

ChadGuide

Planowanie podróży po Europie

Paweł Kauf, Filip Rabiega

2 lutego 2026

Wstęp

Problem: Planowanie podróży wielomiejscowej wymaga sprawdzenia wykładniczej liczby kombinacji kolejności miast. Ręczne porównywanie jest niepraktyczne.

Cel: Znalezienie optymalnych (najtańszych/najkrótszych) tras lotniczych łączących wiele miast europejskich.

Wstęp

ChadGuide jest *ChadRozwiązańiem*.

- ▶ Oprócz optymalizacji wykonuje jeszcze walidacje lotów
- ▶ Wszystko dzieje się przez wygodną oraz przejrzystą aplikację webową

Użyte technologie

- ▶ Python: poetry, pandas, streamlit, pandera, fastapi, uvicorn, pydantic, sse-starlette, playwright, pytest
- ▶ JavaScript
- ▶ SQLite
- ▶ API: Duffel, Gepapify

Demo

- ▶ Nasz projekt można obejrzeć na
<https://github.com/fiadra/chadguide>.
- ▶ Działającą publiczną instancją jest
<https://www.chadguide.site>.

Zbieranie danych o lotach

- ▶ Zbieranie danych o lotach do bazy jest realizowane przez paczkę `flight scanner`, która łączy się z DuffelAPI (148 lotnisk, 89k lotów)
- ▶ Dane w naszej bazie zawsze będą trochę do tyłu względem prawdziwych, ponieważ koszty biletów oraz ich dostępność ciągle się zmieniają. Dane dla tygodnia 01.07.2026–07.07.2026 są ekstrapolowane na cały miesiąc.

Zbieranie danych o atrakcjach

- ▶ Zbieranie danych o atrakcjach dzieje się asynchronicznie podczas działania algorytmu. Służy do tego paczka, która odpytuje Geoapify o dane.
- ▶ Zdjęcia atrakcji zostały zebrane metodami webscrapingu.

Algorytm

- ▶ Sercem projektu jest zmodyfikowany algorytm Dijkstry, który wyznacza drogi Pareto-optymalne względem ceny oraz czasu trwania lotu. Jednocześnie radzi sobie on z grafami, które są zmienne w czasie.
- ▶ Droga p dominuje drogę q , gdy istnieje kryterium, w którym p jest lepsze od q oraz nie istnieje kryterium, w którym p jest gorsze niż q .
- ▶ Droga jest *Pareto optymalna*, gdy nie jest dominowana przez żadną inną drogę.

Algorytm

- ▶ Interesują nas tylko te drogi, które wychodzą i wracają z wyznaczonego miasta, oraz przelatują przez wszystkie miasta, które użytkownik chce odwiedzić.
- ▶ Inne drogi są automatycznie odrzucane.
- ▶ Przed uruchomieniem algorytmu wykonujemy prosty *pruning*, tzn. pozbywamy się tych połączeń w grafie, które prawie na pewno nie będą częścią optymalnego rozwiązania.

Algorytm

```
Input: a timetable graph and a query
Output: a set of Pareto-optimal labels at the terminal

1 foreach node v do
2   list<Label> labelListAt(v) := ∅;
3 PriorityQueue pq := ∅;
4 foreach node v in start interval do
5   Label startLabel := createStartLabel(v);
6   pq.insert(startLabel);

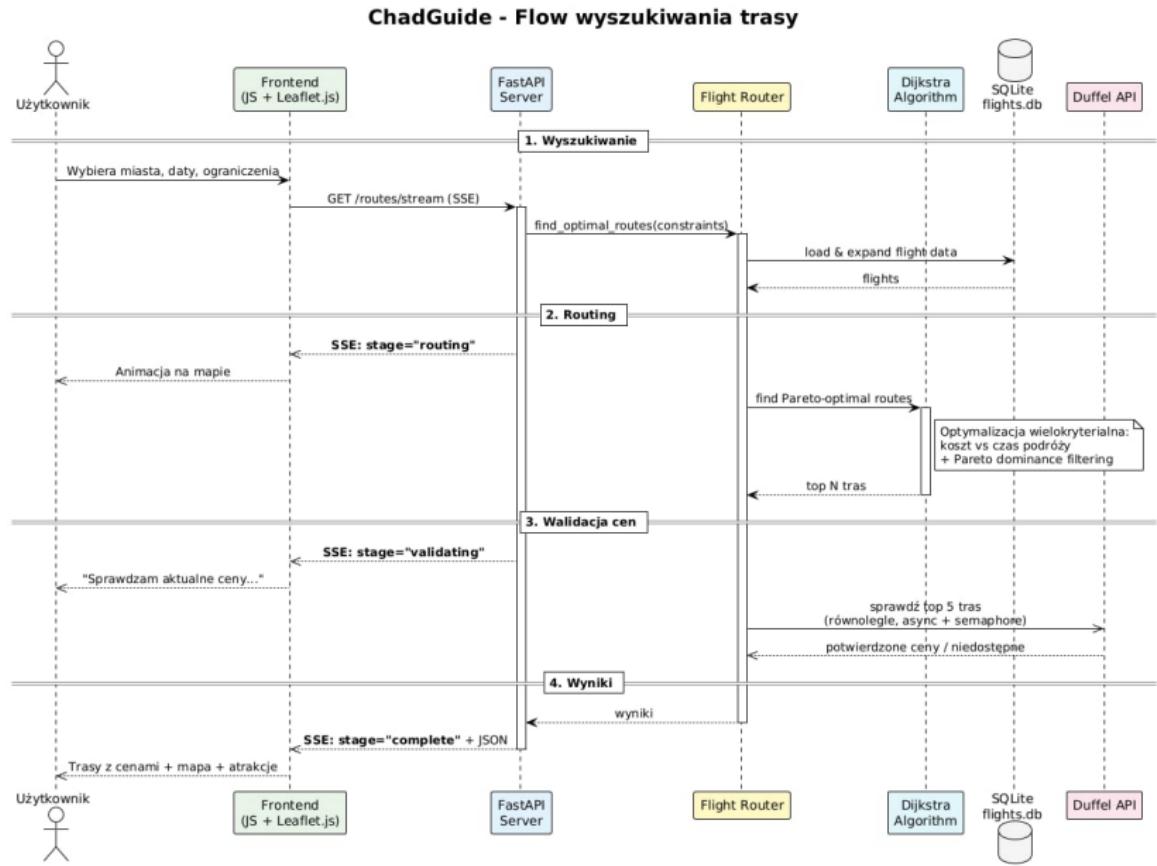
7 while ! pq.isEmpty() do
8   Label label := pq.extractLabel();
9   foreach outgoing edge e=(v,w) of v=label.getNode() do
10    if isInfeasible(e) then continue; // ignore this edge
11    Label newLabel := createLabel(label, e);
12    if newLabel is dominated then continue;
13    // newLabel is not dominated
14    pq.insert(newLabel);
15    labelListAt(w).insert(newLabel);
16    labelListAt(w).removeLabelsDominatedBy(newLabel);
```

Algorithm 1. Pseudocode for the generalized Dijkstra algorithm

Walidacja

- ▶ Znalezione drogi są *walidowane*, tzn. sprawdzamy, czy faktycznie istnieją.
- ▶ Top 5 tras jest weryfikowanych asynchronicznie z API w czasie rzeczywistym.

Diagram



Git, dokumentacja, testy, ...

- ▶ Całość powstała dzięki systemowi kontroli wersji Git.
- ▶ Staraliśmy się prowadzić dokładną dokumentację poszczególnych modułów oraz paczek.
- ▶ Do wszystkich istotnych części projektu napisaliśmy testy, również te wydajnościowe. Do testów została użyta paczka pytest.

Co mogliśmy zrobić lepiej?

- ▶ Za dużo czasu zajęło nam znalezienie odpowiedniego dostawcy danych lotów.
- ▶ blabla

Podsumowanie