Nicolas Haselbacher
Patrick Kristen
Michael Mayr

Flugrouten-Planung

Projektübersicht

Dieses Projekt implementiert ein Flugrouten-Planungssystem, das Flüghäfen und Flüge als gerichteten Graphen modelliert. Auf Basis dieser Graphenstruktur können optimale Routen zwischen zwei Flughäfen nach verschiedenen Kriterien berechnet, sortiert und durchsucht werden.

Hauptfunktionen

- Routenplanung: Berechnen optimaler Flugrouten nach Preis, Reisedauer oder Anzahl der Umstiege
- Sortierung: Sortierer von Routen nach verschiedenen Kriterien
- Suche: Suchen von Flügen und Flughäfen nach verschiedenen Kriterien
- Persistierung: Speichern von berechneten Routen in einer CSV-Datei

Architektur

Der Aufbau des Projekts folgt einer klaren Package-Struktur:

at.hochschule.burgenland.bswe.algo

- model — Datenmodelle
- io — CSV Ein-/Ausgabe
- graph — Graph-Datenstruktur
- algorithm — Routenberechnung
- comparator — Vergleichsoperatoren für Routen
- sorting — Sortieralgorithmen
- search — Suchalgorithmen
- ui — Benutzeroberfläche
- FlightPlannerApplication.java — Runner
- Main.java — Einstiegspunkt

Datenstruktur

Datenmodelle

- Airport: Repäresentiert einen Flughafen mit IATA-Code, Standort und Koordinaten
- Flight: Repräsentiert einen Fluzg mit Airline, Flugnummer, Dauer, Preis und Abflugzeit
- Route: Repräsentiert eine komplette Route bestehend aus mehreren Flügen

Die Daten werden beim Start der Applikation über CSV-Dateien im resources-Verzeichnis eingelesen. Flughäfen und Flüge sind hierbei für das Ausführen erforderlich, Routen können optional bereitgestellt oder mit der Applikation selbst erstellt und gespeichert werden.

```
id,iata,city,country,latitude,longitude
1,VIE,Schwechat,Österreich,48.11083,16.57083
2,JFK,New York,USA,40.63980,-73.77890
```

airports.csv

```
id,origin,destination,airline,flightNumber,duration,price,departureTime
1,VIE,FRA,Lufthansa,LH1234,75,150.50,08:00
2,VIE,MUC,Austrian,OS111,60,120.00,09:30
```

flights.csv

```
id,flights,totalDuration,totalPrice,stopovers
1,24-52,635,539.99,1
2,4,580,662.52,0
```

routes.csv

FlightGraph

Enthält die Graphenstruktur, wobei Flughäfen als Knoten und Flüge als gerichtete Kanten verwendet werden.
Die Implementierung besteht aus:

- Flughafen-Registry: verbindet alle IATA-Codes mit ihren entsprechenden Airport-Objekten
- Adjazentliste: verbindet die Flughäfen mit ihren möglichen Destinationen
- Flight-ID-Map: verbindet die IDs von Flügen mit ihren entsprechenden Flight-Objekten

Diese Struktur erlaubt durch Trennung der Verantwortlichkeiten effizienten Zugriff auf alle relevanten Daten.

Algorithmen

Routenplanung

Die Routenplanung verwendet einen modifizierten Dijkstra-Algorithmus mit Priority Queue.

Begründung

Der Dijkstra-Algorithmus bietet eine bewährte Lösung von Single-Source Shortest-Path-Problemen. Da sich Flugrouten einfach als Graph mit gewichteten Kanten modellieren lassen und der Dijkstra Algorithmus sehr erweiterbar ist, eignet er sich gut für diese Implementierung. Die Javaklasse PriorityQueue erlaubt es, custom Comparator-Implementierungen direkt einzusetzen und erlaubt so mit dem gleichen Grundalgorithmus verschiedene Suchkriterien auf eine effiziente Weise umzusetzen. Dijkstra erlaubt weiterhin das Integrieren von zusätzlichen Nebenbedingungen wie einer Umstiegszeit oder einer maximalen Anzahl an Umstiegen.

Umsetzung

In diesem Projekt wurden die folgenden Bedingungen umgesetzt:

- 20 Minuten Umstiegszeit
- Maximal 3 Umstiege (= 4 Flüge)

- Tageswechsel werden richtig behandelt -> Flüge finden jeden Tag statt
- Verschiedene Gewichtungen:
    - Günstigste Route
    - Langsamste Route
    - Schnellste Route
    - Wenigste Umstiege

Die Berechnung der langsamsten Route lässt sich umsetzen, indem der Comparator negiert wird. Dadurch priorisiert der Dijkstra-Algorithmus die Werte mit der niedrigsten Gewichtung.

Laufzeitkomplexität

Für jeden Flughafen (V), wird die Methode queue.poll(), eine O(log V) komplexe Aktion aufgerufen. Anschließend wird für jede dieser Iterationen bis zu maximal der Anzahl an Flügen (E) die Methode queue.add() aufgerufen, ebenfalls eine O(log V) komplexe Aktion.

Somit ergibt sich für den Dijkstra-Algorithmus die Gesamtkomplexität:
O(V * log V) * O (E * log V) = O((V +E) log V)

Stabiler Sortieralgorithmus

Als stabilen Sortieralgorithmus verwendet das Projekt eine Merge Sort Implementierung.

Begründung

Der größte Vorteil vom Merge Sort ist, neber seiner Stabilität,seine Vorhersagbarkeit und Datenunabhängigkeit. Durch das gleichmäßige Teilen und Zusammenführen der Liste, ist anhand der Listengröße direkt erkennbar, wieviele Schritte benötigt werden. Er ist somit für das Sortieren realistischer „zufälliger“ Listen sehr zuverlässig und schnell.

Laufzeitkomplexität

Bei jeder Rekursion wird die Liste in der Hälfte geteilt -> log n Teilungen und anschließend zusammengefügt, dabei werden in jedem der ebenfalls log n Schritte alle n Elemente durchlaufen. Somit ergibt sich eine Gesamtkomplexität von O(log n) + O(n log n) = O(n log n).

Instabiler Sortieralgorithmus

Als instabilen Sortieralgorithmus verwendet das Projekt eine Quick Sort Implementierung.

Begründung

Der Quick Sort ist ein dem Merge Sort ähnlicher, aber instabiler Sortieralgorithmus. Das macht die Implementierungen ähnlich und relativ gut vergleichbar. Er besitzt außerdem eine ähnliche Geschwindigkeit durch den ähnlichen „Divide and Conquer“ Sortieransatz und eignet sich somit ebenfalls gut für zufällige Listen.

Laufzeitkomplexität

Beim Quick Sort muss zwischen Worst-Case und Average- bzw. Best-Case unterschieden werden. Hier ist die Auswahl des Pivot-Elements entscheidend. Wenn das Pivot-Element die Liste immer genau in 2 Hälften teilt (Best-Case) gleicht die Laufzeit der des Merge Sorts. Wenn jedoch immer (zufällig, die Implementierung im Projekt verwendet immer das „letzte“ Element) das größte bzw. kleinste Element als Pivot-Element ausgewählt wird, spaltet jeder Schritt nur 1 Element ab, was zu n-

Iterationen führt. Somit ist die Gesamtkomplexität beim Quick Sort im Best- bzw. Average-Case ebenfalls O(n log n) aber im Worst-Case $O(n^2)$.

Umsetzung

In diesem Projekt wurden die folgenden Sortierkriterien umgesetzt:

- Preis (aufsteigend)
- Dauer (aufsteigend)
- Anzahl Umstiege (aufsteigend)
- Kombination (Preis, Dauer, Umstiege)

Die Kombination kann durch ein In-Serie-Schalten der entsprechenden Comparator-Implementierung erreicht werden. Sie ist damit einfach um weitere Vergleichskombinationen zu erweitern.

Suchalgorithmus

Als Suchalgorithmus wird die lineare Suche über die FlightGraph Klasse verwendet.

Begründung

Die Struktur der FlightGraph Klasse erlaubt sehr einfache Suche mithilfe der Java Streams API. Da die Datensätze überschaubar groß sind, ist eine lineare Suche ausreichend.

Laufzeitkomplexität

Die lineare Suche hat die Komplexität O(n), wobei n je nach Suchkriterium variiert:

- Abflugort: n = Anzahl der Flüge des Abflugortes
- Zielort: n = Anzahl aller Flüge
- Name der Airline: n = Anzahl aller Flüge
- Flugnummer: n = Anzahl aller Flüge

Worst-Case bei Flugnummer: O(n) (nicht gefunden), Best-Case: O(1) (erstes Element gefunden)

Umsetzung

In diesem Projekt wurden die folgenden Suchkriterien umgesetzt:

- Abflugort
- Zielort
- Name der Airline (case-insensitive)
- Flugnummer (case-insensitive)

Zusätzliche Überlegungen

Bei der Umsetzung des Projekts wurde ein Fokus auf eine klare Trennung der Verantwortlichkeiten gelegt. Als Grundlage für Wartbarkeit und Erweiterbarkeit sind so Datenhaltung, Algorithmen, Ein- und Ausgabe, sowie User-Interaktionen strikt voneinander getrennt.

Da in der Angabe die Implementierung der Berechnung der schnellsten Route, aber im Menü die Ausgabe der langsamsten Route gefordert war, wurden in diesem Projekt bewusst Versionen umgesetzt. Dadurch kann auch demonstriert werden, wie flexibel die Implementierung in der Wahl der Kriterien und Comparators ist.

Das Error-Handling wurde defensiv umgesetzt, beim Einlesen der CSV-Dateien werden fehlerhafte oder unvollständige Zeilen übersprungen, um einen stabilen Programmablauf zu gewährleisten. Benutzereingaben werden validiert und bei ungültigen Eingaben wird die aktuelle Aktion kontrolliert abgebrochen und der User zum Main Menu zurückgeleitet.

Die Testfälle sind darauf ausgelegt, sämtliche für die Algorithmen relevanten Methoden vollständig abzudecken. Sie sollen in möglichst kleinen Einheiten getestet werden und enthalten so keine vollständigen Systemtests.