14      8      9
>  2      1     [3]     4   <  5      7      6     11     10     14      8      9

>  2     (1)  <  3      4      5      7      6     11     10     14      8      9
  l*1    r*2     3      4      5      7      6     11     10     14      8      9
> [1]     2   <  3      4      5      7      6     11     10     14      8      9

   1      2      3      4      5   >  7      6     11     10     14      8     (9)  <
   1      2      3      4      5      7      6    l*8     10     14    r*11     9
   1      2      3      4      5   >  7      6      8     10     14     11      9   <
   1      2      3      4      5      7      6      8    l*9     14     11    r*10
   1      2      3      4      5   >  7      6      8     [9]    14     11     10   <

   1      2      3      4      5   >  7      6     (8)  <  9      14     11     10
   1      2      3      4      5   >  7      6     [8]  <  9      14     11     10
```

```
  1     2     3     4     5   >  7     (6)  <   8     9      14     11     10
  1     2     3     4     5     l*6    r*7     8     9      14     11     10
  1     2     3     4     5   > [6]     7   <   8     9      14     11     10

  1     2     3     4     5     6      7      8     9    >  14     11     (10) <
  1     2     3     4     5     6      7      8     9      l*10    11     r*14
  1     2     3     4     5     6      7      8     9    > [10]    11     14  <

  1     2     3     4     5     6      7      8     9      10   >  11     (14) <
  1     2     3     4     5     6      7      8     9      10   >  11     [14] <
```

Endzustand: 1 2 5 7 12 18 19 25 30 37 45 50.

(e) Mergesort (Darstellung nach Merge-Ebenen; Blöcke mit — getrennt)

```
| 37    12    45    2     18    25     7    30    50     1    19     5 |
| 37    12    45    2     18    25  |  7    30    50     1    19     5 |
| 37    12    45 |  2     18    25  |  7    30    50  |  1    19     5 |
| 37  | 12    45 |  2   | 18    25  |  7  | 30    50  |  1  | 19     5 |
| 37  | 12  | 45 |  2   | 18  | 25  |  7  | 30  | 50  |  1  | 19  |  5 |

| 37  | 12    45 |  2   | 18    25  |  7  |  30   50  |  1  |  5    19 |
| 12    37    45 |  2     18    25  |  7     30   50  |  1     5    19 |
|  2    12    18    25    37    45  |  1     5     7    19    30    50 |
|  1     2     5     7    12    18     19    25    30    37    45    50 |
```

(f) Heap-Sort

```
   Initial array:
|  37    12    45     2    18    25     7    30    50     1    19     5  |

After initial heap construction:
|  50    37    45    30    19    25     7    12     2     1    18     5  |

| (50)   37    45    30    19    25     7    12     2     1    18     5  |
|   5    37    45    30    19    25     7    12     2     1    18  |   50
|  45    37    25    30    19     5     7    12     2     1    18  |   50

| (45)   37    25    30    19     5     7    12     2     1    18  |   50
|  18    37    25    30    19     5     7    12     2     1  |   45    50
|  37    30    25    18    19     5     7    12     2     1  |   45    50

| (37)   30    25    18    19     5     7    12     2     1  |   45    50
|   1    30    25    18    19     5     7    12     2  |   37    45    50
|  30    19    25    18     1     5     7    12     2  |   37    45    50

| (30)   19    25    18     1     5     7    12     2  |   37    45    50
|   2    19    25    18     1     5     7    12  |   30    37    45    50
|  25    19     7    18     1     5     2    12  |   30    37    45    50

| (25)   19     7    18     1     5     2    12  |   30    37    45    50
|  12    19     7    18     1     5     2  |   25    30    37    45    50
```

```
|  19   18    7   12    1    5    2  |  25   30   37   45   50

| (19)  18    7   12    1    5    2  |  25   30   37   45   50
|   2   18    7   12    1    5  |  19   25   30   37   45   50
|  18   12    7    2    1    5  |  19   25   30   37   45   50

| (18)  12    7    2    1    5  |  19   25   30   37   45   50
|   5   12    7    2    1  |  18   19   25   30   37   45   50
|  12    5    7    2    1  |  18   19   25   30   37   45   50

| (12)   5    7    2    1  |  18   19   25   30   37   45   50
|   1    5    7    2  |  12   18   19   25   30   37   45   50
|   7    5    1    2  |  12   18   19   25   30   37   45   50

| ( 7)   5    1    2  |  12   18   19   25   30   37   45   50
|   2    5    1  |   7   12   18   19   25   30   37   45   50
|   5    2    1  |   7   12   18   19   25   30   37   45   50

| ( 5)   2    1  |   7   12   18   19   25   30   37   45   50
|   1    2  |   5    7   12   18   19   25   30   37   45   50
|   2    1  |   5    7   12   18   19   25   30   37   45   50

| ( 2)   1  |   5    7   12   18   19   25   30   37   45   50
|   1  |   2    5    7   12   18   19   25   30   37   45   50
|   1  |   2    5    7   12   18   19   25   30   37   45   50
```

Übung C.2: Sortierabläufe (Beispiel 2) Gegeben sei die Zahlenfolge (12 Elemente):

$$B = (9, 3, 14, 7, 2, 11, 6, 4, 10, 1, 8, 5).$$

Führen Sie dieselben Verfahren durch. (Notation analog Beispiel 1.)

(a) Bubble Sort

```
 14   10   11    7    9    3    6    4    1    2    8    5

 10   11    7    9    3    6    4    1    2    8    5   14
 10    7    9    3    6    4    1    2    8    5   11   14
  7    9    3    6    4    1    2    8    5   10   11   14
  7    3    6    4    1    2    8    5    9   10   11   14
  3    6    4    1    2    7    5    8    9   10   11   14
  3    4    1    2    6    5    7    8    9   10   11   14
  3    1    2    4    5    6    7    8    9   10   11   14
  1    2    3    4    5    6    7    8    9   10   11   14
  1    2    3    4    5    6    7    8    9   10   11   14
```

(b) Selection Sort

```
 14   10   11    7    9    3    6    4    1    2    8    5
```

```
(14)   10    11    7     9     3     6     4    [1]    2     8     5
 1    (10)   11    7     9     3     6     4    14    [2]    8     5
 1     2    (11)   7     9    [3]    6     4    14    10     8     5
 1     2     3    (7)    9    11     6    [4]   14    10     8     5
 1     2     3     4    (9)   11     6     7    14    10     8    [5]
 1     2     3     4     5    (11)  [6]    7    14    10     8     9
 1     2     3     4     5     6    (11)  [7]   14    10     8     9
 1     2     3     4     5     6     7    (11)  14    10    [8]    9
 1     2     3     4     5     6     7     8    (14)  10    11    [9]
```

(c) Insertion Sort

```
 14    10    11    7     9     3     6     4     1     2     8     5

[10]   14|   11    7     9     3     6     4     1     2     8     5
 10    [11]  14|   7     9     3     6     4     1     2     8     5
[7]    10    11    14|   9     3     6     4     1     2     8     5
 7     [9]   10    11    14|   3     6     4     1     2     8     5
[3]    7     9     10    11    14|   6     4     1     2     8     5
 3     [6]   7     9     10    11    14|   4     1     2     8     5
 3     [4]   6     7     9     10    11    14|   1     2     8     5
[1]    3     4     6     7     9     10    11    14|   2     8     5
 1     [2]   3     4     6     7     9     10    11    14|   8     5
 1     2     3     4     6     7     [8]   9     10    11    14|   5
 1     2     3     4     [5]   6     7     8     9     10    11    14|
```

(d) Quicksort

```
>  14     10     11     7      9      3      6      4      1      2      8      5   <

>  14     10     11     7      9      3      6      4      1      2      8     (5)  <
  l*2     10     11     7      9      3      6      4      1     r*14    8      5
>  2      10     11     7      9      3      6      4      1      14     8      5   <
   2     l*1     11     7      9      3      6      4     r*10    14     8      5
>  2      1      11     7      9      3      6      4      10     14     8      5   <
   2      1     l*4     7      9      3      6     r*11    10     14     8      5
>  2      1      4      7      9      3      6      11     10     14     8      5   <
   2      1      4     l*3     9     r*7     6      11     10     14     8      5
>  2      1      4      3      9      7      6      11     10     14     8      5   <
   2      1      4      3     l*5     7      6      11     10     14     8     r*9
>  2      1      4      3     [5]     7      6      11     10     14     8      9   <

>  2      1      4     (3)  <  5      7      6      11     10     14     8      9
   2      1     l*3    r*4     5      7      6      11     10     14     8      9
>  2      1     [3]     4   <  5      7      6      11     10     14     8      9

>  2     (1)  <  3      4      5      7      6      11     10     14     8      9
  l*1    r*2     3      4      5      7      6      11     10     14     8      9
> [1]     2   <  3      4      5      7      6      11     10     14     8      9

   1      2      3      4      5    >  7      6      11     10     14     8     (9)  <
```

```
  1     2     3     4     5     7     6     l*8    10    14    r*11   9
  1     2     3     4     5   > 7     6     8     10    14    11    9   <
  1     2     3     4     5     7     6     8     l*9   14    11    r*10
  1     2     3     4     5   > 7     6     8     [9]   14    11    10  <

  1     2     3     4     5   > 7     6     (8) <   9     14    11    10
  1     2     3     4     5   > 7     6     [8] <   9     14    11    10

  1     2     3     4     5   > 7     (6) <   8     9     14    11    10
  1     2     3     4     5     l*6   r*7     8     9     14    11    10
  1     2     3     4     5   > [6]   7   <   8     9     14    11    10

  1     2     3     4     5     6     7     8     9   > 14    11    (10) <
  1     2     3     4     5     6     7     8     9     l*10  11    r*14
  1     2     3     4     5     6     7     8     9   > [10]  11    14  <

  1     2     3     4     5     6     7     8     9     10  > 11    (14) <
  1     2     3     4     5     6     7     8     9     10  > 11    [14] <
```

Endzustand: 1 2 3 4 5 6 7 8 9 10 11 14.

(e) Mergesort

```
| 14   10   11   7    9    3    6    4    1    2    8    5 |
| 14   10   11   7    9    3  | 6    4    1    2    8    5 |
| 14   10   11 | 7    9    3  | 6    4    1  | 2    8    5 |
| 14  | 10   11 | 7  | 9    3  | 6  | 4    1  | 2  | 8    5 |
| 14  | 10  | 11 | 7  | 9  | 3  | 6  | 4  | 1  | 2  | 8  | 5 |

| 14  | 10   11 |  7  | 3    9  | 6  | 1    4  | 2  | 5    8 |
| 10   11   14 |  3   7    9  | 1    4    6  | 2    5    8 |
|  3    7    9   10   11   14  | 1    2    4    5    6    8 |
|  1    2    3    4    5    6   7    8    9   10   11   14 |
```

Heap-Sort

```
  Initial array:
|  14   10   11    7    9    3    6    4    1    2    8    5  |

After initial heap construction:
|  14   10   11    7    9    5    6    4    1    2    8    3  |

| (14)  10   11    7    9    5    6    4    1    2    8    3  |
|   3   10   11    7    9    5    6    4    1    2    8  |  14
|  11   10    6    7    9    5    3    4    1    2    8  |  14

| (11)  10    6    7    9    5    3    4    1    2    8  |  14
|   8   10    6    7    9    5    3    4    1    2  |  11   14
|  10    9    6    7    8    5    3    4    1    2  |  11   14

| (10)   9    6    7    8    5    3    4    1    2  |  11   14
|   2    9    6    7    8    5    3    4    1  |  10   11   14
```

```
|  9   8   6   7   2   5   3   4   1  |  10  11  14

| ( 9)  8   6   7   2   5   3   4   1  |  10  11  14
|  1   8   6   7   2   5   3   4  |   9   10  11  14
|  8   7   6   4   2   5   3   1  |   9   10  11  14

| ( 8)  7   6   4   2   5   3   1  |   9   10  11  14
|  1   7   6   4   2   5   3  |   8   9   10  11  14
|  7   4   6   1   2   5   3  |   8   9   10  11  14

| ( 7)  4   6   1   2   5   3  |   8   9   10  11  14
|  3   4   6   1   2   5  |   7   8   9   10  11  14
|  6   4   5   1   2   3  |   7   8   9   10  11  14

| ( 6)  4   5   1   2   3  |   7   8   9   10  11  14
|  3   4   5   1   2  |   6   7   8   9   10  11  14
|  5   4   3   1   2  |   6   7   8   9   10  11  14

| ( 5)  4   3   1   2  |   6   7   8   9   10  11  14
|  2   4   3   1  |   5   6   7   8   9   10  11  14
|  4   2   3   1  |   5   6   7   8   9   10  11  14

| ( 4)  2   3   1  |   5   6   7   8   9   10  11  14
|  1   2   3  |   4   5   6   7   8   9   10  11  14
|  3   2   1  |   4   5   6   7   8   9   10  11  14

| ( 3)  2   1  |   4   5   6   7   8   9   10  11  14
|  1   2  |   3   4   5   6   7   8   9   10  11  14
|  2   1  |   3   4   5   6   7   8   9   10  11  14

| ( 2)  1  |   3   4   5   6   7   8   9   10  11  14
|  1  |   2   3   4   5   6   7   8   9   10  11  14
|  1  |   2   3   4   5   6   7   8   9   10  11  14
```

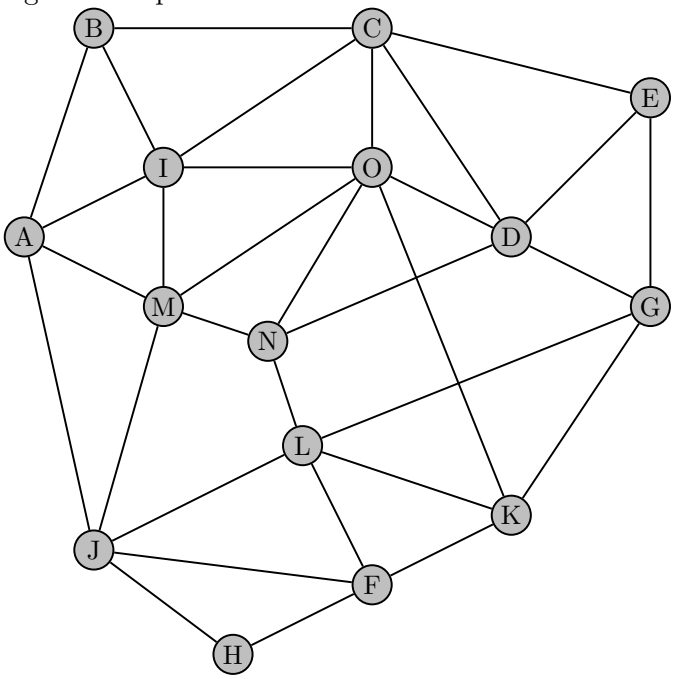
Im Anhang C sind zu zwei Zahlenfolgen die Zwischenschritte zu allen vorgestellten Sortieralgorithmen angegeben.

D Übungsbeispiele zu Datenstrukturen

E Übungsbeispiele zu Graphen-Algorithmen

Übung E.1: Breiten- und Tiefensuche

Gegeben sei der folgende Graph G:



1. Berechnen Sie im Graphen $G = (V, E)$ den *kürzesten Pfad* von A zu G mit *Breitensuche*. Dabei werden alle Kanten in ihrer *lexikographischen*[1] Reihenfolge betrachtet.
2. Führen Sie ausgehend vom Knoten O eine Tiefensuche aus. Die Kanten werden ebenfalls in *lexikographischer* Reihenfolge betrachtet. Schreiben Sie alle Knoten in der Reihenfolge auf in der sie besucht werden.

Lösung:

(1) Breitensuche ab A (lexikographische Betrachtung der Nachbarn). Ebenen (Distanzen):

[1] Ausgehend vom Knoten A wird beispielsweise die Kante $[A, B]$ vor $[A, C]$ betrachtet, da B alphabetisch vor C kommt.

$0:\ \{A\}$
$1:\ \{B, I, J, M\}$ (alle mit Elternknoten A)
$2:\ \{C, O, F, H, L, N\}$ mit Eltern $C \leftarrow B$, $O \leftarrow I$, $F \leftarrow J$,
$H \leftarrow J$, $L \leftarrow J$, $N \leftarrow M$
$3:\ \{D, E, K, G\}$ mit Eltern $D \leftarrow C$, $E \leftarrow C$, $K \leftarrow O$, $G \leftarrow L$.

Kürzester Pfad $A \to G$: $A \to J \to L \to G$ (Länge 3).

BFS-Baum-Kanten: $[A, B], [A, I], [A, J], [A, M], [B, C], [I, O], [J, F]$,
$[J, H], [J, L], [M, N], [C, D], [C, E], [O, K], [L, G]$.

(2) Tiefensuche ab O (lexikographische Nachbarn). Besuchsreihenfolge (Preorder):
$O, C, B, A, I, M, J, F, H, K, G, D, E, N, L$.

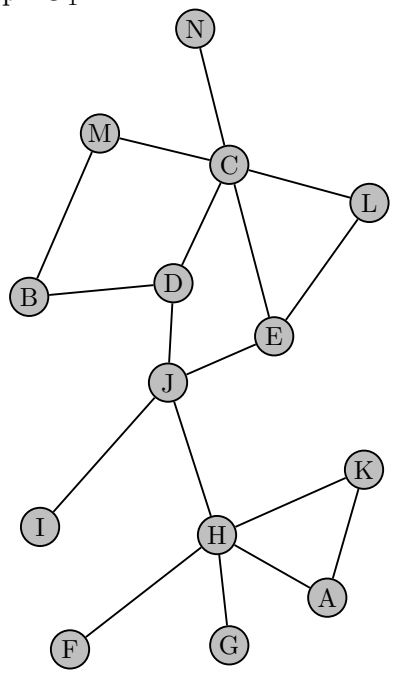
DFS-Baum-Kanten: $[O, C], [C, B], [B, A], [A, I], [I, M], [M, J], [J, F], [F, H]$,
$[F, K], [K, G], [G, D], [D, E], [D, N], [N, L]$.

(Kontrolle: Alle 15 Knoten wurden genau einmal betreten; bei beiden Traversierungen wurde stets der alphabetisch früheste noch nicht besuchte Nachbar gewählt.)

Übung E.2: Breiten- und Tiefensuche

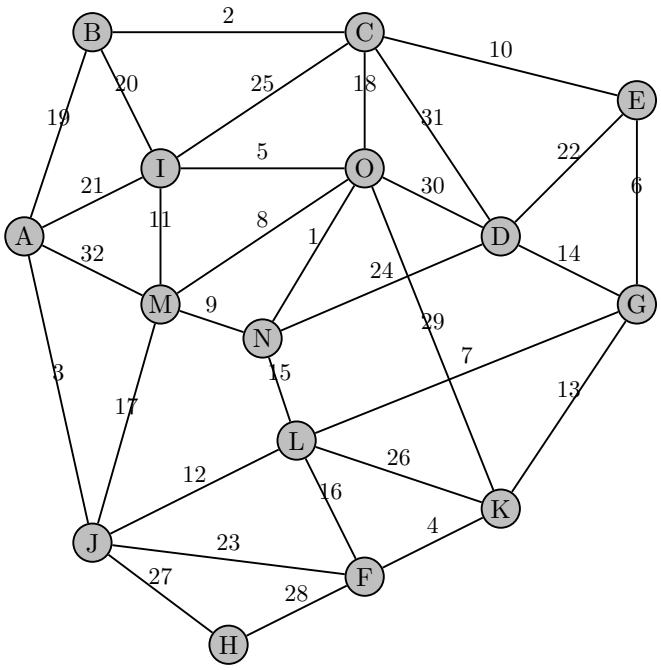
Führen Sie die Algorithmen Breitensuche (BFS) und Tiefensuche (DFS) auf dem Graphen G_1 aus, wobei jeweils mit Knoten D gestartet wird. Geben Sie in der Tabelle rechts die Entdeckungs- und Fertigstellungszeiten $(\tau_d(v), \tau_f(v)$ für $v \in V)$ an.

Graph G_1:



	BFS		DFS	
Schritt	entdeckt	abgeschlossen	entdeckt	abgeschlossen
1	D		D	
2	B		B	
3	C		M	
4	J		C	
5		D	E	
6	M		J	
7		B	H	
8	N		A	
9	L		K	
10	E			K
11		C		A
12	H		F	
13	I			F
14		J	G	
15		M		G
16		N		H
17		L	I	
18		E		I
19	A			J
20	F		L	
21	G			L
22	K			E
23		H	N	
24		I		N
25		A		C
26		F		M
27		G		B
28		K		D

Übung E.3: Minimaler Spannbaum Gegeben sei der folgende Graph G:

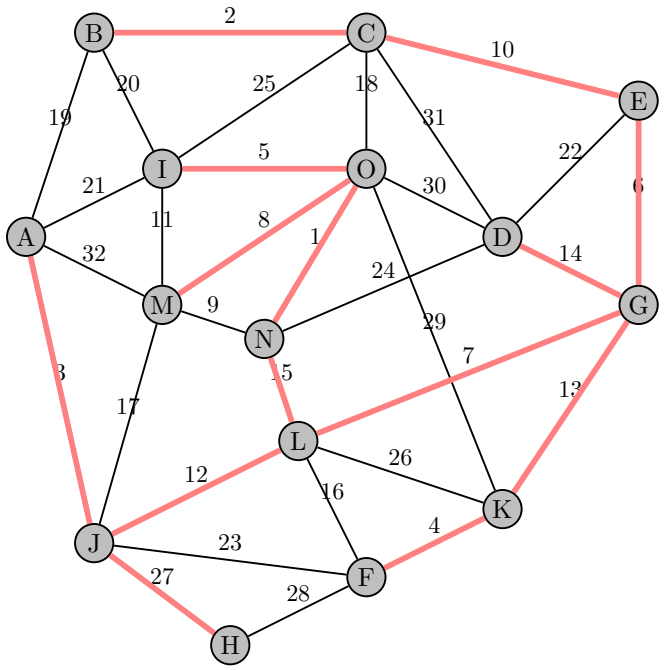


Berechnen Sie den *minimalen Spannbaum* zum Graphen G mit

1. dem Algorithmus von Kruskal, und
2. dem Algorithmus von Prim (Startknoten B).

Geben Sie alle Zwischenschritte an, d.h. welche Kanten in jedem Schritt betrachtet und schließlich verwendet werden. Welches Gesamtgewicht hat der minimale Spannbaum?

Lösung:



Das Gesamtgewicht des minimalen Spannbaumes beträgt 127!

Zwischenschritte des Algorithmus von Prim (Startknoten B):

Notation: $[X, Y]\,(w = \omega)$ bezeichnet eine Kante zwischen X und Y mit Gewicht ω. Lange Mengen werden auf zwei Zeilen umgebrochen.

1. $T = \{B\}$
   $F = \{[B, C]\,(w = 2), [B, A]\,(w = 19), [B, I]\,(w = 20)\}$, wähle $[B, C]\,(w = 2)$.
2. $T = \{B, C\}$
   $F = \{[C, E]\,(w = 10), [B, A]\,(w = 19), [B, I]\,(w = 20), [C, I]\,(w = 25), [C, D]\,(w = 31)\}$, wähle $[C, E]\,(w = 10)$.
3. $T = \{B, C, E\}$
   $F = \{[E, G]\,(w = 6), [B, A]\,(w = 19), [B, I]\,(w = 20), [C, I]\,(w = 25), [C, D]\,(w = 31), [E, D]\,(w = 22)\}$, wähle $[E, G]\,(w = 6)$.
4. $T = \{B, C, E, G\}$
   $F = \{[G, L]\,(w = 7), [G, K]\,(w = 13), [G, D]\,(w = 14), [B, A]\,(w = 19), [B, I]\,(w = 20), [C, I]\,(w = 25), [C, D]\,(w = 31), [E, D]\,(w = 22)\}$, wähle $[G, L]\,(w = 7)$.
5. $T = \{B, C, E, G, L\}$
   $F = \{[L, J]\,(w = 12), [G, K]\,(w = 13), [G, D]\,(w = 14), [L, N]\,(w = 15), [L, F]\,(w = 16), [J, M]\,(w = 17), [B, A]\,(w = 19), [B, I]\,(w = 20), [E, D]\,(w = 22), [J, F]\,(w = 23), [C, I]\,(w = 25), [L, K]\,(w = 26), [J, H]\,(w = 27), [C, D]\,(w = 31)\}$, wähle $[L, J]\,(w = 12)$.
6. $T = \{B, C, E, G, L, J\}$

$F = \{[J,A]\,(w = 3), [G,K]\,(w = 13), [G,D]\,(w = 14), [L,N]\,(w = 15), [L,F]\,(w = 16), [J,M]\,(w = 17), [B,A]\,(w = 19), [B,I]\,(w = 20), [E,D]\,(w = 22), [J,F]\,(w = 23), [C,I]\,(w = 25), [L,K]\,(w = 26), [J,H]\,(w = 27), [C,D]\,(w = 31)\}$, wähle $[J,A]\,(w = 3)$.

7. $T = \{B, C, E, G, L, J, A\}$
   $F = \{[G,K]\,(w = 13), [G,D]\,(w = 14), [L,N]\,(w = 15), [L,F]\,(w = 16), [J,M]\,(w = 17), [B,A]\,(w = 19), [B,I]\,(w = 20), [E,D]\,(w = 22), [J,F]\,(w = 23), [C,I]\,(w = 25), [L,K]\,(w = 26), [J,H]\,(w = 27), [C,D]\,(w = 31)\}$, wähle $[G,K]\,(w = 13)$.
8. $T = \{B, C, E, G, L, J, A, K\}$
   $F = \{[K,F]\,(w = 4), [G,D]\,(w = 14), [L,N]\,(w = 15), [L,F]\,(w = 16), [J,M]\,(w = 17), [B,A]\,(w = 19), [B,I]\,(w = 20), [E,D]\,(w = 22), [J,F]\,(w = 23), [C,I]\,(w = 25), [L,K]\,(w = 26), [J,H]\,(w = 27), [F,H]\,(w = 28), [K,O]\,(w = 29), [C,D]\,(w = 31), [A,M]\,(w = 32)\}$, wähle $[K,F]\,(w = 4)$.
9. $T = \{B, C, E, G, L, J, A, K, F\}$
   $F = \{[G,D]\,(w = 14), [L,N]\,(w = 15), [L,F]\,(w = 16), [J,M]\,(w = 17), [B,A]\,(w = 19), [B,I]\,(w = 20), [E,D]\,(w = 22), [J,F]\,(w = 23), [C,I]\,(w = 25), [L,K]\,(w = 26), [J,H]\,(w = 27), [F,H]\,(w = 28), [K,O]\,(w = 29), [C,D]\,(w = 31), [A,M]\,(w = 32)\}$, wähle $[G,D]\,(w = 14)$.
10. $T = \{B, C, E, G, L, J, A, K, F, D\}$
    $F = \{[L,N]\,(w = 15), [L,F]\,(w = 16), [J,M]\,(w = 17), [B,A]\,(w = 19), [B,I]\,(w = 20), [E,D]\,(w = 22), [J,F]\,(w = 23), [C,I]\,(w = 25), [L,K]\,(w = 26), [J,H]\,(w = 27), [F,H]\,(w = 28), [K,O]\,(w = 29), [C,D]\,(w = 31), [A,M]\,(w = 32)\}$, wähle $[L,N]\,(w = 15)$.
11. $T = \{B, C, E, G, L, J, A, K, F, D, N\}$
    $F = \{[N,O]\,(w = 1), [N,M]\,(w = 9), [J,M]\,(w = 17), [B,A]\,(w = 19), [B,I]\,(w = 20), [E,D]\,(w = 22), [J,F]\,(w = 23), [C,I]\,(w = 25), [L,K]\,(w = 26), [J,H]\,(w = 27), [F,H]\,(w = 28), [K,O]\,(w = 29), [C,D]\,(w = 31), [A,M]\,(w = 32), [N,D]\,(w = 24)\}$, wähle $[N,O]\,(w = 1)$.
12. $T = \{B, C, E, G, L, J, A, K, F, D, N, O\}$
    $F = \{[O,I]\,(w = 5), [O,M]\,(w = 8), [N,M]\,(w = 9), [J,M]\,(w = 17), [B,A]\,(w = 19), [B,I]\,(w = 20), [E,D]\,(w = 22), [J,F]\,(w = 23), [C,I]\,(w = 25), [L,K]\,(w = 26), [J,H]\,(w = 27), [F,H]\,(w = 28), [K,O]\,(w = 29), [O,C]\,(w = 18), [O,D]\,(w = 30), [C,D]\,(w = 31), [A,M]\,(w = 32)\}$, wähle $[O,I]\,(w = 5)$.
13. $T = \{B, C, E, G, L, J, A, K, F, D, N, O, I\}$
    $F = \{[O,M]\,(w = 8), [N,M]\,(w = 9), [I,M]\,(w = 11), [J,M]\,(w = 17), [B,A]\,(w = 19), [B,I]\,(w = 20), [E,D]\,(w = 22), [J,F]\,(w = 23), [C,I]\,(w = 25), [L,K]\,(w = 26), [J,H]\,(w = 27), [F,H]\,(w = 28), [K,O]\,(w = 29), [O,C]\,(w = 18), [O,D]\,(w = 30), [C,D]\,(w = 31), [A,M]\,(w = 32)\}$, wähle $[O,M]\,(w = 8)$.
14. $T = \{B, C, E, G, L, J, A, K, F, D, N, O, I, M\}$
    $F = \{[J,H]\,(w = 27), [F,H]\,(w = 28), [K,O]\,(w = 29), [O,C]\,(w = 18), [O,D]\,(w =$

$30), [B, A]\,(w = 19), [B, I]\,(w = 20), [E, D]\,(w = 22), [J, F]\,(w = 23), [C, I]\,(w = 25), [L, K]\,(w = 26), [C, D]\,(w = 31), [A, M]\,(w = 32)\}$, wähle $[J, H]\,(w = 27)$.

15. $T = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\}$ vollständig.

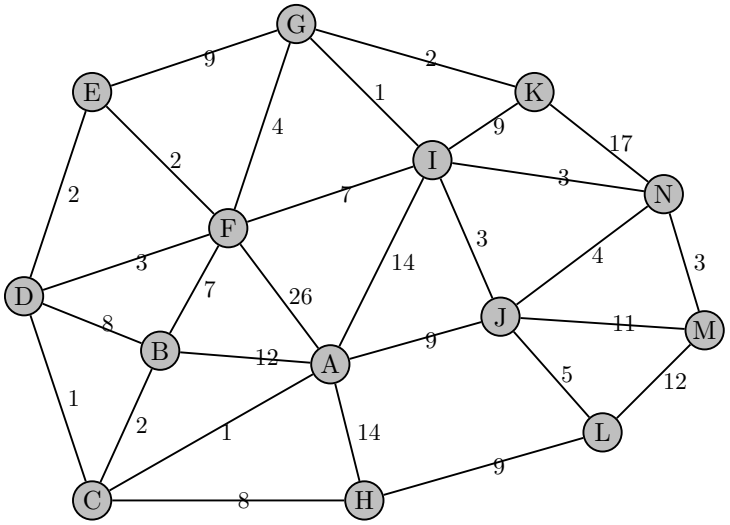
Gewählte Reihenfolge der Kanten: $[B, C]\,(w = 2), [C, E]\,(w = 10), [E, G]\,(w = 6), [G, L]\,(w = 7), [L, J]\,(w = 12), [J, A]\,(w = 3),$

$[G, K]\,(w = 13), [K, F]\,(w = 4), [G, D]\,(w = 14), [L, N]\,(w = 15), [N, O]\,(w = 1), [O, I]\,(w = 5), [O, M]\,(w = 8), [J, H]\,(w = 27)$. Summe = 127.

Übung E.4: Algorithmus von Dijkstra

Berechnen Sie mit dem Algorithmus von Dijkstra für G_1 den kürzesten Weg vom Knoten B zum Knoten N. Der Algorithmus kann beendet werden, sobald der Zielknoten fertiggestellt wird.

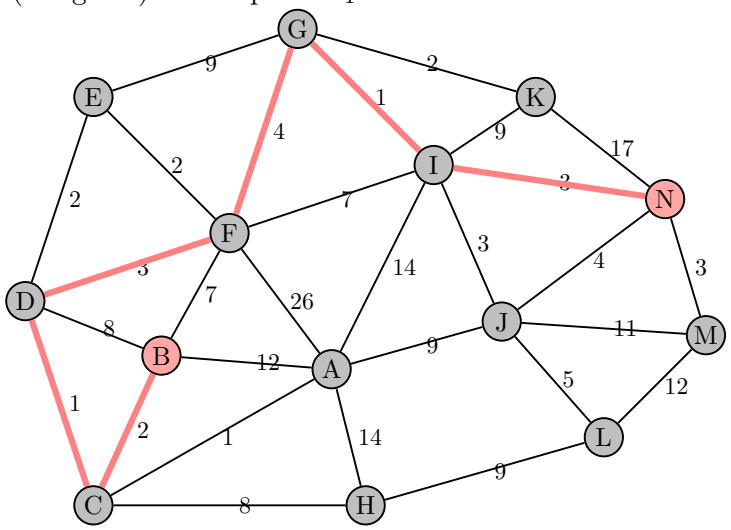
Graph G_1:



Geben Sie in der Tabelle an, welcher Knoten in jeder Iteration fertiggestellt wird, und welche Werte (δ-Werte) aktualisiert werden. Geben Sie zusätzlich die Länge des kürzesten Weges an, und zeichnen Sie diesen in den Graphen ein.

Abgeschlossen	Aktuelle δ-Werte ("∞- unbekannt)													
	A	B	C	D	E	F	G	H	I	J	K	L	M	N
B (0)	12	0	2	8	∞	7	∞	∞	∞	∞	∞	∞	∞	∞
C (2)	3	0	2	3	∞	7	∞	10	∞	∞	∞	∞	∞	∞
A (3)	3	0	2	3	∞	7	∞	10	17	12	∞	∞	∞	∞
D (3)	3	0	2	3	5	6	∞	10	17	12	∞	∞	∞	∞
E (5)	3	0	2	3	5	6	14	10	17	12	∞	∞	∞	∞
F (6)	3	0	2	3	5	6	10	10	13	12	∞	∞	∞	∞
G (10)	3	0	2	3	5	6	10	10	11	12	12	∞	∞	∞
H (10)	3	0	2	3	5	6	10	10	11	12	12	19	∞	∞
I (11)	3	0	2	3	5	6	10	10	11	12	12	19	∞	14
J (12)	3	0	2	3	5	6	10	10	11	12	12	17	23	14
K (12)	3	0	2	3	5	6	10	10	11	12	12	17	23	14
N (14) (Stop)	3	0	2	3	5	6	10	10	11	12	12	17	23	14

Kürzester Pfad (Länge 14) im Graphen G_1:



Literaturverzeichnis

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (o. J.). “Algorithmen – Eine Einführung” (3. Auflage). Berlin: De Gruyter.

[2] Sedgewick, R., & Wayne, K. (2016). “Algorithmen – Eine umfassende Einführung”. 4. Auflage. München: Pearson.

[3] Papadimitriou, C. H. (1994). “Computational Complexity”. Reading, MA: Addison-Wesley

[4] Reinhard Diestel: “Graph Theory”. Springer, 5th edition

[5] Robin J. Wilson: “Introduction to Graph Theory”. Pearson, 5th edition

[6] Douglas B. West: “Introduction to Graph Theory”. Pearson, 2nd edition

[7] J.A. Bondy, U.S.R. Murty: “Graph Theory”. Springer Graduate Texts in Mathematics