

Übung 4 - Java Projekt

Dokumentation

Raja Abdulhadi, Alexander R. Brenner, Julia Michler - BSWE3B

Oktober 2025

Inhaltsverzeichnis

1	Einleitung	3
2	Architektur und Aufbau	3
2.1	Schichtenmodell	3
2.2	Datenmodell	3
2.3	CSV-Verarbeitung mit OpenCSV	3
2.4	Graphenbasierte Modellierung	4
2.5	Pathfinding-Algorithmen	4
2.5.1	Dijkstra-Algorithmus - Kürzeste bzw. günstigste Route	4
2.5.2	BFS - Route mit den wenigsten Umstiegen	5
3	Benutzeroberfläche und Funktionalität	6
3.1	Konsolenmenü	6
3.2	Routenberechnung	6
3.3	Sortierfunktionalität	7
3.4	Suchfunktionalität	7
4	Sortieralgorithmen	8
4.1	MergeSort - Stabile Sortierung	8
4.2	QuickSort - Instabile Sortierung	8
4.3	Auswahl zwischen MergeSort und QuickSort	9

1 Einleitung

Im Rahmen der vierten Übung der Lehrveranstaltung *Algorithmen und Programmier-techniken* wurde ein Java-basiertes System zur Verwaltung und Berechnung von Flugrouten entwickelt.

2 Architektur und Aufbau

2.1 Schichtenmodell

Das System folgt dem Prinzip der *Separation of Concerns (SoC)* und ist in drei logische Ebenen gegliedert:

Ebene	Verantwortung	Beispielklassen
Repository	Einlesen und Schreiben von CSV-Dateien	<code>AirportRepository</code> , <code>FlightRepository</code>
Service	Bereitstellung der geladenen Daten, Koordination	<code>RoutingDataService</code>
Algorithmus/Business	Routenberechnung (Dijkstra, BFS)	<code>RoutingCalculator</code>

Diese Struktur stellt sicher, dass technische Aspekte (Dateizugriff) klar von der Geschäftslogik (Routenberechnung) getrennt bleiben. Eine spätere Migration von CSV-Dateien zu einer Datenbank wäre dadurch mit minimalem Aufwand möglich.

2.2 Datenmodell

Die Kern-Modelle spiegeln die realen Entitäten des Problems wider:

- **Airport:** Repräsentiert einen Flughafen mit IATA-Code, Stadt, Land und Koordinaten.
- **Flight:** Repräsentiert einen gerichteten Flug (Kante) zwischen zwei Flughäfen mit Attributen wie Airline, Preis, Dauer und Abflugzeit.
- **Route:** Bündelt eine Sequenz von Flügen mit Gesamtpreis, Gesamtdauer und Anzahl der Umstiege.

2.3 CSV-Verarbeitung mit OpenCSV

Zum Einlesen der CSV-Dateien wurde die Bibliothek **OpenCSV** [3] eingesetzt. Diese erlaubt eine deklarative Abbildung von CSV-Spalten auf Java-Attribute mittels Annotationen:

```

1 public class Airport {
2
3     /** Unique numeric identifier for the airport. */
4     @CsvBindByName(column = "id")
5     private int id;
6
7     /** Three-letter IATA code (e.g., VIE, JFK). */
8     @CsvBindByName(column = "iata")
9     private String iata;

```

```

10
11     ...
12 }

```

Listing 1: Beispiel: Mapping einer CSV-Spalte auf ein Attribut

Die Klassen `AirportRepository`, `FlightRepository` und `RouteRepository` kapseln den Dateizugriff vollständig und behandeln auch mögliche Fehler (z. B. fehlerhafte Formatierungen oder fehlende Dateien).

2.4 Graphenbasierte Modellierung

Für die Berechnung von Flugrouten wurde ein gerichteter Graph verwendet, da Flüge stets eine Richtung besitzen (z. B. VIE nach JFK). Die interne Repräsentation erfolgt über folgende Struktur:

- **Node:** Ein Knoten entspricht einem Flughafen.
- **Edge:** Eine Kante repräsentiert einen Flug mit Gewicht (z. B. Preis oder Dauer).
- **Graph:** Verwaltet alle Knoten und Kanten in einer Map-basierten Struktur.

Diese Modellierung ermöglicht die Anwendung klassischer Pathfinding-Algorithmen wie Dijkstra [2, 4, 6] oder BFS [1, 5].

2.5 Pathfinding-Algorithmen

Die zentrale Aufgabe des Projekts besteht darin, Routen zwischen Flughäfen zu finden, die entweder die geringsten Gesamtkosten, die kürzeste Flugdauer oder die wenigsten Umstiege aufweisen. Damit handelt es sich algorithmisch um ein klassisches *Pathfinding*-Problem in einem gerichteten, gewichteten Graphen. Im Folgenden werden die beiden gewählten Algorithmen erläutert und ihre Auswahl begründet.

2.5.1 Dijkstra-Algorithmus - Kürzeste bzw. günstigste Route

Für die Berechnung der kostengünstigsten oder schnellsten Verbindung wurde der **Dijkstra-Algorithmus** gewählt. Er stellt einen der effizientesten exakten Algorithmen zur Bestimmung kürzester Wege in einem Graphen mit ausschließlich positiven Kantengewichten dar [2]. Da sowohl Flugpreise als auch Flugdauern stets positiv sind, erfüllt das zugrundeliegende Modell genau diese Voraussetzung.

Andere Ansätze wie *Bellman-Ford* oder *Floyd-Warshall* bieten zwar ebenfalls korrekte Lösungen, sind jedoch in diesem Szenario deutlich weniger effizient, da sie auf Graphen mit negativen Kanten bzw. auf vollständige Distanzmatrizen ausgelegt sind. A^* wäre eine weitere Alternative, benötigt jedoch eine geeignete Heuristik (z. B. Luftlinienentfernung), die im Kontext von Flugrouten problematisch ist: geografische Nähe bedeutet nicht automatisch kürzere Reisezeit oder geringere Kosten.

Damit stellt Dijkstra hier den idealen Kompromiss aus Genauigkeit, Laufzeiteffizienz und Implementierbarkeit dar.

```

1 PriorityQueue<String> queue =
2     new PriorityQueue<>(Comparator.comparingDouble(distance::get));

```

```
3 queue.add(origin);
```

Listing 2: Ausschnitt: Priority-Queue im Dijkstra-Algorithmus

Die Prioritätswarteschlange (`PriorityQueue`) stellt sicher, dass stets der Knoten mit der aktuell geringsten bekannten Distanz als Nächstes verarbeitet wird. Dies erlaubt eine effiziente Berechnung selbst in größeren Datensätzen mit hunderten Flughäfen und Verbindungen.

Komplexitätsanalyse

- **Zeit:** $O((V + E) \log V)$
- **Speicher:** $O(V + E)$

Zur Implementierung diente die Referenzimplementierung von Baeldung [2] sowie ergänzende Beispiele aus *W3Schools DSA* [4] und einem erklärenden YouTube-Tutorial [6].

2.5.2 BFS - Route mit den wenigsten Umstiegen

Für die Berechnung der Verbindung mit der geringsten Anzahl an Flügen wurde die **Breadth-First Search (BFS)**-Strategie verwendet. BFS arbeitet auf ungewichteten Graphen und garantiert den kürzesten Pfad in Bezug auf die Anzahl der Kanten - im Kontext dieses Projekts also die minimale Zahl an Umstiegen [1].

Die Wahl fiel bewusst auf BFS, da in diesem Fall nicht Kosten oder Zeit, sondern die reine Fluganzahl optimiert werden soll. Ein gewichteter Algorithmus wie Dijkstra oder A* wäre hier ungeeignet, da eine künstliche Gewichtung (z. B. „1 pro Flug“) den linearen BFS-Ansatz nur unnötig verkomplizieren würde.

Komplexitätsanalyse

- **Zeit:** $O(V + E)$
- **Speicher:** $O(V)$

BFS bietet somit eine sehr performante Lösung für ungewichtete Suchprobleme, während Dijkstra die optimale Wahl für gewichtete Szenarien mit positiven Kosten darstellt [5]. Zusammen bilden sie die zentrale algorithmische Grundlage der Routing-Komponente des Projekts.

3 Benutzeroberfläche und Funktionalität

3.1 Konsolenmenü

Das System bietet eine interaktive Konsolenoberfläche, die über die Klasse `ConsoleMenu` realisiert wird. Diese ermöglicht es Benutzern, alle Funktionen des Flugroutenplaners über eine einfache Menüstruktur zu nutzen.

Die Anwendung startet in der `Main`-Klasse und initialisiert automatisch das Konsolenmenü:

```
1 public static void main(String[] args) {  
2     ConsoleMenu consoleMenu = new ConsoleMenu();  
3     consoleMenu.startMenu();  
4 }
```

Listing 3: Einstiegspunkt der Anwendung

Das Hauptmenü bietet vier Optionen:

- **Option 1:** Flugroute berechnen
- **Option 2:** Flugrouten sortieren
- **Option 3:** Flüge suchen
- **Option 9:** Programm beenden

Das Menü läuft in einer Schleife, bis der Benutzer explizit die Option zum Beenden wählt.

3.2 Routenberechnung

Die Routenberechnung (Option 1) ermöglicht es, optimale Flugverbindungen zwischen zwei Flughäfen zu finden. Der Benutzer wird zunächst aufgefordert, den IATA-Code des Abflughafens einzugeben. Anschließend wird eine Liste aller verfügbaren IATA-Codes angezeigt, um die Eingabe zu erleichtern.

Nach der Eingabe von Abflug- und Zielflughafen (mit Validierung, dass beide unterschiedlich sind) kann der Benutzer zwischen vier Optimierungskriterien wählen:

1. **Günstigste Route:** Minimiert die Gesamtkosten (Dijkstra mit Preis als Gewicht)
2. **Schnellste Route:** Minimiert die Gesamtflugdauer (Dijkstra mit Dauer als Gewicht)
3. **Wenigste Umstiege:** Minimiert die Anzahl der Flüge (BFS)
4. **Langsamste Route:** Maximiert die Gesamtflugdauer (Dijkstra mit negiertem Gewicht)

Die gefundene Route wird anschließend formatiert ausgegeben und zeigt Flugdetails, Gesamtdauer, Gesamtpreis und Anzahl der Umstiege an.

3.3 Sortierfunktionalität

Die Sortierfunktion (Option 2) ermöglicht es, eine Liste von Routen nach verschiedenen Kriterien zu sortieren. Der Benutzer gibt zunächst eine komma-separierte Liste von Routen-IDs ein, die sortiert werden sollen.

Anschließend kann zwischen zwei Sortieralgorithmen gewählt werden:

- **Stabile Sortierung (MergeSort):** Erhält die relative Reihenfolge gleichwertiger Elemente
- **Instabile Sortierung (QuickSort):** Kann die Reihenfolge gleichwertiger Elemente ändern, ist jedoch in der Praxis oft schneller

Für die Sortierreihenfolge stehen folgende Komparatoren zur Verfügung:

1. **Preis (aufsteigend):** Sortiert nach Gesamtpreis, günstigste zuerst
2. **Dauer (aufsteigend):** Sortiert nach Gesamtflugdauer, schnellste zuerst
3. **Dauer (absteigend):** Sortiert nach Gesamtflugdauer, langsamste zuerst
4. **Umstiege (aufsteigend):** Sortiert nach Anzahl der Umstiege, wenigste zuerst
5. **Kombiniert:** Sortiert primär nach Airline, sekundär nach Dauer, tertiär nach Umstiegen

Die sortierten Routen werden anschließend ausgegeben.

3.4 Suchfunktionalität

Die Suchfunktion (Option 3) ermöglicht es, Flüge nach verschiedenen Kriterien zu durchsuchen. Der Benutzer kann zwischen vier Suchoptionen wählen:

1. **Suche nach Abflughafen:** Findet alle Flüge, die von einem bestimmten IATA-Code starten
2. **Suche nach Zielflughafen:** Findet alle Flüge, die zu einem bestimmten IATA-Code fliegen
3. **Suche nach Airline:** Findet alle Flüge einer bestimmten Fluggesellschaft
4. **Suche nach Flugnummer:** Findet einen spezifischen Flug anhand seiner Flugnummer

Alle Suchvorgänge sind case-insensitive (Groß-/Kleinschreibung wird ignoriert) und durchsuchen die gesamte Flugdatenbank. Wenn keine passenden Flüge gefunden werden, wird eine entsprechende Meldung ausgegeben.

4 Sortialgorithmen

Neben den Pathfinding-Algorithmen implementiert das System auch zwei klassische Sortialgorithmen, die zur Sortierung von Routen verwendet werden können. Beide Algorithmen erben von der abstrakten Basisklasse `SortingAlgorithm` und implementieren die Methode `sort(List<Route>, Comparator<Route>)`.

4.1 MergeSort - Stabile Sortierung

MergeSort wurde als stabile Sortierlösung implementiert. Der Algorithmus arbeitet nach dem Divide-and-Conquer-Prinzip:

1. Die Liste wird rekursiv in zwei Hälften geteilt
2. Beide Hälften werden rekursiv sortiert
3. Die sortierten Hälften werden zu einer sortierten Liste zusammengeführt

Vorteile von MergeSort

- **Stabilität:** Die relative Reihenfolge gleichwertiger Elemente bleibt erhalten
- **Vorhersagbare Laufzeit:** Garantiert $O(n \log n)$ in allen Fällen (best, average, worst)
- **Keine Worst-Case-Degradierung:** Im Gegensatz zu QuickSort gibt es keine ungünstigen Eingabemuster

Komplexitätsanalyse

- **Zeit:** $O(n \log n)$ (best, average, worst case)
- **Speicher:** $O(n)$ (zusätzlicher Speicher für das Merging)

MergeSort eignet sich besonders, wenn die Stabilität der Sortierung wichtig ist, beispielsweise wenn Routen mit identischen Preisen in ihrer ursprünglichen Reihenfolge erhalten bleiben sollen.

4.2 QuickSort - Instabile Sortierung

QuickSort wurde als instabile, aber in der Praxis oft schnellere Alternative implementiert. Der Algorithmus verwendet ebenfalls Divide-and-Conquer:

1. Ein Pivot-Element wird ausgewählt (randomisiert, um Worst-Case zu vermeiden)
2. Die Liste wird um das Pivot partitioniert (kleinere Elemente links, größere rechts)
3. Die beiden Partitionen werden rekursiv sortiert

Vorteile von QuickSort

- **Hervorragende durchschnittliche Laufzeit:** $O(n \log n)$ im Durchschnitt
- **Geringerer Speicherbedarf:** $O(\log n)$ für den Rekursionsstack
- **Bessere Cache-Lokalität:** In-Place-Sortierung führt zu weniger Speicherzugriffen

- **Randomisierter Pivot:** Vermeidet Worst-Case-Szenarien durch zufällige Pivot-Auswahl

Komplexitätsanalyse

- **Zeit:** $O(n \log n)$ average case, $O(n^2)$ worst case (selten durch Randomisierung)
- **Speicher:** $O(\log n)$ average case für Rekursionsstack

QuickSort eignet sich besonders für große Datenmengen, wenn Stabilität nicht erforderlich ist und maximale Performance im Durchschnittsfall gewünscht wird.

4.3 Auswahl zwischen MergeSort und QuickSort

Die Wahl zwischen den beiden Algorithmen hängt von den Anforderungen ab:

Kriterium	MergeSort	QuickSort
Stabilität	Ja	Nein
Worst-Case-Garantie	$O(n \log n)$	$O(n^2)$
Durchschnitts-Performance	$O(n \log n)$	$O(n \log n)$ (oft schneller)
Speicherbedarf	$O(n)$	$O(\log n)$
Cache-Lokalität	Schlechter	Besser

Im Kontext dieses Projekts kann der Benutzer je nach Anforderung zwischen beiden Algorithmen wählen: Wenn die ursprüngliche Reihenfolge gleichwertiger Routen wichtig ist, sollte MergeSort verwendet werden. Wenn maximale Performance bei großen Datenmengen im Vordergrund steht und Stabilität nicht erforderlich ist, ist QuickSort die bessere Wahl.

Einsatz von KI-Tools

Wir möchten darauf hinweisen, dass während der Bearbeitung der Aufgabe unterstützende generative KI-Tools (GitHub Copilot, ChatGPT, Cursor) verwendet wurden. Dies betraf insbesondere eher triviale oder sich wiederholende Tätigkeiten, wie beispielsweise das Erstellen von Kommentaren und JavaDocs, das Erstellen von Testdaten im CSV-Format oder das Anlegen einer `.gitignore`-Datei. Darüber hinaus wurden auch Unit-Tests für einige Klassen generiert, diese wurden jedoch stets sorgfältig überprüft und bei Bedarf angepasst. Ebenso trugen die Tools dazu bei, Notizen schneller in eine lesbare Form zu überführen und unsere Gedanken in einem akademisch angemessenen Stil zu formulieren. Alle generierten Inhalte wurden sorgfältig überprüft und bei Bedarf angepasst, um sicherzustellen, dass sie korrekt und relevant sind.

Der zentrale Aspekt der Arbeit - Architektur, Strukturierung und die zugrundeliegenden Algorithmen - wurden eigenständig ausgewählt und erarbeitet. Das Ziel des Einsatzes von KI war es, zeitintensive Nebentätigkeiten effizienter zu gestalten, um mehr Aufmerksamkeit auf die konzeptionellen und algorithmischen Aspekte richten zu können.

Dementsprechend betrachten wir diese Arbeit eindeutig als Eigenleistung.

Literatur

- [1] Baeldung. Breadth-first search in java, 2025. Abgerufen am 24.10.2025. URL: <https://www.baeldung.com/java-breadth-first-search>.
- [2] Baeldung. Dijkstra's algorithm in java, 2025. Abgerufen am 24.10.2025. URL: <https://www.baeldung.com/java-dijkstra>.
- [3] OpenCSV Project. Opencsv documentation, 2025. Abgerufen am 24.10.2025. URL: <https://opencsv.sourceforge.net/project-info.html>.
- [4] W3Schools. Graphs - dijkstra algorithm, 2025. Abgerufen am 24.10.2025. URL: https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php.
- [5] YouTube. Breadth-first search (bfs) algorithm tutorial, 2025. Abgerufen am 24.10.2025. URL: <https://www.youtube.com/watch?v=7Cox-J7onXw>.
- [6] YouTube. Dijkstra algorithm explained, 2025. Abgerufen am 24.10.2025. URL: <https://www.youtube.com/watch?v=bZkzH5x0SKU>.