

# ChadGuide

## Planowanie podróży po Europie

Paweł Kauf, Filip Rabięga

2 lutego 2026

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

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

*ChadGuide* jest *ChadRozwiązaniem*.

- ▶ 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>.

# 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    $\text{list}\langle\text{Label}\rangle \text{labelListAt}(v) := \emptyset;$ 
3  $\text{PriorityQueue } pq := \emptyset;$ 
4 foreach node  $v$  in start interval do
5    $\text{Label } \text{startLabel} := \text{createStartLabel}(v);$ 
6    $pq.\text{insert}(\text{startLabel});$ 
7 while  $\neg pq.\text{isEmpty}()$  do
8    $\text{Label } \text{label} := pq.\text{extractLabel}();$ 
9   foreach outgoing edge  $e=(v,w)$  of  $v=\text{label.getNode}()$  do
10    if  $\text{isInfeasible}(e)$  then  $\text{continue};$  // ignore this edge
11     $\text{Label } \text{newLabel} := \text{createLabel}(\text{label}, e);$ 
12    if  $\text{newLabel}$  is dominated then  $\text{continue};$ 
13    //  $\text{newLabel}$  is not dominated
14     $pq.\text{insert}(\text{newLabel});$ 
15     $\text{labelListAt}(w).\text{insert}(\text{newLabel});$ 
16     $\text{labelListAt}(w).\text{removeLabelsDominatedBy}(\text{newLabel});$ 
```

**Algorithm 1.** Pseudocode for the generalized Dijkstra algorithm



- ▶ Znalezione drogi są *walidowane*, tzn. sprawdzamy, czy faktycznie istnieją.
- ▶ 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.
- ▶ Robi to porozumiewając się z DuffelAPI.

- ▶ Dla każdego miasta w bazie mamy do 5 zdjęć popularnych atrakcji z tego miasta.
- ▶ Atrakcje te są wyświetlane podczas oczekiwania na zwrócenie wyników.