Prinzipien der Programmierung

Daniel Merkle

Wintersemester 2024

(Teil 07)

Programmierparadigmen

- Ein **Programmierparadigma** ist ein grundlegender Stil der Programmierung, der bestimmt, wie Lösungen für Probleme formuliert und Programme strukturiert werden.
- Es bietet einen Rahmen von Konzepten, Methoden und Regeln, die die Art und Weise definieren, wie Programme entwickelt und geschrieben werden.
- Unterschiedliche Paradigmen bieten verschiedene Sichtweisen auf Probleme und beeinflussen, wie Programme ausgeführt und gesteuert werden.

Lernziele

- Verstehen, was ein Programmierparadigma ist.
- Die Unterschiede zwischen imperativer, prozeduraler und objektorientierter Programmierung klar erkennen.
- Wissen, wann welches Paradigma sinnvoll eingesetzt werden kann.
- Einblicke in weitere Programmierparadigmen erhalten.

Imperatives Programmieren

- **Imperatives Programmieren** ist ein Paradigma, bei dem Programme aus Anweisungen bestehen, die den Computer dazu bringen, bestimmte Aktionen auszuführen.
- Der Fokus liegt auf dem **"Wie"** ein Ergebnis erreicht wird.
- Es beschreibt Schritt für Schritt, wie der Computer ein Problem lösen soll, durch die Veränderung des Programmzustands.

Eigenschaften des imperativen Programmierens

- **Sequentielle Ausführung**: Anweisungen werden in der Reihenfolge ausgeführt, in der sie geschrieben sind.
- **Zustandsänderungen**: Variablen können verändert werden, was den Zustand des Programms beeinflusst.
- **Kontrollstrukturen**: Verwendung von Schleifen (`for` , `while`), Bedingungen (`if` , `else`) und Sprunganweisungen (`break` , `continue`).
- **Abstraktionsniveau**:
  - **Niedrigstufige imperative Sprachen** (z. B. C, Assembler):
    - **Direkte Manipulation von Hardware und Speicher**.
  - **Höherstufige imperative Sprachen** (z. B. Java, Python):
    - **Abstraktion der Speicherverwaltung**.

Imperatives Programmieren: Niedrigstufige Sprachen

- **Direkte Manipulation von Hardware und Speicher**:
  - Zugriff auf Speicheradressen und Hardwarekomponenten.
  - Verwendung von Zeigern und direktem Speicherzugriff.
- **Beispiel in C**:

```c
int a = 10;
int *p = &a; // 'p' ist ein Zeiger auf die Speicheradresse von 'a'
printf("Wert von a: %d\n", *p);
```

Imperatives Programmieren: Höherstufige Sprachen

- **Abstraktion der Speicherverwaltung**:
  - Speicherzugriff wird vom Laufzeitsystem oder der virtuellen Maschine verwaltet.
  - Kein direkter Zugriff auf Speicheradressen.
- **Beispiel in Java**:

```
int a = 10; // 'a' ist eine Variable, Speicherzugriff wird von der JVM verwa
System.out.println("Wert von a: " + a);
```

Prozedurale Programmierung

- **Prozedurale Programmierung** ist ein Ansatz innerhalb des imperativen Paradigmas, der Programme durch die Verwendung von **Prozeduren** oder **Funktionen** strukturiert.
- Ziel ist es, den Code in kleinere, wiederverwendbare und übersichtliche Einheiten zu gliedern.
- Fördert die **Modularität** und **Wiederverwendbarkeit** von Code.

Merkmale der prozeduralen Programmierung

- **Prozeduren/Funktionen**: Definieren von Codeblöcken mit spezifischer Funktionalität.
- **Parameterübergabe**: Daten werden an Funktionen übergeben und zurückgegeben.
- **Top-Down-Ansatz**: Beginnt mit einem Hauptprogramm und zerlegt Aufgaben in Unterfunktionen.
- **Keine Objekte**: Daten und Funktionen sind getrennt, keine Kapselung wie in OOP.

Beispiel: Prozedurale Programmierung in C

```c
#include <stdio.h>

void greet(char name[]) {
    printf("Hallo, %s!\n", name);
}

int main() {
    greet("Anna");
    greet("Ben");
    return 0;
}
```

Beispiel (prozedural)

- Daten und Funktionen sind **getrennt**.
- Keine Kapselung in Klassen.

```
// Daten
int radius = 5;

// Funktion
double berechneKreisflaeche(int r) {
    return Math.PI * r * r;
}

// Nutzung
double flaeche = berechneKreisflaeche(radius);
System.out.println("Fläche: " + flaeche);
```

Beispiel (OOP)

- Daten und Funktionen werden in einer **Klasse** zusammengefasst.
- Vorteil: **Kapselung** und besserer Code-Überblick.

```
class Kreis {
    int radius;

    Kreis(int radius) {
        this.radius = radius;
    }

    double berechneFlaeche() {
        return Math.PI * radius * radius;
    }
}

// Nutzung
Kreis kreis = new Kreis(5);
double flaeche = kreis.berechneFlaeche();
System.out.println("Fläche: " + flaeche);
```

grober Vergleich der Ansätze

Prozedural vs. Objektorientiert

Merkmal	Prozedural	Objektorientiert
Daten und Verhalten	Getrennt	Gemeinsam in Klassen
Kapselung	Keine	Ja
Erweiterbarkeit	Schwerer	Einfacher durch Vererbung

Vergleich: Imperativ vs. Prozedural

- **Imperatives Programmieren** ist der übergeordnete Stil, bei dem der Programmfluss durch Anweisungen bestimmt wird.
- **Prozedurale Programmierung** ist eine Methode innerhalb des imperativen Paradigmas, die zusätzliche Struktur durch Prozeduren bietet.
- **Hauptunterschied**:
  - Imperativ: Fokus auf Anweisungsfolge und Zustandsänderungen.
  - Prozedural: Fokus auf Strukturierung durch Funktionen/Prozeduren zur Verbesserung der Modularität.

Objektorientierte Programmierung (OOP)

- **Objektorientierte Programmierung** ist ein Paradigma, das Programme als Sammlung von **Objekten** modelliert, die miteinander interagieren.
- Objekte kombinieren **Daten** (Attribute) und **Methoden** (Funktionen).
- Zentrale Konzepte:
  - **Kapselung**
  - **Abstraktion**
  - **Vererbung**
  - **Polymorphie**

Grundkonzepte der OOP

- **Klassen**: Baupläne für Objekte, definieren Eigenschaften und Verhalten.
- **Objekte**: Instanzen von Klassen mit eigenem Zustand.
- **Kapselung**: Verbergen der internen Zustände und Bereitstellung von Schnittstellen.
- **Vererbung**: Ermöglicht das Erben von Eigenschaften und Methoden einer übergeordneten Klasse.
- **Polymorphie**: Fähigkeit, je nach Kontext unterschiedliche Implementierungen zu verwenden.

Beispiel: OOP in Java

```java
public class Fahrzeug {
    protected String marke;

    public Fahrzeug(String marke) {
        this.marke = marke;
    }

    public void fahren() {
        System.out.println(marke + " fährt.");
    }
}

public class Auto extends Fahrzeug {
    public Auto(String marke) {
        super(marke);
    }

    @Override
    public void fahren() {
        System.out.println(marke + " Auto fährt auf der Straße.");
```

Unterschiede zwischen prozeduraler und objektorientierter Programmierung

- **Daten und Funktionen**:
  - Prozedural: Daten und Funktionen sind getrennt.
  - OOP: Daten und Funktionen sind in Objekten zusammengefasst.
- **Modularität**:
  - Prozedural: Durch Funktionen und Module.
  - OOP: Durch Klassen und Objekte.
- **Wiederverwendbarkeit**:
  - Prozedural: Funktionen können wiederverwendet werden.
  - OOP: Vererbung und Polymorphie fördern Wiederverwendbarkeit auf höherer Ebene.
- **Abstraktion**:
  - OOP bietet stärkere Mechanismen zur Abstraktion durch Klassen und Schnittstellen.

Wann welches Paradigma verwenden?

- **Imperatives/Prozedurales Programmieren**:
  - Geeignet für kleinere Projekte oder Systeme mit klar definierten Abläufen.
  - Wenn Performance und Ressourcenmanagement kritisch sind.
- **Objektorientierte Programmierung**:
  - Ideal für komplexe Systeme mit vielen interagierenden Komponenten.
  - Wenn Wartbarkeit, Erweiterbarkeit und Modellierung realer Entitäten wichtig sind.
- **Es hängt vom Anwendungsfall, den Anforderungen und den Vorlieben des Entwicklerteams ab.**

Zusammenfassung der Paradigmen

- **Imperatives Programmieren**:
  - Grundlegendes Paradigma, auf dem andere aufbauen.
  - Fokus auf direkte Anweisungen und Zustandsänderungen.
  - Umfasst sowohl niedrig- als auch höherstufige Sprachen.
- **Prozedurale Programmierung**:
  - Erweiterung des imperativen Paradigmas durch Strukturierung mit Funktionen.
  - Bietet Modularität und Code-Wiederverwendung.
- **Objektorientierte Programmierung**:
  - Modelliert Systeme durch Objekte, die Daten und Verhalten kombinieren.
  - Unterstützt Kapselung, Vererbung und Polymorphie.

Weitere Programmierparadigmen

- **Funktionales Programmieren**:
  - Fokus auf die Berechnung durch Auswertung von Funktionen.
  - Vermeidung von Zustandsänderungen und Seiteneffekten.
  - Beispiele: Haskell, Erlang, F#.
- **Logisches Programmieren**:
  - Programme bestehen aus logischen Aussagen und Regeln.
  - Der Computer zieht Schlussfolgerungen aus gegebenen Fakten.
  - Beispiel: Prolog.
- **Deklaratives Programmieren**:
  - Beschreibt das **"Was"**, nicht das **"Wie"**.
  - Beispiele: SQL, HTML, Regex.

Funktionales Programmieren

- **Funktionen** sind die grundlegenden Bausteine.
- **Reinheit**: Funktionen sind frei von Seiteneffekten.
- **Unveränderlichkeit**: Datenstrukturen werden nicht verändert, sondern neue erstellt.
- **Höhere Ordnung**: Funktionen können Funktionen als Argumente nehmen oder zurückgeben.

Beispiel: Funktionales Programmieren in Haskell

```haskell
-- Quadratzahl berechnen
square x = x * x

-- Liste von Zahlen quadrieren
squares = map square [1, 2, 3, 4, 5] -- Ergebnis: [1,4,9,16,25]
```

Logisches Programmieren

- **Programme** werden als eine Menge von Fakten und Regeln formuliert.
- **Ziel**: Antworten auf Anfragen durch logische Schlussfolgerungen finden.
- **Kein** expliziter Kontrollfluss.
- **Anwendung**: KI, Wissensdatenbanken.

Beispiel: Logisches Programmieren in Prolog

```
mutter(viktoria, eduard).
mutter(viktoria, alice).
vater(albert, eduard).
vater(albert, alice).

elternteil(X, Y) :- mutter(X, Y).
elternteil(X, Y) :- vater(X, Y).
```

Komplexeres Beispiel: Wer hat welches Haustier? 🐾

- **Drei Personen**: Anna, Bob und Clara.
- **Drei Haustiere**: Hund, Katze, Vogel.
- Die Hinweise:
    i. Clara hat den Vogel. 🐦
    ii. Anna mag keine Katzen. ❌🐱
    iii. Bob hat keinen Hund. ❌🐶

Aufgabe: Finde heraus, wer welches Haustier hat.

Prolog-Programm: Definition der Lösung

```
% Struktur: Person - Haustier
solve(Haustiere) :-
    Haustiere = [anna-Haustier1, bob-Haustier2, clara-Haustier3],
    member(Haustier1, [hund, katze, vogel]),
    member(Haustier2, [hund, katze, vogel]),
    member(Haustier3, [hund, katze, vogel]),
    % Bedingung: Clara hat den Vogel
    member(clara-vogel, Haustiere),
    % Bedingung: Anna mag keine Katzen
    \+ member(anna-katze, Haustiere),
    % Bedingung: Bob hat keinen Hund
    \+ member(bob-hund, Haustiere),
    % Jede Person hat ein anderes Haustier
    all_different([Haustier1, Haustier2, Haustier3]).

% Hilfsregel: Alle Elemente in der Liste sind verschieden
all_different([]).
all_different([X | Rest]) :-
    \+ member(X, Rest),
    all_different(Rest).
```

Prolog-Programm: Abfrage und Ergebnis

Abfrage:

```
?- solve(Haustiere).
```

Ergebnis:

```
Haustiere = [anna-hund, bob-katze, clara-vogel].
```

Interpretation der Lösung:

- **Anna** hat den Hund 🐶.
- **Bob** hat die Katze 🐱.
- **Clara** hat den Vogel 🐦.

Was macht Prolog hier?

1. **Erzeuge mögliche Kombinationen**:
   - `member/2`: Überprüft, welches Haustier zu welcher Person passt.
   - Kombiniert alle möglichen Zuordnungen.
2. **Prüfe Bedingungen**:
   - Ausschlussregeln (`\+`): Entferne ungültige Kombinationen.
   - Regeln: Clara hat den Vogel, Anna mag keine Katzen, etc.
3. **Finde die Lösung**:
   - Prolog sucht nach einer Kombination, die alle Bedingungen erfüllt.

Wie funktioniert `member/2`?

Eigene Implementierung in Prolog

```
% Basisfall: X ist das erste Element der Liste
member(X, [X | _]).

% Rekursiver Fall: Suche X in der Restliste
member(X, [_ | Tail]) :-
    member(X, Tail).
```

Warum Prolog?

- **Logik statt Imperativ**:
  - Sie beschreiben **was** wahr sein soll, nicht **wie** die Lösung berechnet wird.
- **Anwendungen**:
  - Rätsel und Puzzles.
  - Wissensbasierte Systeme.
  - Künstliche Intelligenz.

👉 **Prolog denkt logisch für Sie!**

Nachteile von Prolog

1. **Schwierige Laufzeitvorhersage**:
   - Prolog sucht systematisch durch den Suchraum.
   - Die Laufzeit kann bei großen Probleminstanzen unvorhersehbar lang werden.
   - Backtracking kann ineffizient sein, wenn viele Kombinationen geprüft werden müssen.
2. **Begrenzte Anwendungsgebiete**:
   - Prolog eignet sich hervorragend für logische Probleme und Wissensbasen.
   - Für numerisch intensive oder hardware-nahe Anwendungen ist Prolog weniger geeignet.
3. **Eingeschränkte Lesbarkeit für komplexe Programme**:
   - Regeln und Logik können bei größeren Programmen schwer verständlich werden.
   - Fehlerdiagnose (z. B. warum eine Regel fehlschlägt) ist nicht immer intuitiv.
4. **Nicht deklarative Aspekte**:
   - Trotz des deklarativen Ansatzes müssen manchmal Implementierungsdetails wie das Setzen von Cut-Operatoren (`!`) berücksichtigt werden, was die Eleganz vermindert.
5. **Kleine Community und Werkzeugunterstützung**:
   - Weniger verbreitet als andere Sprachen, wodurch die Anzahl der Bibliotheken und Tools begrenzt ist.

Deklaratives Programmieren

- **Beschreibt** das gewünschte Ergebnis, nicht der Weg dorthin.
- **Der Computer** findet den besten Weg, um das Ergebnis zu erzielen.
- **Anwendungsbereiche**:
  - Datenbankabfragen (SQL).
  - Konfigurationsmanagement (Ansible).
  - Mustererkennung (Reguläre Ausdrücke).

Beispiel: Deklaratives Programmieren mit SQL

```sql
SELECT name, preis FROM produkte WHERE preis < 100 ORDER BY preis ASC;
```

Integer Lineare Programmierung (ILP) als Beispiel für deklaratives Programmieren

- **Deklaratives Programmieren**:
  - Beschreibt **was** erreicht werden soll, nicht **wie** es umgesetzt wird.
- **Integer Lineare Programmierung**:
  - Mathematisches Optimierungsverfahren mit linearen Gleichungen und Ungleichungen.
  - Ziel ist es, eine lineare Zielfunktion zu maximieren oder zu minimieren unter gegebenen Nebenbedingungen.
  - Variablen sind auf ganze Zahlen beschränkt.

Beispiel eines ILP-Problems

```
Maximiere:     z = 5x + 4y

Unter den Nebenbedingungen:
    2x + 3y ≤ 12
    x + y ≤ 5
    x ≥ 0, y ≥ 0
    x, y ∈ ℕ₀
```

- **Erläuterung**:
  - **Zielfunktion**: Definiert das Optimierungsziel.
  - **Nebenbedingungen**: Einschränkungen, die erfüllt sein müssen.
- **Deklarativ**:
  - Der Programmierer spezifiziert das Problem; der Solver findet die Lösung ohne explizite Algorithmen zu programmieren.

Problemstellung

Produktionsplanung mit ILP

Ein Unternehmen möchte drei Produkte A, B und C in zwei Fabriken F_1 und F_2 effizient produzieren.

Ziel:

Minimiere die Produktionskosten.

Gegeben:

- **Produktionsanforderungen:**
  - A: Mindestens 30 Einheiten
  - B: Mindestens 20 Einheiten
  - C: Mindestens 40 Einheiten
- **Kapazitäten der Fabriken:**
  - F_1: Maximal 50 Einheiten
  - F_2: Maximal 60 Einheiten
- **Produktionskosten (EUR pro Einheit):**

Produkt / Fabrik	F_1	F_2
A	4	3
B	5	4
C	7	6

Mathematisches Modell

Zielfunktion:

$$Z = 4x_{A1} + 3x_{A2} + 5x_{B1} + 4x_{B2} + 7x_{C1} + 6x_{C2} \quad \text{(Minimiere die Kosten)}$$

Nebenbedingungen:

1. **Produktionsanforderungen:**

   $$x_{A1} + x_{A2} \geq 30, \quad x_{B1} + x_{B2} \geq 20, \quad x_{C1} + x_{C2} \geq 40$$

2. **Kapazitäten:**

   $$x_{A1} + x_{B1} + x_{C1} \leq 50, \quad x_{A2} + x_{B2} + x_{C2} \leq 60$$

3. **Nicht-Negativität und Ganzzahligkeit:**

   $$x_{ij} \geq 0 \quad \text{und ganzzahlig}$$

Lösung mit Gurobi/Python

```python
from gurobipy import Model, GRB

# Modell erstellen
model = Model("Produktionsplanung")

# Variablen
x_A1 = model.addVar(vtype=GRB.INTEGER, name="x_A1")
x_A2 = model.addVar(vtype=GRB.INTEGER, name="x_A2")
x_B1 = model.addVar(vtype=GRB.INTEGER, name="x_B1")
x_B2 = model.addVar(vtype=GRB.INTEGER, name="x_B2")
x_C1 = model.addVar(vtype=GRB.INTEGER, name="x_C1")
x_C2 = model.addVar(vtype=GRB.INTEGER, name="x_C2")

# Zielfunktion
model.setObjective(
    4 * x_A1 + 3 * x_A2 +
    5 * x_B1 + 4 * x_B2 +
    7 * x_C1 + 6 * x_C2,
    GRB.MINIMIZE
)
```

Lösung

```
Optimal solution found (tolerance 1.00e-04)
Best objective 4.400000000000e+02, best bound 4.400000000000e+02, gap 0.0000%
x_A1:  0.0
x_A2: 30.0
x_B1: 20.0
x_B2:  0.0
x_C1: 10.0
x_C2: 30.0
Optimale Kosten: 440.0
```

Ereignisgesteuertes Programmieren

- **Reaktion** auf Ereignisse, die von Benutzeraktionen oder Systemereignissen ausgelöst werden.
- **Programme** warten auf Ereignisse und führen entsprechende Handler aus.
- **Anwendungen**:
  - GUI-Entwicklung.
  - Webentwicklung (JavaScript).

Beispiel: Ereignisgesteuertes Programmieren in JavaScript

```
document.getElementById("myButton").addEventListener("click", function() {
    alert("Button wurde geklickt!");
});
```

Nebenläufiges Programmieren

- **Mehrere** Prozesse oder Threads werden gleichzeitig ausgeführt.
- **Ziel**: Bessere Nutzung von Mehrkernprozessoren und Verbesserung der Performance.
- **Herausforderungen**:
  - Synchronisation von Threads.
  - Vermeidung von Race Conditions.

Beispiel: Nebenläufiges Programmieren in Go

```go
package main

import (
    "fmt"
    "time"
)

func output(text string) {
    for i := 0; i < 3; i++ {
        fmt.Println(text)
        time.Sleep(100 * time.Millisecond)
    }
}

func main() {
    go output("Goroutine")
    output("Main Funktion")
}
```

Paradigmen können kombiniert werden

- **Moderne Sprachen** unterstützen oft mehrere Paradigmen.
- **Beispiele**:
  - **Python**: Imperativ, objektorientiert, funktional.
  - **Scala**: Objektorientiert, funktional.
  - **JavaScript**: Imperativ, funktional, ereignisgesteuert.
  - **Java**: Imperativ, objektorientiert, funktional, nebenläufig.
- **Vorteil**: Flexibilität, um das beste Werkzeug für die jeweilige Aufgabe zu wählen.

Prinzipien der Programmierung und Programmierparadigmen

- **Prinzipien** wie DRY (Don't Repeat Yourself), KISS (Keep It Simple, Stupid) und Modularität gelten unabhängig vom Paradigma.
- **Anwendung** dieser Prinzipien verbessert die Qualität und Wartbarkeit von Code.
- **Paradigmen** bieten unterschiedliche Werkzeuge, um diese Prinzipien umzusetzen.

DRY - Don't Repeat Yourself

- **Ziel**: Redundanz vermeiden, um Konsistenz zu gewährleisten und Wartung zu erleichtern.
- **Anwendung in Prozeduralem Programmieren**:
  - Nutzung von Funktionen, um wiederholten Code zu vermeiden.
- **Anwendung in OOP**:
  - Nutzung von Vererbung und Schnittstellen.
  - Wiederverwendung von Klassen und Methoden.

Beispiel: DRY in OOP

```java
public abstract class Tier {
    protected String name;

    public Tier(String name) {
        this.name = name;
    }

    public abstract void geraeuschMachen();

    public void schlafen() {
        System.out.println(name + " schläft.");
    }
}

public class Katze extends Tier {
    public Katze(String name) {
        super(name);
    }

    @Override
    public void geraeuschMachen() {
        System.out.println(name + " miaut.");
    }
}
```

KISS - Keep It Simple, Stupid

- **Ziel**: Einfachheit bevorzugen, um Verständlichkeit und Wartbarkeit zu erhöhen.
- **Anwendung**:
  - **Imperativ**: Klare und direkte Anweisungen ohne unnötige Komplexität.
  - **Funktional**: Nutzung einfacher, reiner Funktionen.

Ockhams Rasiermesser

Beispiel: KISS in Funktionalem Programmieren

```haskell
-- Berechnung der Fakultät
fakultät 0 = 1
fakultät n = n * fakultät (n - 1)
```

Separation of Concerns

- **Ziel**: Aufteilung eines Programms in unterschiedliche Bereiche, die jeweils eine spezifische Funktionalität bieten.
- **Vorteile**:
  - Erleichtert Wartung und Erweiterung.
  - Bessere Übersicht und Modularität.

Modularität

- **Ziel**: Code in unabhängige, austauschbare Module aufzuteilen.
- **Vorteile**:
  - Erleichtert Testen und Debugging.
  - Module können unabhängig entwickelt und aktualisiert werden.
  - Modularität ist die konkrete Umsetzung von Separation on Concerns

Beispiel: MVC-Pattern

```
Model (Daten)
|
Controller (Logik)
|
View (Präsentation)
```

Abstraktion

- **Ziel**: Komplexität reduzieren, indem Details verborgen und nur notwendige Informationen präsentiert werden.
- **Anwendung in OOP**:
  - Verwendung von abstrakten Klassen und Schnittstellen.
  - Ermöglicht Fokus auf **Was** eine Komponente tut, nicht **Wie** sie es tut.

Zusammenfassung

- **Programmierparadigmen** bieten verschiedene Ansätze zur Problemlösung.
- **Imperatives Programmieren** bildet die Grundlage vieler Sprachen.
- **Prozedurale Programmierung** strukturiert Code durch Funktionen.
- **Objektorientierte Programmierung** modelliert Systeme durch Objekte.
- **Prinzipien** wie DRY und KISS sind universell und verbessern die Codequalität.
- Die **Wahl des Paradigmas** hängt von vielen Faktoren ab, einschließlich Projektanforderungen und Teamexpertise.

Ausblick

- **Weiterführende Themen**:
  - Tieferes Eintauchen in funktionales und logisches Programmieren.
  - Untersuchung von Paradigmen wie **reaktive Programmierung**.
- **Praktische Anwendung**:
  - Entwicklung kleiner Projekte in verschiedenen Paradigmen zur Vertiefung des Verständnisses.

Algorithmen

Übersicht

- **Einführung in Algorithmen**
- **Historischer Hintergrund**
- **Wichtigkeit der Effizienz**
- **Sortieralgorithmen**
  - Selection Sort
  - Andere Sortierverfahren (kurze Übersicht)
- **Suchalgorithmen**
  - Lineare Suche
  - Binäre Suche
- **Laufzeitkomplexität und Effizienz**
- **Algorithmen in der Praxis**
  - Machine Learning
  - Graphalgorithmen
  - Chemieinformatik
- **Zusammenfassung und Ausblick**

Lernziele

- Verstehen, was ein **Algorithmus** ist und wie er in der Informatik verwendet wird.
- Vertrautheit mit dem **Selection-Sort-Algorithmus** und seiner Funktionsweise.
- Erklären können, wie **lineare** und **binäre Suchalgorithmen** funktionieren.
- Grundlegendes Verständnis der **Laufzeitkomplexität** und eine kurze Einführung zur **Groß-O-Notation**.
- Kennenlernen von Anwendungen von Algorithmen in verschiedenen Bereichen.

Herkunft des Begriffs *Algorithmus*

- Der Begriff **Algorithmus** leitet sich vom Namen des persischen Mathematikers **Al-Chwarizmi** ab.
- **Al-Chwarizmi** lebte im 9. Jahrhundert und verfasste bedeutende Werke über Mathematik und Astronomie.
- Sein Buch über **indisch-arabische Zahlensysteme** und Rechenmethoden verbreitete das dezimale Zahlensystem.
- Das Wort *Algorithmus* wurde später als Bezeichnung für ein systematisches Verfahren oder eine Methode zur Lösung von Problemen übernommen.

Was ist ein Algorithmus (vereinfacht)?

- Ein (deterministischer) **Algorithmus** ist eine eindeutige und endliche Folge von Anweisungen zur Lösung eines Problems.
- Wichtige Eigenschaften von Algorithmen (unvollständig):
  - **Determiniertheit** Gleiche Startbedingungen führen zur gleichen Ausgabe.
  - **(Nicht-)Determinismus** Der nächste Handlungsschritt ist eindeutig definiert.
  - **Effektivität**: Der Effekt jeder Anweisung ist eindeutig festgelegt.
  - **Finitheit**: Der Algorithmus hat eine endliche Beschreibung.
- **Beispiele**:
  - Kochrezepte
  - Mathematische Verfahren (z.B. Euklidischer Algorithmus)
  - Computerprogramme
- Abgrenzung / Übergang zu **Heuristik** (oft randomisiert)

Beispiel: Wege durch ein Labyrinth: Verschiedene Ansätze

- Ein Labyrinth symbolisiert ein Problem mit mehreren möglichen Lösungswegen.
- Vier Ansätze zur Problemlösung:
  - 🛤️ Deterministisch
  - 🎲 Probabilistisch
  - ⚙️ Parallel
  - 🌀 Nicht-deterministisch

Deterministische Suche

- **Vorgehen**:
  - Systematisch alle möglichen Wege ausprobieren, z. B. immer zuerst nach rechts, dann nach unten.
- **Eigenschaften**:
  - **Vorhersehbar**: Der Algorithmus folgt immer denselben Regeln.
  - je nach Ansatz/Algorithmus, potentiell **langsam**, da alle möglichen Wege nacheinander geprüft werden.

Szenario:

1. Betritt das Labyrinth.
2. Probiere jeden Pfad streng nach festgelegten Regeln aus.
3. Finde den Ausgang.

Probabilistische Suche

- **Vorgehen**:
  - Wähle bei jeder Kreuzung **zufällig** einen Weg.
- **Eigenschaften**:
  - **Unvorhersehbar**, da Entscheidungen auf Zufall beruhen.
  - Kann schnell eine Lösung finden – oder auch nicht.
  - Funktioniert oft besser, wenn man mehrere Durchläufe macht.

Szenario:

1. Stehe an einer Kreuzung.
2. Würfle oder entscheide zufällig, wohin du gehst.
3. Wenn du den Ausgang nicht findest, fange von vorne an.

Parallele Suche

- **Vorgehen**:
  - Probiere alle möglichen Wege gleichzeitig aus.
- **Eigenschaften**:
  - **Effizient**, da alle Pfade gleichzeitig untersucht werden (abhängig von der Definition von "Effizienz")
  - Benötigt jedoch mehr **Ressourcen** (z. B. viele Sucher, parallele Prozesse).

Szenario:

1. Schicke an jeder Kreuzung einen Sucher in jede mögliche Richtung.
2. Der erste Sucher, der den Ausgang findet, beendet die Suche.

Nicht-Deterministische Suche

- **Vorgehen**:
  - Ein **Orakel** sagt an jeder Kreuzung den richtigen Weg.
- **Eigenschaften**:
  - **Perfekt**: Finde immer den Ausgang, wenn es einen gibt.
  - Wird in der Theorie verwendet, nicht in der Praxis.

Szenario:

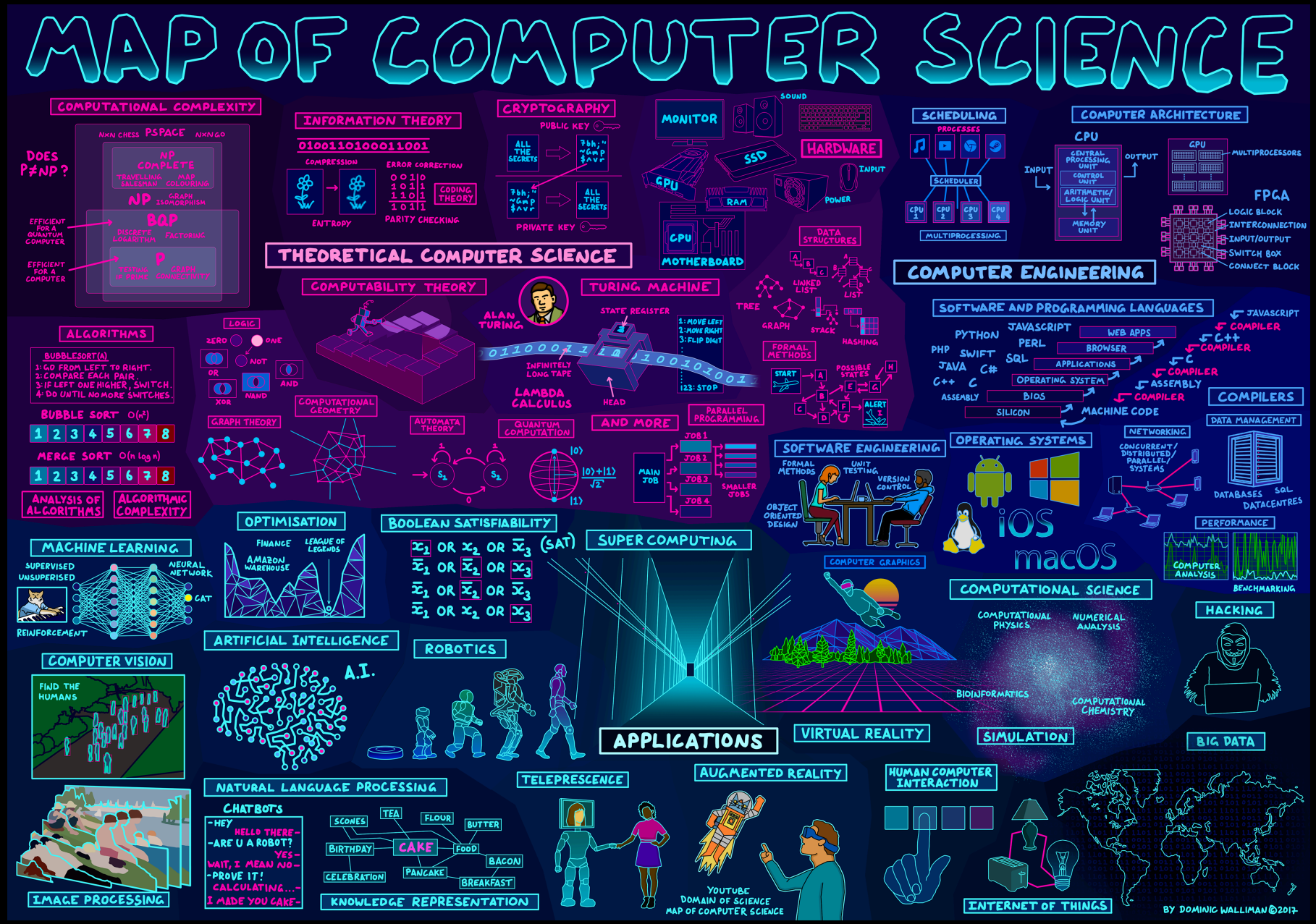
1. Du betrittst das Labyrinth.
2. Das Orakel zeigt dir sofort, wohin du gehen musst, um den Ausgang zu erreichen.
3. Beende die Suche ohne unnötige Schritte.

Vergleich

Ansatz	Eigenschaften	Anwendung
🛤️ Deterministisch	Vorhersehbar	"klassische" Probleme
🎲 Probabilistisch	Unvorhersehbar, potenziell schnell, potentiell nicht optimal	Heuristische Ansätze
⚙️ Parallel	"Effizient", ressourcenintensiv	Große/viele Probleminstanzen
🌀 Nicht-deterministisch	Perfekt, theoretisch	Komplexitätstheorie, Analyse von Algorithmen

Bedeutung von Algorithmen in der Informatik

- **Algorithmen** sind das Herzstück jedes Computerprogramms, jedes Betriebssystems, jeder Software, ...
- Sie bestimmen, **wie effizient** eine Aufgabe gelöst wird.
- Anwendung in vielen Bereichen:
  - **Datenverarbeitung**
  - **Netzwerkkommunikation**
  - **Künstliche Intelligenz**
  - **Datenbanken**


MAP OF COMPUTER SCIENCE
COMPUTATIONAL COMPLEXITY
DOES P≠NP?
NXN CHESS
PSPACE
NXNGO
NP COMPLETE
TRAVELLING SALESMAN
MAP COLOURING
NP
GRAPH ISOMORPHISM
BQP
DISCRETE LOGARITHM
FACTORING
P
TESTING IF PRIME
GRAPH CONNECTIVITY
EFFICIENT FOR A QUANTUM COMPUTER
EFFICIENT FOR A COMPUTER
INFORMATION THEORY
0100110100011001
COMPRESSION
ENTROPY
ERROR CORRECTION
CODING THEORY
PARITY CHECKING
CRYPTOGRAPHY
PUBLIC KEY
ALL THE SECRETS
PRIVATE KEY
THEORETICAL COMPUTER SCIENCE
COMPUTABILITY THEORY
ALAN TURING
TURING MACHINE
STATE REGISTER
INFINITELY LONG TAPE
HEAD
1: MOVE LEFT
2: MOVE RIGHT
3: FLIP DIGIT
123: STOP
LAMBDA CALCULUS
AND MORE
ALGORITHMS
BUBBLESORT(A)
1: GO FROM LEFT TO RIGHT.
2: COMPARE EACH PAIR.
3: IF LEFT ONE HIGHER, SWITCH.
4: DO UNTIL NO MORE SWITCHES.
BUBBLE SORT O(n²)
MERGE SORT O(n log n)
ANALYSIS OF ALGORITHMS
ALGORITHMIC COMPLEXITY
LOGIC
ZERO
ONE
OR
NOT
NAND
AND
XOR
GRAPH THEORY
COMPUTATIONAL GEOMETRY
AUTOMATA THEORY
QUANTUM COMPUTATION
PARALLEL PROGRAMMING
MAIN JOB
JOB 1
JOB 2
JOB 3
JOB 4
SMALLER JOBS
MONITOR
SOUND
SSD
HARDWARE
INPUT
CPU
RAM
POWER
MOTHERBOARD
DATA STRUCTURES
LINKED LIST
LIST
TREE
GRAPH
STACK
HASHING
FORMAL METHODS
START
POSSIBLE STATES
ALERT
SCHEDULING
PROCESSES
SCHEDULER
MULTIPROCESSING
COMPUTER ARCHITECTURE
CPU
CENTRAL PROCESSING UNIT
CONTROL UNIT
ARITHMETIC/ LOGIC UNIT
MEMORY UNIT
INPUT
OUTPUT
GPU
MULTIPROCESSORS
FPGA
LOGIC BLOCK
INTERCONNECTION
INPUT/OUTPUT
SWITCH BOX
CONNECT BLOCK
COMPUTER ENGINEERING
SOFTWARE AND PROGRAMMING LANGUAGES
PYTHON
JAVASCRIPT
PHP
SWIFT
PERL
SQL
JAVA
C#
C++
C
ASSEMBLY
WEB APPS
BROWSER
APPLICATIONS
OPERATING SYSTEM
BIOS
SILICON
MACHINE CODE
JAVASCRIPT
COMPILER
C++
C
ASSEMBLY
COMPILERS
DATA MANAGEMENT
DATABASES
SQL
DATACENTRES
NETWORKING
CONCURRENT/ DISTRIBUTED/ PARALLEL/ SYSTEMS
PERFORMANCE
COMPUTER ANALYSIS
BENCHMARKING
SOFTWARE ENGINEERING
FORMAL METHODS
UNIT TESTING
VERSION CONTROL
OBJECT ORIENTED DESIGN
OPERATING SYSTEMS
iOS
macOS
MACHINE LEARNING
SUPERVISED
UNSUPERVISED
NEURAL NETWORK
CAT
REINFORCEMENT
OPTIMISATION
FINANCE
LEAGUE OF LEGENDS
AMAZON WAREHOUSE
BOOLEAN SATISFIABILITY
(SAT)
SUPER COMPUTING
COMPUTER GRAPHICS
COMPUTATIONAL SCIENCE
COMPUTATIONAL PHYSICS
NUMERICAL ANALYSIS
BIOINFORMATICS
COMPUTATIONAL CHEMISTRY
HACKING
ARTIFICIAL INTELLIGENCE
A.I.
ROBOTICS
COMPUTER VISION
FIND THE HUMANS
APPLICATIONS
VIRTUAL REALITY
SIMULATION
BIG DATA
IMAGE PROCESSING
NATURAL LANGUAGE PROCESSING
CHATBOTS
-HEY
-HELLO THERE-
-ARE U A ROBOT?
-YES-
-WAIT, I MEAN NO-
-PROVE IT!
-CALCULATING...-
-I MADE YOU CAKE-
KNOWLEDGE REPRESENTATION
SCONES
TEA
FLOUR
BUTTER
CAKE
BIRTHDAY
FOOD
BACON
CELEBRATION
PANCAKE
BREAKFAST
TELEPRESCENCE
AUGMENTED REALITY
YOUTUBE
DOMAIN OF SCIENCE
MAP OF COMPUTER SCIENCE
HUMAN COMPUTER INTERACTION
INTERNET OF THINGS
BY DOMINIC WALLIMAN ©2017

Karte von Information is Beautiful Awards

Video dazu

Effizienz von Algorithmen

- Die **Effizienz** eines Algorithmus bezieht sich auf den Ressourcenverbrauch, z.B.,
  - **Zeit** (Laufzeit)
  - **Speicherplatz**
- **Warum ist Effizienz wichtig?**
  - Ressourcen sind begrenzt (z.B. CPU-Zeit, Speicher).
  - Effiziente Algorithmen ermöglichen die Verarbeitung großer Datenmengen.
  - **Beispiel**:
    - Eine Wettervorhersage muss rechtzeitig bereitstehen, um nützlich zu sein.
    - Das menschliche Genom hat 3.2 Milliarden Nukleotidbasen.
    - Es gibt Datenbanken mit ca. 100 Millionen dokumentierten und bekannten chemischen Reaktionen.

Informationen sortieren

- **Sortieren** ist ein grundlegendes Problem in der Informatik.
- Unsortierte Daten erschweren das Auffinden und Verarbeiten von Informationen.
- **Anwendungen**:
  - Datenbanksysteme
  - Suchalgorithmen
  - Datenanalyse

Sortieralgorithmen

- Es gibt viele Sortieralgorithmen mit unterschiedlichen Eigenschaften:
  - **Selection Sort**
  - **Insertion Sort**
  - **Merge Sort**
  - **Quick Sort**
- Die Wahl des Algorithmus hängt von Faktoren wie Datengröße und -struktur ab.

Selection Sort

- **Funktionsweise**:
  - Finde das kleinste Element in der unsortierten Liste.
  - Tausche es mit dem ersten Element der unsortierten Liste.
  - Wiederhole diesen Vorgang für die restlichen Elemente.
- **Eigenschaften**:
  - Einfach zu verstehen und zu implementieren.
  - Laufzeitkomplexität: $O(n^2)$
  - Nicht effizient für große Datensätze.

Visualisierung von Selection Sort

Visualisierung

Übungsaufgabe!

Suchalgorithmen

- **Suche** ist eine weitere grundlegende Aufgabe in der Informatik.
- **Zwei grundlegende Suchalgorithmen**:
  - **Lineare Suche**
  - **Binäre Suche**
- Effizienz der Suche hängt von der Datenstruktur und Sortierung ab.

Lineare Suche

- **Funktionsweise**:
  - Durchläuft das Array sequenziell von Anfang bis Ende.
  - Vergleicht jedes Element mit dem gesuchten Wert.
  - Stoppt, wenn das Element gefunden wurde oder das Ende erreicht ist.
- **Eigenschaften**:
  - Laufzeitkomplexität: O(n)
  - Einfach zu implementieren.
  - Funktioniert auf unsortierten und sortierten Arrays.

Beispiel: Lineare Suche in Java

```
public static int linearSearch(int[] array, int searched) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == searched) {
            return i;
        }
    }
    return -1; // Element nicht gefunden
}
```

Binäre Suche

- **Voraussetzung**: Das Array muss **sortiert** sein.
- **Funktionsweise**:
  - Beginnt mit dem mittleren Element des Arrays.
  - Vergleicht den gesuchten Wert mit dem mittleren Element.
  - Halbiert den Suchbereich nach jedem Vergleich.
- **Eigenschaften**:
  - Laufzeitkomplexität: $O(\log n)$
  - Deutlich effizienter als die lineare Suche bei großen Datensätzen.

PDF Foliensatz (auch auf Webseite)

Beispiel: Binäre Suche in Java

```java
public static int binarySearch(int[] array, int searched) {
    int begin = 0;
    int end = array.length - 1;

    while (begin <= end) {
        int middle = (begin + end) / 2;

        ...

    }

    return -1; // Element nicht gefunden
}
```

Übung

Visualisierung der Binären Suche

Eine Visualisierung

Laufzeitkomplexität

- **Laufzeitkomplexität** misst, wie sich die Laufzeit eines Algorithmus mit der Größe der Eingabe verändert.
- Wird häufig mit der **Groß-O-Notation** dargestellt.
- Hilft beim Vergleich von Algorithmen hinsichtlich ihrer Effizienz.

Groß-O-Notation

- **O(1)**: Konstante Zeit, unabhängig von der Eingabemenge.
- **O(n)**: Lineare Zeit, proportional zur Größe der Eingabe.
- **O(log n)**: Logarithmische Zeit, wächst langsam mit der Eingabegröße.
- **O(n^2)**: Quadratische Zeit, Laufzeit steigt quadratisch mit der Eingabegröße.

Vergleich der Laufzeiten

Algorithmus	Laufzeitkomplexität
Lineare Suche	$O(n)$
Binäre Suche	$O(\log n)$
Selection Sort	$O(n^2)$
Quick Sort	$O(n \log n)$

- **Beobachtung**: Algorithmen mit niedrigerer Komplexität sind bei großen Datenmengen (Größe der Eingabedaten) effizienter.

Graphentheorie und Graphalgorithmen

- **Graphentheorie**: Studium von Graphen, die aus Knoten (Vertices) und Kanten (Edges) bestehen.
- **Anwendungen**:
  - Netzwerkoptimierung
  - Soziale Netzwerke
  - Routenplanung

Kürzeste-Wege-Algorithmus: Dijkstra (hier: single source)

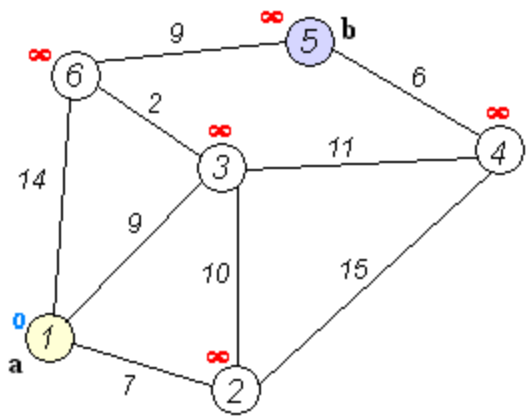
- **Ziel**: Finden des kürzesten Pfades von einem Startknoten zu allen anderen Knoten in einem gewichteten Graphen.
- **Funktionsweise**:
  - Beginnt beim Startknoten.
  - Aktualisiert die kürzesten Distanzen zu benachbarten Knoten.
  - Wählt den Knoten mit der geringsten vorläufigen Distanz als nächsten aus.
- **Anwendung**: Navigationssysteme, Netzwerk-Routing.

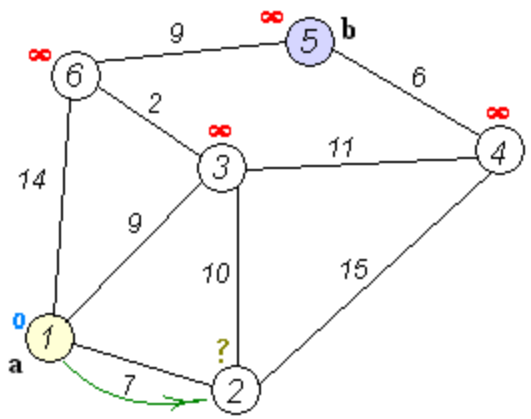
Pseudocode: Dijkstras Algorithmus

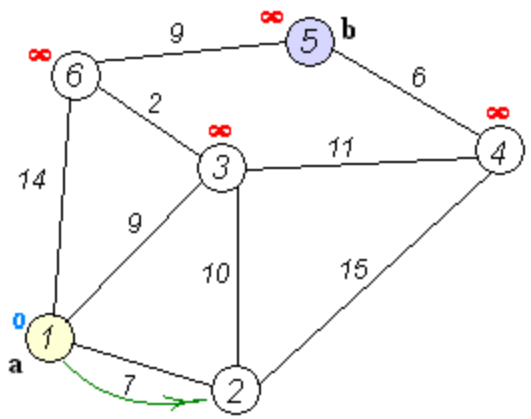
- Verwendet "PriorityQueue" (Prioritätswarteschlange). **Wichtige Datenstruktur** (aber fortgeschritteneres Thema, daher ...).

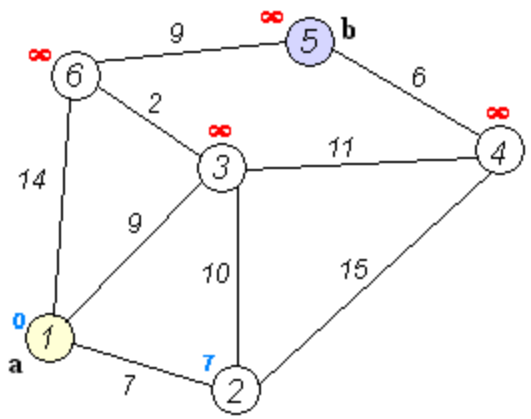
```
function Dijkstra(graph, startNode):
    create distance map, set all distances to infinity
    distance[startNode] = 0
    create priority queue Q
    add startNode to Q

    while Q is not empty:
        currentNode = node in Q with smallest distance
        remove currentNode from Q

        for each neighbor of currentNode:
            alt = distance[currentNode] + edge_weight(currentNode, neighbor)
            if alt < distance[neighbor]:
                distance[neighbor] = alt
                add neighbor to Q
```

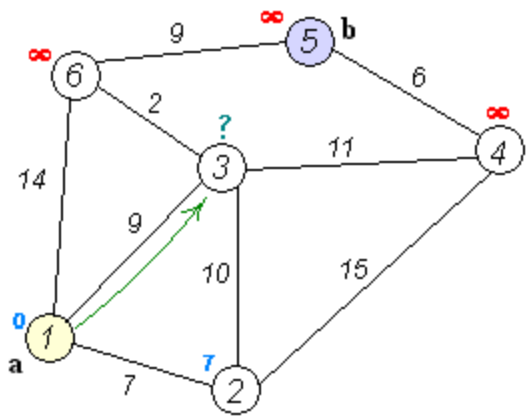
Beispiel von wikipedia (Startknoten: 1)

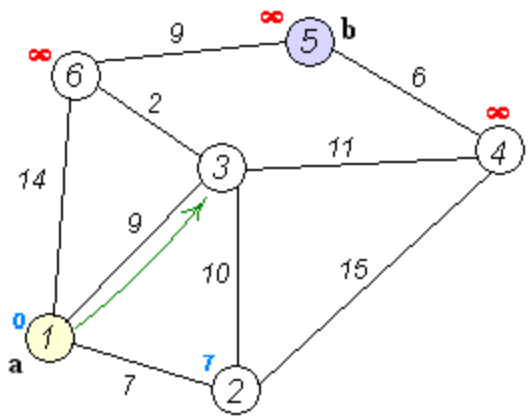

9
∞
5
b
∞
6
6
2
∞
3
11
∞
4
14
9
10
15
0
1
a
∞
2
7

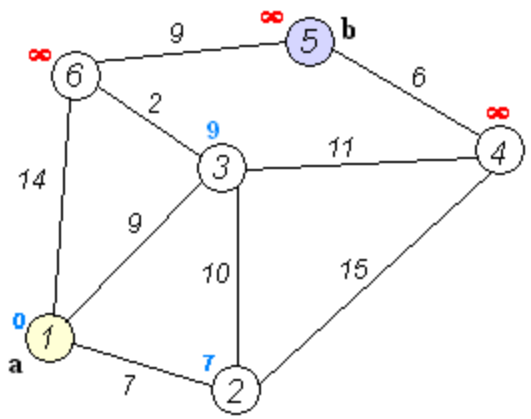

∞
9
∞
5
b
∞
6
6
2
∞
3
11
∞
4
14
9
10
15
0
1
?
a
7
2

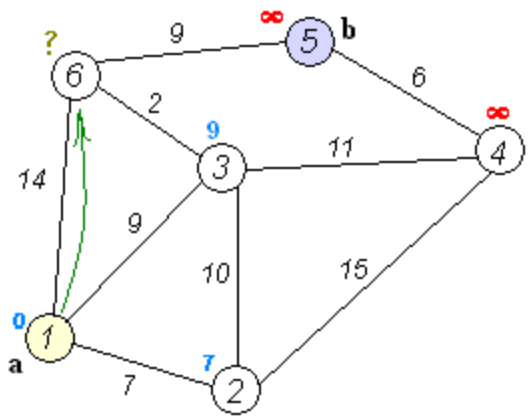

∞
9
∞
5
b
∞
6
2
6
∞
3
11
4
14
9
10
15
0
1
a
7
2

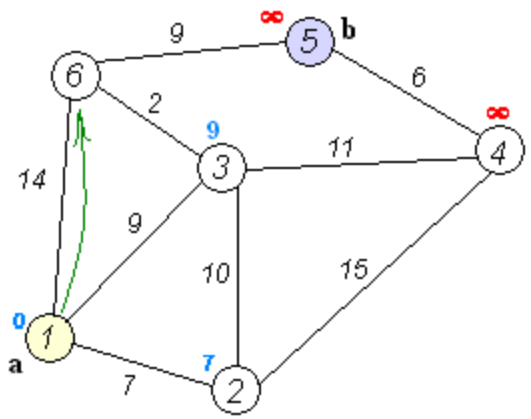

9
∞
5
b
∞
6
6
2
∞
3
11
∞
4
14
9
10
15
0
1
a
7
7
2

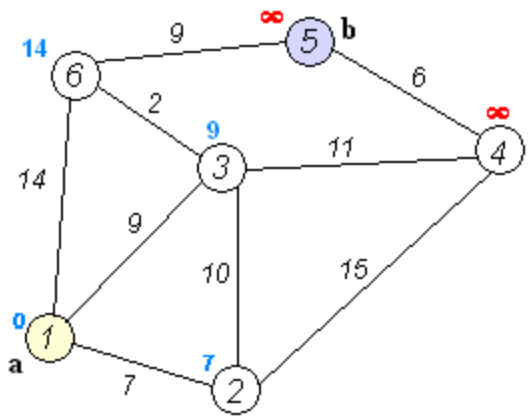

∞
9
∞
5
b
6
∞
6
2
?
3
11
4
14
9
10
15
0
1
a
7
7
2

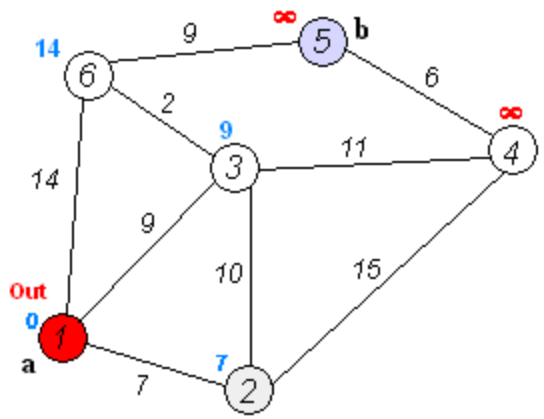

∞
∞
b
9
6
2
∞
11
14
9
10
15
0
a
7
7
1
2
3
4
5
6

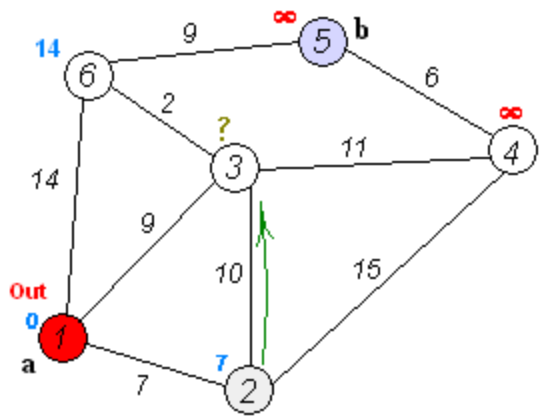

∞
9
∞
5
b
6
∞
6
2
9
3
11
4
14
9
10
15
0
1
a
7
7
2

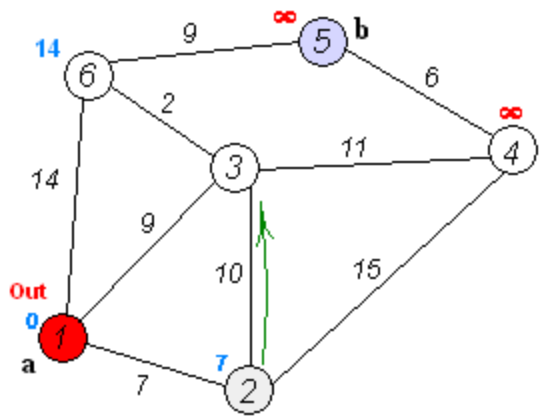

?
9
∞
5
b
6
2
6
9
∞
11
4
3
14
9
10
15
0
1
a
7
7
2

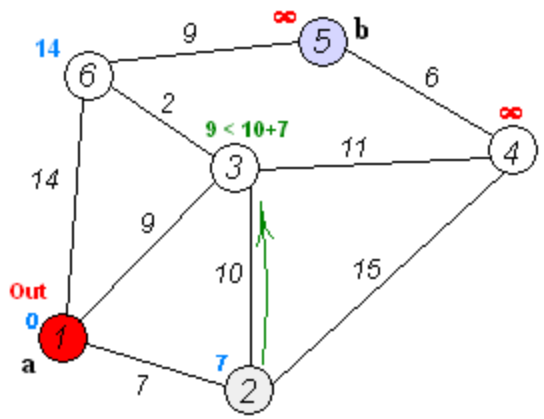

9
∞
5
b
6
6
2
9
∞
11
4
3
14
9
10
15
0
1
a
7
7
2

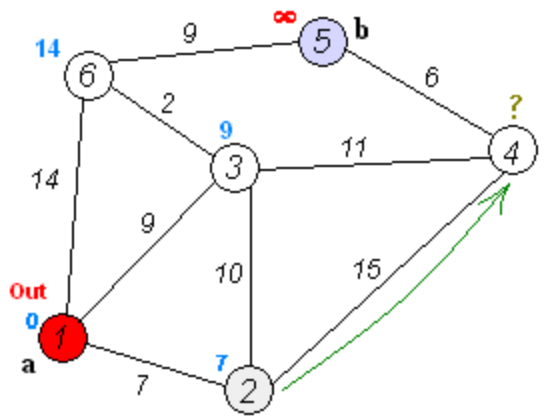

∞
b
9
5
14
6
6
2
∞
9
4
3
11
14
9
10
15
0
1
a
7
7
2

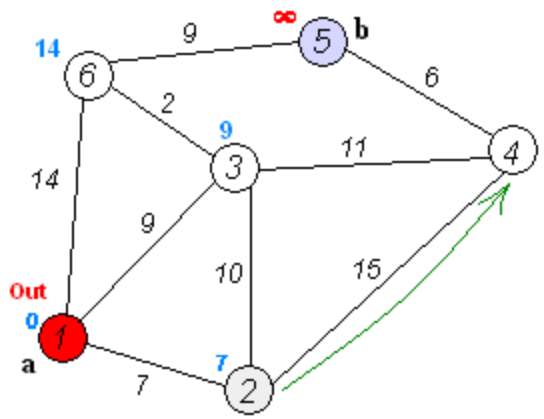

∞
b
9
14
6
5
6
2
∞
9
4
3
11
14
9
10
15
Out
0
1
a
7
7
2

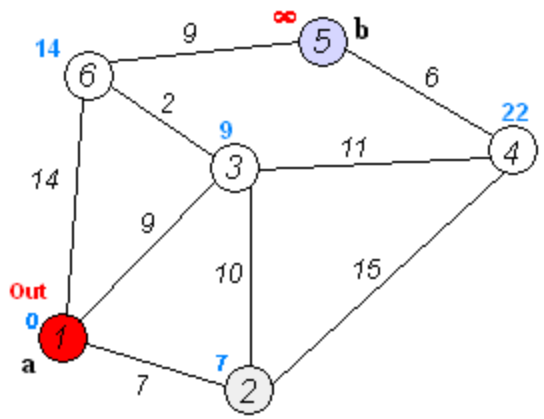

14
9
∞
5
b
6
6
2
?
∞
3
11
4
14
9
10
15
Out
0
1
a
7
7
2

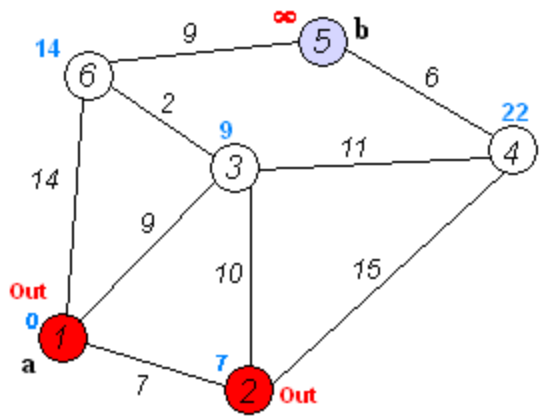

14
9
∞
5
b
6
6
2
∞
11
4
3
14
9
10
15
Out
0
1
a
7
7
2

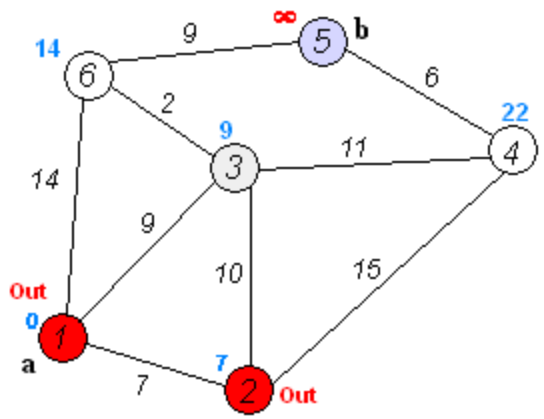

14
9
∞
5
b
6
6
2
∞
9 < 10+7
11
4
3
14
9
10
15
Out
0
1
a
7
7
2

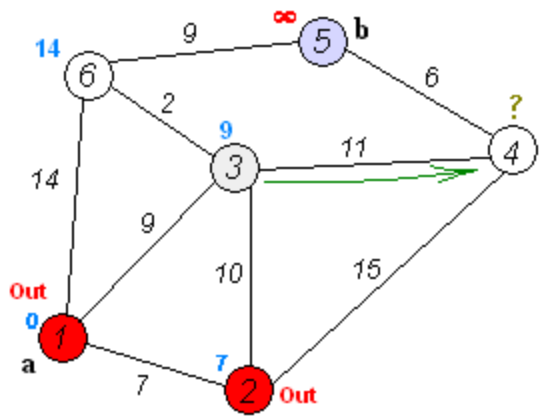

∞
b
9
14
6
5
6
2
?
9
3
11
4
14
9
10
15
Out
0
1
a
7
7
2

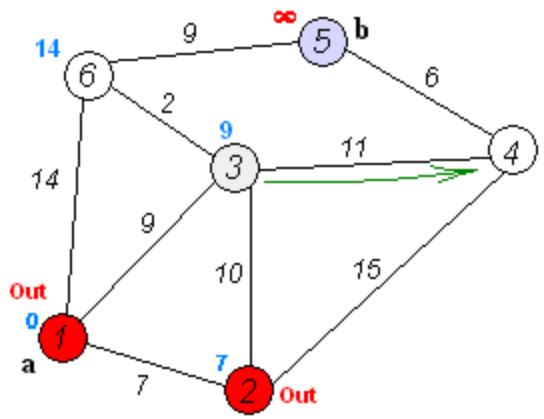

∞
b
14
9
5
6
6
2
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2

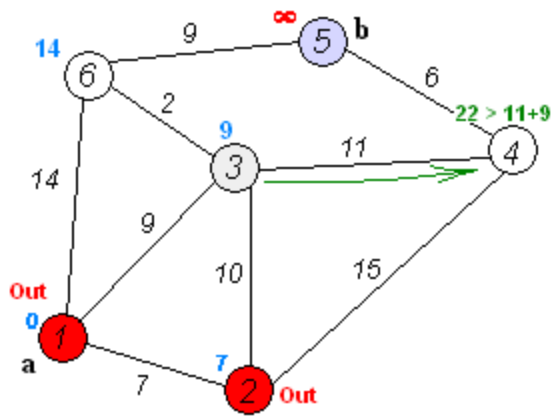

∞
b
14
9
5
6
6
2
22
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2

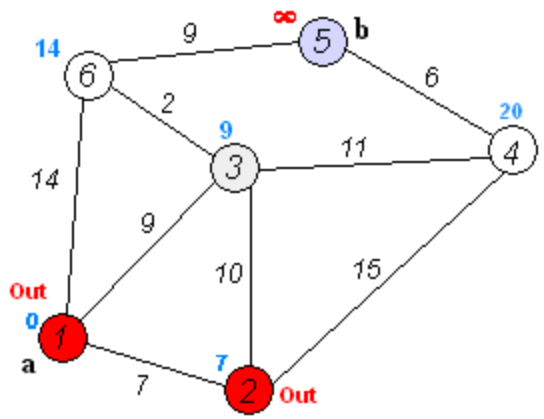

∞
b
14
9
5
6
6
2
22
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

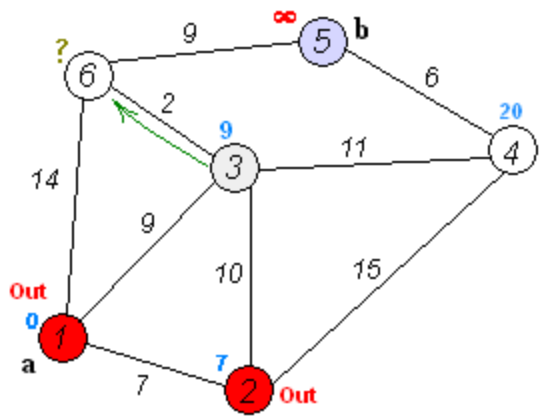

∞
b
9
14
5
6
6
2
22
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

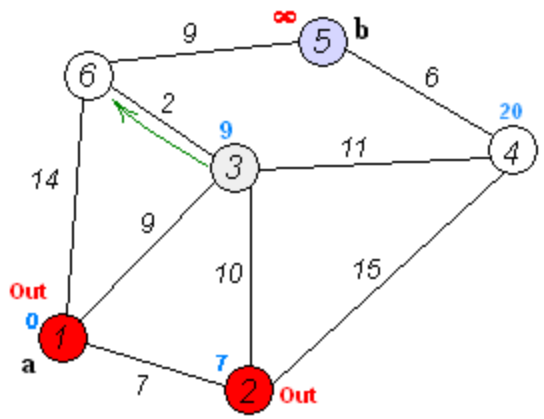

∞
b
14
9
5
6
6
2
?
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

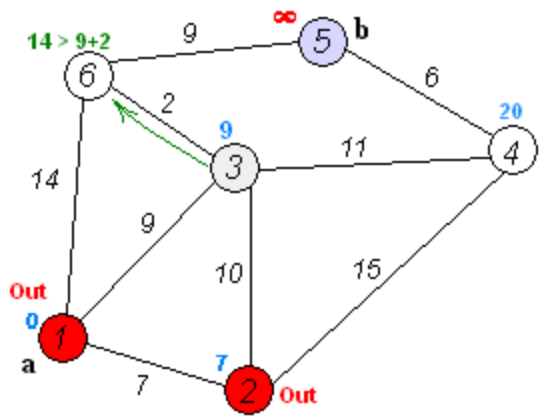

∞
b
14
9
5
6
6
2
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

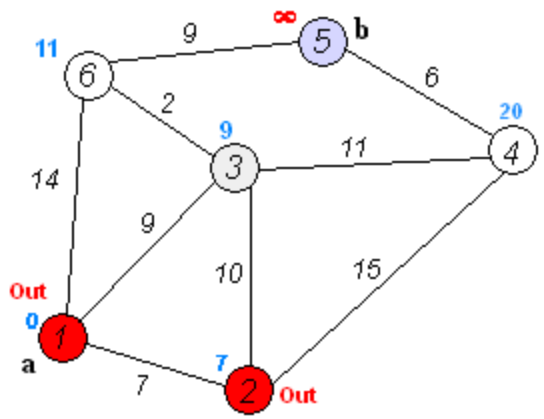

∞
b
9
14
6
5
6
2
22 > 11+9
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

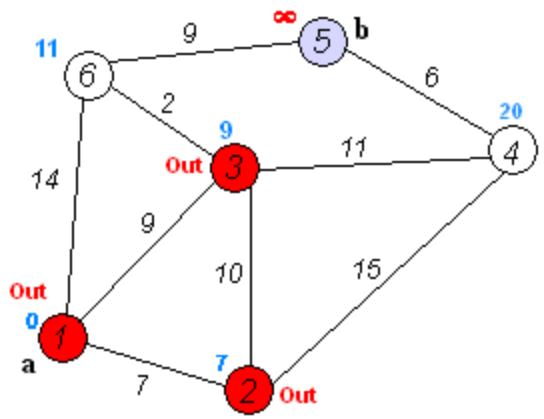

∞
b
9
14
5
6
6
2
20
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

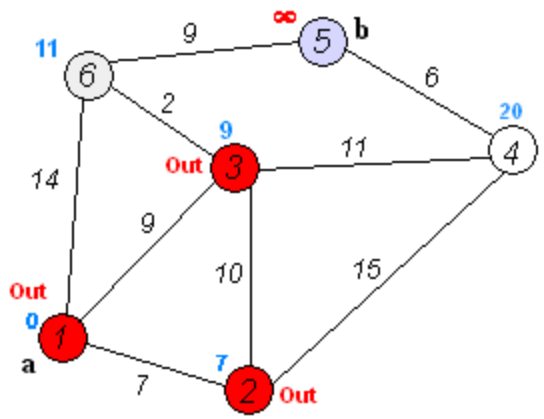

∞
b
?
9
5
6
6
2
9
20
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

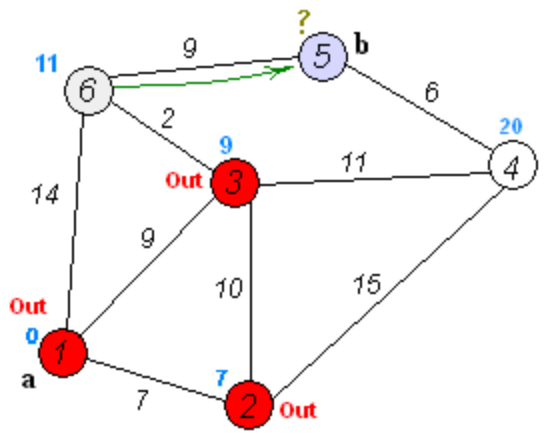

∞
9
5
b
6
6
2
9
20
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

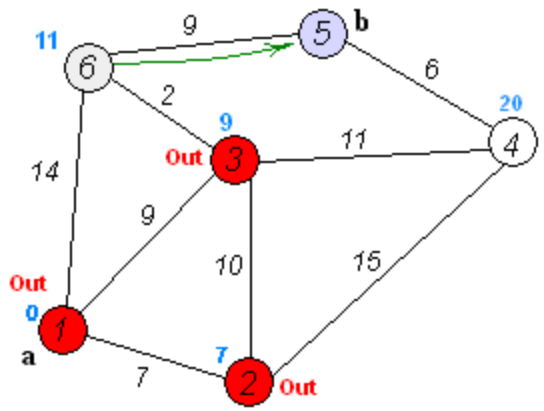

14 > 9+2
9
∞
5
b
6
6
2
9
20
3
11
4
14
9
10
15
Out
0
1
a
7
7
2
Out

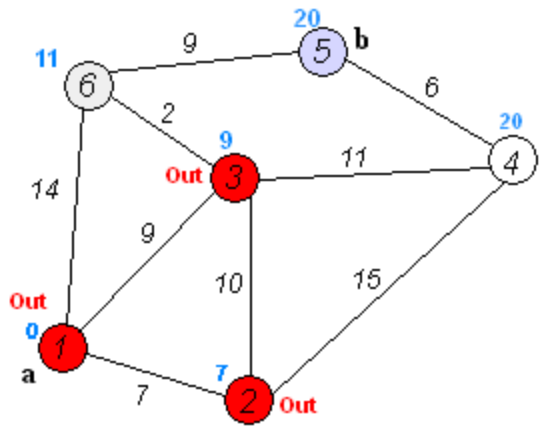

∞
b
11
9
5
6
6
2
20
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

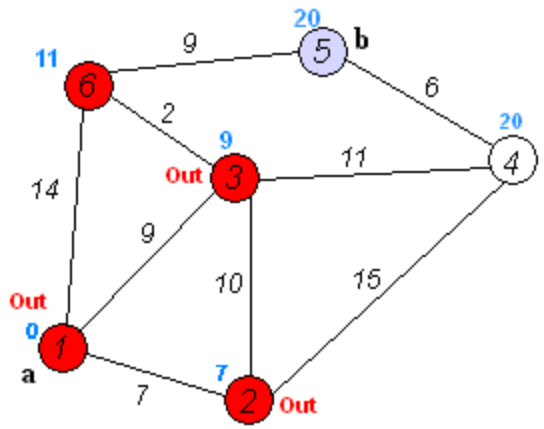

∞
b
11
9
5
6
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
2
7
Out

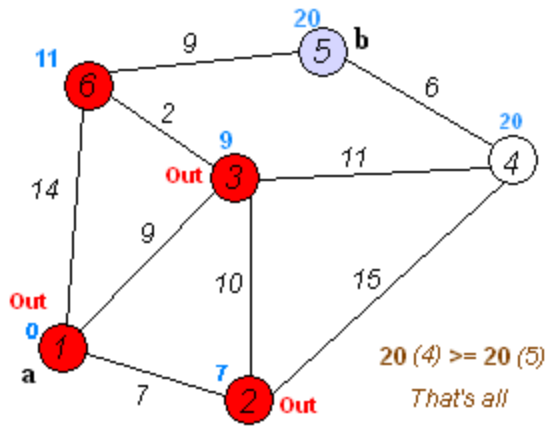

∞
b
11
9
5
6
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
2
7
Out


?
11
9
5
b
6
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
2
Out


11
9
5
b
6
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
2
Out
7


20
5
b
11
9
6
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
2
Out
7


20
5
b
11
9
6
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
2
Out
7


20
5
b
11
6
9
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
2
Out
20 (4) >= 20 (5)
That's all

Pseudocode: Dijkstras Algorithmus

- Verwendet "PriorityQueue" (Prioritätswarteschlange). **Wichtige Datenstruktur** (aber fortgeschritteneres Thema, daher ...).

```
function Dijkstra(graph, startNode):
    create distance map, set all distances to infinity
    distance[startNode] = 0
    create priority queue Q
    add startNode to Q

    while Q is not empty:
        currentNode = node in Q with smallest distance
        remove currentNode from Q

        for each neighbor of currentNode:
            alt = distance[currentNode] + edge_weight(currentNode, neighbor)
            if alt < distance[neighbor]:
                distance[neighbor] = alt
                add neighbor to Q
```

Dijkstra's Algorithmus, einfacher (single source shortest path)

Einleitung

- **Ziel**: Finden des kürzesten Pfades von einem Startknoten zu allen anderen Knoten in einem gewichteten Graphen ohne negative Kantengewichte.
- **Anwendungen**:
  - Routenplanung und Navigation
  - Netzwerkoptimierung
  - Verkehrsflusssteuerung

Grundprinzip des Algorithmus

1. **Initialisierung**:
    - Setze die Distanz zum Startknoten auf 0.
    - Setze die Distanzen zu allen anderen Knoten auf unendlich.
    - Markiere alle Knoten als unbesucht.
2. **Algorithmusschritte**:
    - **Schritt 1**: Wähle den unbesuchten Knoten mit der kleinsten aktuellen Distanz.
    - **Schritt 2**: Für alle Nachbarn des ausgewählten Knotens:
        - Berechne die alternative Distanz.
        - Aktualisiere die Distanz, wenn der neue Weg kürzer ist.
    - **Schritt 3**: Markiere den ausgewählten Knoten als besucht.
    - **Schritt 4**: Wiederhole die Schritte 1–3, bis alle Knoten besucht sind.

Java-Implementierung: Hauptmethode

```java
public static int[] dijkstra(int[][] graph, int startNode) {
    int n = graph.length;
    int[] dist = new int[n];
    boolean[] visited = new boolean[n];
    int[] predecessor = new int[n];

    // Initialisierung
    for (int i = 0; i < n; i++) {
        dist[i] = Integer.MAX_VALUE;
        visited[i] = false;
        predecessor[i] = -1;
    }
    dist[startNode] = 0;

    // Hauptschleife
    for (int i = 0; i < n - 1; i++) {
        int u = selectMinDistanceNode(dist, visited);
        visited[u] = true;
```

Hilfsmethode: Knoten mit minimaler Distanz auswählen

```java
public static int selectMinDistanceNode(int[] dist, boolean[] visited) {
    int min = Integer.MAX_VALUE;
    int minIndex = -1;

    for (int v = 0; v < dist.length; v++) {
        if (!visited[v] && dist[v] <= min) {
            min = dist[v];
            minIndex = v;
        }
    }
    return minIndex;
}
```

- **Zweck**: Findet den unbesuchten Knoten mit der kleinsten aktuellen Distanz.

Pfadrekonstruktion

```
public static List<Integer> getShortestPath(int[] predecessor, int targetNode) {
    List<Integer> path = new ArrayList<>();
    int currentNode = targetNode;

    while (currentNode != -1) {
        path.add(currentNode);
        currentNode = predecessor[currentNode];
    }
    Collections.reverse(path);
    return path;
}
```

- **Zweck**: Rekonstruiert den kürzesten Pfad vom Startknoten zum Zielknoten.

Beispiel: Graph erstellen (ZERO-indexed!!)

```
int[][] graph = {
   //  0  1  2  3  4  5
     {  0, 7, 9, 0, 0,14}, // 0
     {  7, 0,10,15, 0, 0}, // 1
     {  9,10, 0,11, 0, 2}, // 2
     {  0,15,11, 0, 6, 0}, // 3
     {  0, 0, 0, 6, 0, 9}, // 4
     { 14, 0, 2, 0, 9, 0}  // 5
};
```

- **Graphdetails**:
  - Knoten von 0 bis 5
  - `graph[u][v]` gibt das Gewicht der Kante von `u` nach `v` an.
  - Ein Wert von 0 bedeutet, dass keine direkte Kante existiert.

Dijkstra-Algorithmus ausführen

```
int startNode = 0;
int[] predecessor = dijkstra(graph, startNode);

// Kürzeste Wege zu allen Knoten ausgeben
for (int targetNode = 0; targetNode < graph.length; targetNode++) {
    List<Integer> path = getShortestPath(predecessor, targetNode);
    System.out.println("Kürzester Weg von " + startNode + " zu " + targetNode +
}
```

- **Startknoten**: 0
- **Ausgabe**: Kürzeste Wege vom Startknoten zu allen anderen Knoten.
- Nicht sehr clever. Warum?

Dijkstra's Algorithmus: Schritt-für-Schritt-Durchlauf

Ausgangssituation

- **Graph** (Adjazenzmatrix):

	0	**1**	**2**	**3**	**4**	**5**
0	0	7	9	0	0	14
1	7	0	10	15	0	0
2	9	10	0	11	0	2
3	0	15	11	0	6	0
4	0	0	0	6	0	9
5	14	0	2	0	9	0

- **Startknoten**: `0`
- **Initialisierung**:
  - `dist[] = [0, ∞, ∞, ∞, ∞, ∞]`
  - `visited[] = [F, F, F, F, F, F]`

Iteration 1

- **Aktueller Zustand:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	∞	∞	∞	∞	∞
visited[]	**F**	F	F	F	F	F

- **Gewählter Knoten**: `0` (dist = 0)
- **Aktualisierung der Nachbarn von Knoten 0:**
  - Knoten 1: `dist[1] = min(∞, 0 + 7) = 7`
  - Knoten 2: `dist[2] = min(∞, 0 + 9) = 9`
  - Knoten 5: `dist[5] = min(∞, 0 + 14) = 14`
- **Markiere Knoten 0 als besucht**:
  - `visited[0] = T`

Iteration 1 (Fortsetzung)

- **Aktualisierter Zustand nach Iteration 1:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	**7**	**9**	∞	∞	**14**
visited[]	**T**	F	F	F	F	F

Iteration 2

- **Aktueller Zustand:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	7	9	∞	∞	14
visited[]	T	**F**	F	F	F	F

- **Gewählter Knoten**: `1` (dist = 7)
- **Aktualisierung der Nachbarn von Knoten 1:**
  - Knoten 2: `dist[2]` bleibt `9` (kein kürzerer Weg)
  - Knoten 3: `dist[3] = min(∞, 7 + 15) = 22`
- **Markiere Knoten 1 als besucht**:
  - `visited[1] = T`

Iteration 2 (Fortsetzung)

- **Aktualisierter Zustand nach Iteration 2:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	7	9	**22**	∞	14
visited[]	T	**T**	F	F	F	F

Iteration 3

- **Aktueller Zustand:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	7	9	22	∞	14
visited[]	T	T	**F**	F	F	F

- **Gewählter Knoten**: `2` (dist = 9)
- **Aktualisierung der Nachbarn von Knoten 2:**
  - Knoten 3: `dist[3] = min(22, 9 + 11) = 20`
  - Knoten 5: `dist[5] = min(14, 9 + 2) = 11`
- **Markiere Knoten 2 als besucht**:
  - `visited[2] = T`

Iteration 3 (Fortsetzung)

- **Aktualisierter Zustand nach Iteration 3:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	7	9	**20**	∞	**11**
visited[]	T	T	**T**	F	F	F

Iteration 4

- **Aktueller Zustand:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	7	9	20	∞	11
visited[]	T	T	T	F	F	**F**

- **Gewählter Knoten**: `5` (dist = 11)
- **Aktualisierung der Nachbarn von Knoten 5:**
  - Knoten 4: `dist[4] = min(∞, 11 + 9) = 20`
- **Markiere Knoten 5 als besucht**:
  - `visited[5] = T`

Iteration 4 (Fortsetzung)

- **Aktualisierter Zustand nach Iteration 4:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	7	9	20	**20**	11
visited[]	T	T	T	F	F	**T**

Iteration 5

- **Aktueller Zustand:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	7	9	20	20	11
visited[]	T	T	T	**F**	F	T

- **Gewählter Knoten**: `3` (dist = 20)
- **Aktualisierung der Nachbarn von Knoten 3:**
  - Knoten 4: `dist[4] = min(20, 20 + 6) = 20` (bleibt unverändert)
- **Markiere Knoten 3 als besucht**:
  - `visited[3] = T`

Iteration 5 (Fortsetzung)

- **Aktualisierter Zustand nach Iteration 5:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	7	9	20	20	11
visited[]	T	T	T	**T**	F	T

Iteration 6

- **Aktueller Zustand:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	7	9	20	20	11
visited[]	T	T	T	T	**F**	T

- **Gewählter Knoten**: `4` (dist = 20)
- **Aktualisierung der Nachbarn von Knoten 4:**
  - Keine unbesuchten Nachbarn mit kürzerem Weg
- **Markiere Knoten 4 als besucht**:
  - `visited[4] = T`

Iteration 6 (Fortsetzung)

- **Aktualisierter Zustand nach Iteration 6:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	7	9	20	20	11
visited[]	T	T	T	T	**T**	T

Endergebnis

- **Finale dist[] und visited[] Arrays:**

Knoten	**0**	**1**	**2**	**3**	**4**	**5**
dist[]	0	7	9	20	20	11
visited[]	T	T	T	T	T	T

- **Vorgänger (predecessor[])**:

Knoten	**Vorgänger**
0 (Start)	-1
1	0
2	0
3	2
4	5
5	2

Kürzeste Pfade vom Startknoten 0

- **Pfad zu Knoten 0**: `[0]` (Distanz: 0)
- **Pfad zu Knoten 1**: `[0, 1]` (Distanz: 7)
- **Pfad zu Knoten 2**: `[0, 2]` (Distanz: 9)
- **Pfad zu Knoten 3**: `[0, 2, 3]` (Distanz: 20)
- **Pfad zu Knoten 4**: `[0, 2, 5, 4]` (Distanz: 20)
- **Pfad zu Knoten 5**: `[0, 2, 5]` (Distanz: 11)

Programmausgabe

```
Kürzester Weg von 0 zu 0: [0]
Kürzester Weg von 0 zu 1: [0, 1]
Kürzester Weg von 0 zu 2: [0, 2]
Kürzester Weg von 0 zu 3: [0, 2, 3]
Kürzester Weg von 0 zu 4: [0, 2, 3, 4]
Kürzester Weg von 0 zu 5: [0, 2, 5]
```

- **Interpretation**:
  - Zeigt die kürzesten Pfade und die Reihenfolge der Knoten vom Startknoten zu jedem anderen Knoten.

Objektorientierte Implementierung

Prinzipien der Modularität und Kapselung

Ziel

- Veranschaulichung des Dijkstra-Algorithmus in objektorientierter Struktur

Warum Objektorientierung?

- **Modularität**: Trennung der Verantwortlichkeiten (Graph, Algorithmus)
- **Kapselung**: Daten und Funktionen in Klassen gebündelt
- **Lesbarkeit**: Klare Struktur erleichtert Wartung und Erweiterung

Komponenten

- `Graph` : Repräsentiert den Graphen (Adjazenzmatrix)
- `DijkstraAlgorithm` : Implementiert den Algorithmus
- `DijkstraDemo` : Führt das Programm aus

Die Klasse `Graph`

```
class Graph {
    private final int[][] adjacencyMatrix;
    private final int numberOfNodes;

    public Graph(int[][] adjacencyMatrix) {
        this.adjacencyMatrix = adjacencyMatrix;
        this.numberOfNodes = adjacencyMatrix.length;
    }

    public int getNumberOfNodes() {
        return numberOfNodes;
    }

    public int getEdgeWeight(int from, int to) {
        return adjacencyMatrix[from][to];
    }

    public boolean hasEdge(int from, int to) {
        return adjacencyMatrix[from][to] != 0;
    }
}
```

- Kapselt die Adjazenzmatrix
- Bietet einfache Schnittstellen für:
  - Anzahl der Knoten
  - Abfrage von Kanten und deren Gewicht

Die Klasse `DijkstraAlgorithm`

Übersicht

```java
class DijkstraAlgorithm {
    private final Graph graph;
    private final int[] distances;
    private final boolean[] visited;
    private final int[] predecessors;

    public DijkstraAlgorithm(Graph graph) {
        this.graph = graph;
        int n = graph.getNumberOfNodes();
        this.distances = new int[n];
        this.visited = new boolean[n];
        this.predecessors = new int[n];
    }
    //...
```

- Kapselt die Algorithmuslogik
- Verwendet `Graph`-Objekt, um auf Knoten und Kanten zuzugreifen

Die Methode `findShortestPaths`

Implementierung des Dijkstra-Algorithmus

```
public int[] findShortestPaths(int startNode) {
    initialize(startNode);

    for (int i = 0; i < graph.getNumberOfNodes() - 1; i++) {
        int currentNode = selectMinDistanceNode();
        visited[currentNode] = true;

        updateDistances(currentNode);
    }

    return predecessors;
}
```

- **Schritte:**
    i. Initialisiere Distanzen und Vorgänger.
    ii. Wähle den Knoten mit der geringsten Distanz.
    iii. Aktualisiere Distanzen zu seinen Nachbarn.

Hilfsmethoden der `DijkstraAlgorithm`

Initialisierung

```java
private void initialize(int startNode) {
    Arrays.fill(distances, Integer.MAX_VALUE);
    Arrays.fill(visited, false);
    Arrays.fill(predecessors, -1);
    distances[startNode] = 0;
}
```

Auswahl des nächsten Knotens

```java
private int selectMinDistanceNode() {
    int minDistance = Integer.MAX_VALUE;
    int minIndex = -1;

    for (int i = 0; i < distances.length; i++) {
        if (!visited[i] && distances[i] <= minDistance) {
            minDistance = distances[i];
            minIndex = i;
        }
    }

    return minIndex;
}
```

Aktualisierung der Distanzen

```
private void updateDistances(int currentNode) {
    for (int neighbor = 0; neighbor < graph.getNumberOfNodes(); neighbor++) {
        if (!visited[neighbor] && graph.hasEdge(currentNode, neighbor)) {
            int newDistance = distances[currentNode] + graph.getEdgeWeight(currentNode, neighbor)
            if (newDistance < distances[neighbor]) {
                distances[neighbor] = newDistance;
                predecessors[neighbor] = currentNode;
            }
        }
    }
}
```

Hauptprogramm: `DijkstraDemo`

Verwendung der Klassen

```java
public class DijkstraDemo {
    public static void main(String[] args) {
        int[][] adjacencyMatrix = {
            {0, 10, 0, 0, 0, 0},
            {10, 0, 5, 0, 0, 0},
            {0, 5, 0, 20, 1, 0},
            {0, 0, 20, 0, 0, 2},
            {0, 0, 1, 0, 0, 3},
            {0, 0, 0, 2, 3, 0}
        };

        Graph graph = new Graph(adjacencyMatrix);
        DijkstraAlgorithm dijkstra = new DijkstraAlgorithm(graph);

        int startNode = 0;
        int[] predecessors = dijkstra.findShortestPaths(startNode);

        System.out.println("Shortest distances: " + Arrays.toString(dijkstra.getDistances()));
        System.out.println("Predecessors: " + Arrays.toString(predecessors));
    }
}
```

Zusammenfassung

- **Vorteile der OO-Implementierung:**
  - **Modularität**: Graph und Algorithmus klar getrennt.
  - **Kapselung**: Details der Adjazenzmatrix und Logik versteckt.
  - **Lesbarkeit**: Leicht erweiterbar und wartbar.
- **Nächste Schritte: (Beispiele)**
  - Erweiterung für ungerichtete Graphen.
  - Verwendung einer PriorityQueue.
  - Unterstützung für größere Graphen.

Wichtige Aspekte beider Implementierungen

- Hier: **Keine PriorityQueue verwendet**:
  - Auswahl des nächsten Knotens erfolgt durch lineare Suche über alle Knoten.
  - Zeitkomplexität: $O(|V|^2)$, wobei V die Menge der Knoten ist.
- **Adjazenzmatrix**:
  - `int[][] graph` erleichtert die Darstellung von Graphen in kleiner Größe.
  - Nicht effizient für Graphen mit vielen Knoten oder wenigen Kanten (spärliche Graphen).

Zusammenfassung

- **Dijkstra's Algorithmus**:
  - Effiziente Methode zur Berechnung der kürzesten Wege in einem gewichteten Graphen ohne negative Kantengewichte.
- **Implementierung ohne PriorityQueue**:
  - Einfacher zu verstehen und zu implementieren.
  - Geeignet für kleinere Graphen.
- **Verwendung einer Adjazenzmatrix**:
  - Intuitive Darstellung von Graphen für Bildungszwecke.
- Fokus auf **Einfachheit** und **Klarheit**, nicht auf Effizienz

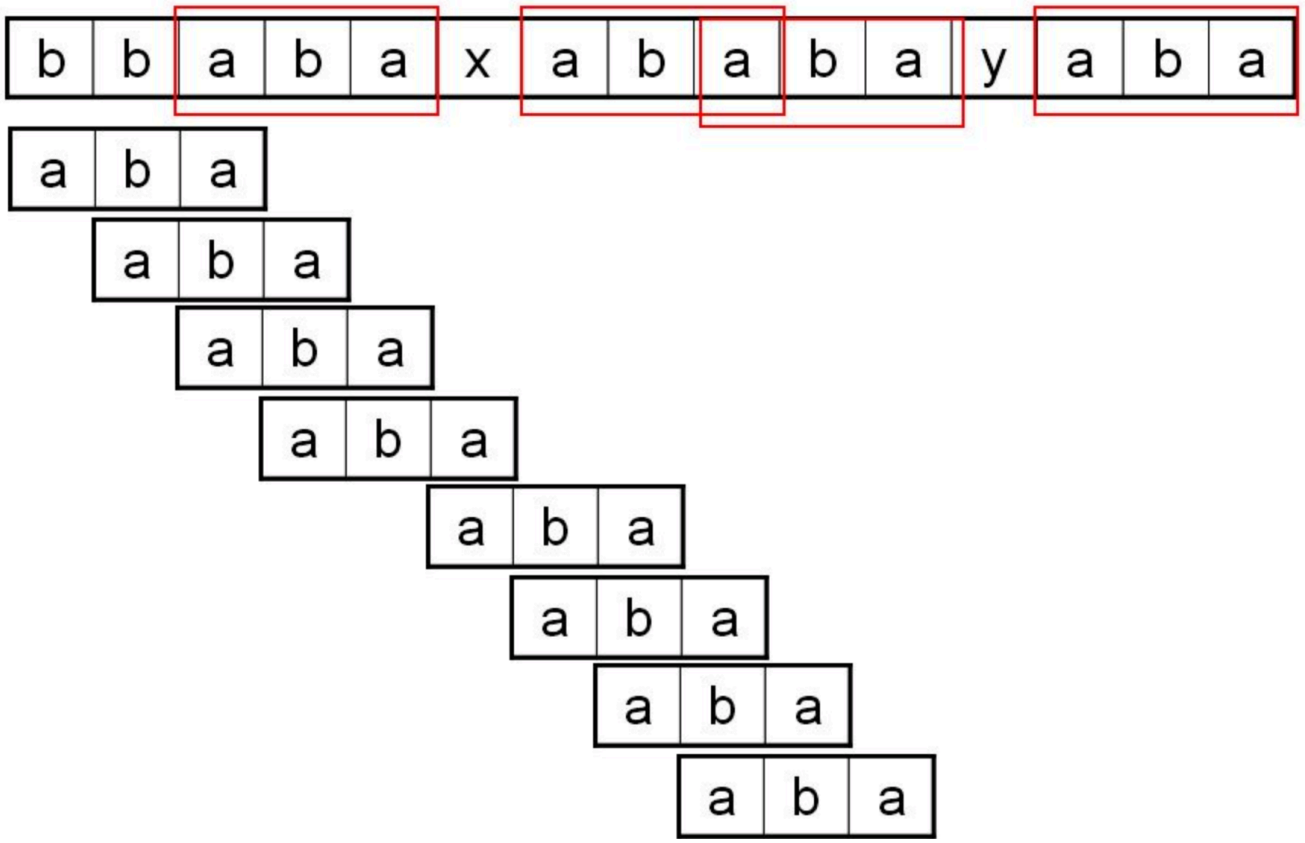
Grenzen dieser Implementierung

- **Effizienz**:
  - Nicht geeignet für große Graphen aufgrund der $O(|V|^2)$-Zeitkomplexität.
- **Speicherverbrauch**:
  - Adjazenzmatrix benötigt $O(|V|^2)$ Speicherplatz.
- **Flexibilität**:
  - Schwieriger anzupassen für Graphen mit dynamischer Größe oder für spärliche Graphen.

Mustersuche (Pattern Search)

- **Ziel**: Finden eines bestimmten Musters in einem Text oder einer Datenmenge.
- **Anwendungen**:
  - Textverarbeitung (z.B. Suchen von Wörtern in Dokumenten)
  - Bioinformatik (z.B. DNA-Sequenzanalyse)
- **Algorithmen**:
  - Naiver Suchalgorithmus
  - Knuth-Morris-Pratt (KMP)
  - Boyer-Moore

Naïve Approach



O(n + m)

Knuth-Morris-Pratt (KMP) Algorithmus

- **Vorteil**: Effiziente Suche durch Vorverarbeitung des Musters.
- **Funktionsweise**:
  - Erstellt eine Präfix-Tabelle (Failure Function), um Übersprünge bei der Suche zu ermöglichen.
  - Vermeidet redundante Vergleiche.
- **Laufzeitkomplexität**: O(n + m), wobei n die Länge des Textes und m die Länge des Musters ist. (im Gegensatz zu O(n * m) beim naïven Ansatz.
- Nicht in diesem Kurs.
- Wichtige Datenstruktur um effiziente Lösungen zahlreicher Probleme im Bereich der Stringverarbeitung zu finden: **suffix trees**

Pseudocode: KMP-Algorithmus

```
function KMP_search(text, pattern):
    prefixTable = compute_prefix_function(pattern)
    i = 0 // index for text
    j = 0 // index for pattern

    while i < length(text):
        if pattern[j] == text[i]:
            i++
            j++
            if j == length(pattern):
                return "Pattern found at position " + (i - j)
                j = prefixTable[j - 1]
        else:
            if j != 0:
                j = prefixTable[j - 1]
            else:
                i++
    return "Pattern not found"
```

Machine Learning und Algorithmen

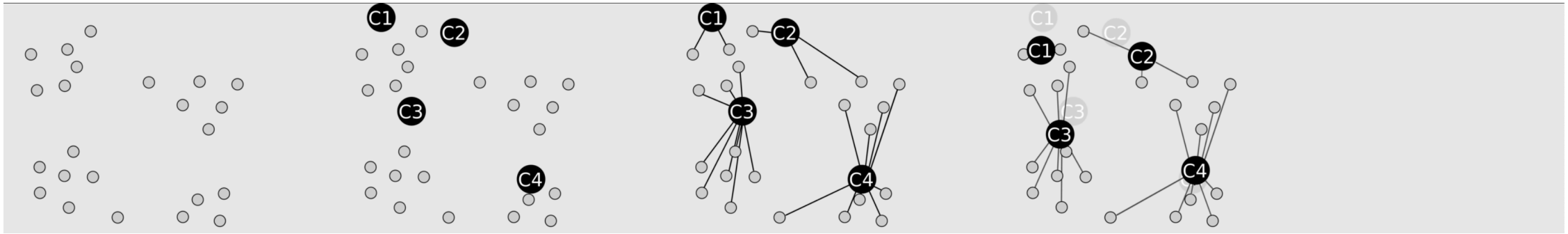
- **Machine Learning (ML)**: Teilgebiet der KI, bei dem Algorithmen aus Daten lernen.
- **Arten von ML**:
  - Überwachtes Lernen (supervised learning))
  - Unüberwachtes Lernen (unsupervised learning)
  - ...
- **Algorithmen**:
  - Entscheidungsbäume
  - Neuronale Netze
  - Clustering-Methoden

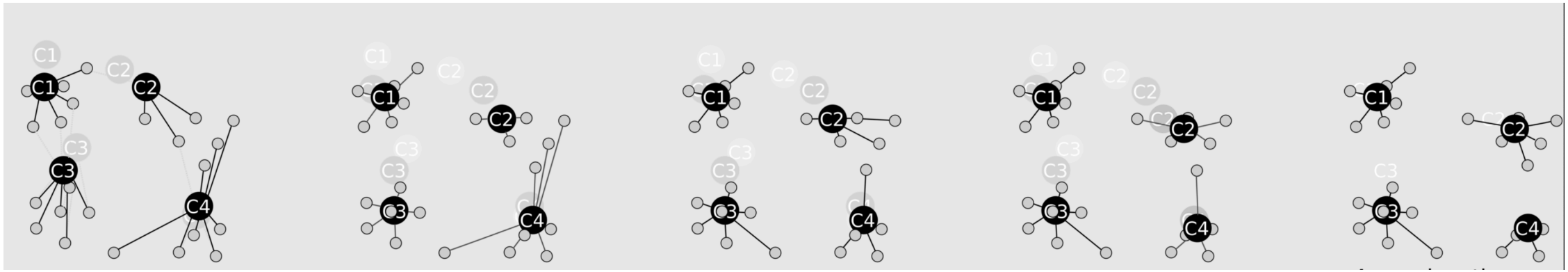
K-Means Clustering

- **Unüberwachter Lernalgorithmus** zur Gruppierung von Datenpunkten.
- **Ziel**: Partitionierung der Daten in k Cluster, wobei jeder Datenpunkt zum Cluster mit dem nächstgelegenen Mittelwert gehört.
- **Funktionsweise**:
  - Initialisierung von k Zentroiden.
  - Zuordnung der Datenpunkte zum nächstgelegenen Zentrum.
  - Aktualisierung der Zentroiden basierend auf den zugewiesenen Punkten.
  - Wiederholung bis zur Konvergenz.

Pseudocode: K-Means Clustering

```
function KMeans(data, k):
    initialize centroids randomly
    repeat:
        assign each data point to the nearest centroid
        compute new centroids as the mean of assigned points
    until centroids do not change
    return clusters
```


C1
C2
C3
C4


C1
C2
C3
C4

Anwendungen von K-Means

- **Marketing**: Kundensegmentierung.
- **Bildverarbeitung**: Farbreduktion.
- **Anomalieerkennung**: Identifikation von Ausreißern.

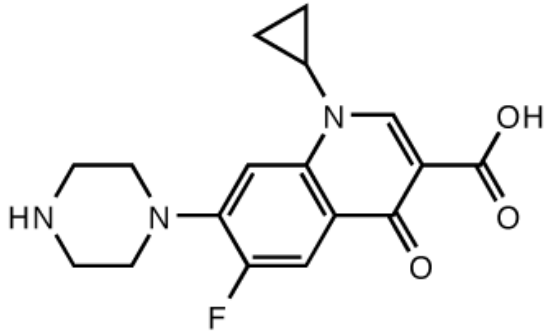
Algorithmen in der Chemieinformatik

- **Chemieinformatik** nutzt Algorithmen zur Verarbeitung, Modellierung, Analyse und Vorhersage des Verhaltens chemischer Systeme.
- **Anwendungen**:
  - Molekülstrukturanalyse
  - Vorhersage chemischer Reaktionen und Prozesse
  - Wirkstoff -suche und -design in der Pharmazie
  - Design von Mikroben
  - ...

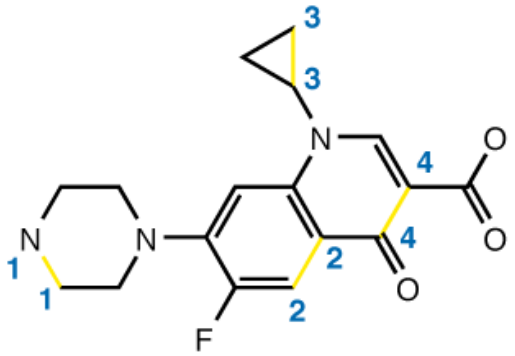
SMILES – Molekülrepräsentation

- Tafel: Was ist eine Normalform / canonical representation? (Mathematisches Model vs. Representation)
- Tafel: Normalform einer rationaler Zahl.
- **SMILES**: Simplified Molecular Input Line Entry System.
- **Ziel**: Textuelle Beschreibung (als String) von Molekülstrukturen.
- **Vorteile**:
  - Einfach zu speichern und zu finden.
  - Ermöglicht algorithmische Verarbeitung chemischer Verbindungen.

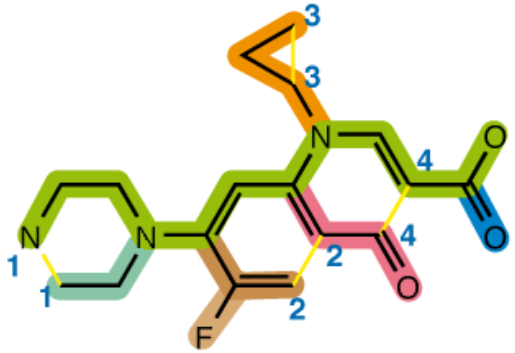
A



B



C



D

```
N1CCN(CC1)C(C(F)=C2)=CC(=C2C4=O)N(C3CC3)C=C4C(=O)O
```

- **Spannbaum**
- Wieviele Spannbäume hat ein Graph?
- Konvertierung Graph nach String. Eindeutig? NEIN!
- Verwandt: Ist eine Adjazenzmatrix eindeutig für einen Graphen? NEIN! (Beispiel: Tafel)

Schritte zur Berechnung der SMILES-Normalform

1. **Graphrepräsentation**:
    - Molekül wird als Graph dargestellt.
    - Atome sind Knoten, Bindungen sind Kanten.
2. **Atompriorität**:
    - Atome werden nach bestimmten Regeln priorisiert.
3. **Traversal**:
    - Graph wird mit einem Traversierungsalgorithmus durchlaufen (z.B. DFS).
4. **Standardisierung**:
    - Ergebnis wird in eine kanonische Form gebracht, um eindeutige Repräsentationen zu gewährleisten.

link

Laufzeitkomplexität in der Praxis

- **Effiziente Algorithmen** sind entscheidend für große Datenmengen.
- **Beispiel**:
  - Ein Algorithmus mit O(n log n) ist bei großen n deutlich schneller als einer mit $O(n^2)$.
- Wichtig: Theorie vs. Praxis

Zusammenfassung

- **Algorithmen** sind grundlegende Werkzeuge in der Informatik.
- **Effizienz** und **Komplexität** sind wichtige Kriterien bei der Auswahl von Algorithmen.
- **Sortier- und Suchalgorithmen** und die Wahl der richtigen **Datenstrukturen** bilden die Basis für viele Anwendungen.
- **Anwendungen** von Algorithmen sind de-facto endlos: sie reichen von Machine Learning, Kryptographie, Routenplanung, Bildverarbeitung, Modellierung, Simulation, Robotik, ... sie sind die Bausteine von allen effizienten Systemen.

Ende Teil 07