Sztuczna inteligencja i inżynieria wiedzy Lista 1

Mikołaj Olesiński

Marzec 2025

Cały kod dostępny jest w pliku lista1.ipynb

1 Wprowadzenie

Ten raport opisuje implementację i porównanie algorytmów służących do wyszukiwania optymalnych tras w transporcie publicznym. W projekcie zaimplementowano dwa główne algorytmy: Dijkstrę oraz A*, wraz z różnymi funkcjami heurystycznymi dla algorytmu A*. Naszym celem było zbadanie efektywności tych algorytmów pod względem czasu obliczeń, optymalności trasy oraz liczby przesiadek.

2 Opis teoretyczny

2.1 Algorytm Dijkstry

Algorytm Dijkstry to klasyczny algorytm wyszukiwania najkrótszej ścieżki w grafie ważonym o nieujemnych wagach krawędzi. Jego główną zaletą jest gwarancja znalezienia optymalnego rozwiązania. W kontekście wyszukiwania tras w transporcie publicznym, algorytm Dijkstry:

- Rozpoczyna od przystanku startowego, przypisując mu koszt 0.
- Wykorzystuje kolejkę priorytetową do wyboru przystanku o najniższym koszcie.
- Dla każdego sąsiada wybranego przystanku, aktualizuje koszt dotarcia do niego, jeśli znaleziono tańszą trasę.
- Proces powtarza się, aż wszystkie przystanki zostaną przetworzone lub osiągnięty zostanie przystanek docelowy.

Złożoność czasowa algorytmu wynosi:

$$O(E + V \log V),$$

gdzie E to liczba krawędzi (połączeń), a V to liczba wierzchołków (przystanków).

3 Algorytm A*

Algorytm A* stanowi rozszerzenie algorytmu Dijkstry poprzez wykorzystanie funkcji heurystycznej do przewidywania odległości od aktualnego punktu do celu. A* używa funkcji oceny:

$$f(n) = g(n) + h(n),$$

gdzie:

- $g(n)$ to koszt dotarcia do węzła n od startu (jak w Dijkstrze),
- $h(n)$ to funkcja heurystyczna szacująca koszt z węzła n do celu.

Zalety algorytmu A*

- Zazwyczaj przeszukuje mniejszą liczbę węzłów niż Dijkstra, co przyspiesza działanie.
- Gwarantuje znalezienie optymalnego rozwiązania, jeśli funkcja heurystyczna jest w dobrym zakresie.
- Może być dostosowany do różnych scenariuszy poprzez wybór odpowiedniej heurystyki (np. odległość euklidesowa, Manhattan, rzeczywisty czas podróży).

4 Przykład działania

4.1 Algorytm Djisktry w oparciu o kryterium czasu

Algorytm Dijkstry znajduje najszybszą trasę, uwzględniając rzeczywiste godziny odjazdów i przyjazdów. Przystanki są przetwarzane według najkrótszego czasu dotarcia, a połączenia wybierane tak, aby podróż trwała jak najmniej. Do znalezienia najszybszego połączenia z uwzględnieniem przesiadek używamy funkcji `find_earliest_connection`.

- **Start:** Wilczyce - Sosnowa

- **Cel:** Kołobrzeska

Linia	Przystanek początkowy	Przystanek końcowy
911	Wilczyce - Sosnowa (07:10:00)	C.H. Korona (07:25:00)
131	C.H. Korona (07:26:00)	Brücknera (07:27:00)
128	Brücknera (07:29:00)	KROMERA (07:35:00)
116	KROMERA (07:36:00)	Kasprowicza (07:39:00)
118	Kasprowicza (07:42:00)	most Milenijny (07:55:00)
104	most Milenijny (07:56:00)	Kwiska (08:02:00)
122	Kwiska (08:03:00)	KUŹNIKI (08:12:00)
129	KUŹNIKI (08:14:00)	Kołobrzeska (08:15:00)

- **Liczba przesiadek:** 7
- **Koszt trasy:** 75.0 min
- **Czas obliczeń:** 1.7978 sek

4.2 Algorytm A* w oparciu o kryterium czasu

Algorytm A* znajduje najkrótszą trasę, uwzględniając rzeczywiste godziny odjazdów oraz funkcję heurystyczną przewidującą czas podróży do celu. Podobnie jak w Dijkstrze, analizujemy możliwe połączenia, ale dodatkowo oceniamy, które z nich mogą prowadzić do szybszego dotarcia do celu. Do wyboru najwcześniejszego połączenia, biorąc pod uwagę przesiadki, wykorzystujemy funkcję `find_earliest_connection`. Dzięki temu algorytm jest bardziej efektywny, ponieważ priorytetowo traktuje trasy, które mają większe szanse na szybsze dotarcie do celu.

- **Start:** Wilczyce - Sosnowa
- **Cel:** Kołobrzeska

Linia	Przystanek początkowy	Przystanek końcowy
911	Wilczyce - Sosnowa (07:10:00)	C.H. Korona (07:25:00)
131	C.H. Korona (07:26:00)	Brücknera (07:27:00)
128	Brücknera (07:29:00)	KROMERA (07:35:00)
116	KROMERA (07:36:00)	Kasprowicza (07:39:00)
118	Kasprowicza (07:42:00)	most Milenijny (07:55:00)
104	most Milenijny (07:56:00)	Kwiska (08:02:00)
122	Kwiska (08:03:00)	KUŹNIKI (08:12:00)
129	KUŹNIKI (08:14:00)	Kołobrzeska (08:15:00)

- **Liczba przesiadek:** 7
- **Koszt trasy:** 75.0 min
- **Czas obliczeń:** 1.0641 sek

4.3 Algorytm A* w oparciu o kryterium przesiadek

W celu znalezienia optymalnej trasy z minimalną liczbą przesiadek wykorzystujemy klasę `AstarRouterForTransfers`. Algorytm A* poszukuje najkrótszej ścieżki w grafie przystanków, uwzględniając zarówno czas przejazdu, jak i liczbę przesiadek.

- **Start:** Wilczyce - Sosnowa
- **Cel:** Kołobrzeska

Linia	Przystanek początkowy	Przystanek końcowy
911	Wilczyce - Sosnowa (07:10:00)	PL. GRUNWALDZKI (07:38:00)
149	PL. GRUNWALDZKI (07:44:00)	Kołobrzeska (08:17:00)

- **Liczba przesiadek:** 1
- **Koszt trasy:** 78.0 min
- **Czas obliczeń:** 37.3787 sek

Podczas przeszukiwania grafu, dla każdego przystanku analizowane są dostępne połączenia. Priorytetowo wybierane są te, które umożliwiają kontynuację podróży bez zmiany linii, co pozwala ograniczyć liczbę przesiadek. W przypadku konieczności zmiany linii doliczany jest dodatkowy koszt w postaci kary za przesiadkę.

Do wyznaczania najlepszej trasy wykorzystujemy algorytm A*, który ocenia możliwe ścieżki na podstawie przewidywanego czasu podróży oraz liczby przesiadek. Podczas eksploracji dostępnych połączeń uwzględniane są zarówno połączenia bezpośrednie, jak i te wymagające zmiany linii, przy czym każda przesiadka podlega dodatkowej ocenie.

Algorytm kończy działanie po znalezieniu najkrótszej trasy do przystanku docelowego, minimalizując zarówno czas podróży, jak i liczbę przesiadek.

4.4 Zoptymalizowany Algorytm A*

Wykorzystuje zmodyfikowaną heurystykę, minimalizując czas podróży i liczbę przesiadek.

4.4.1 Funkcja kary oparta o przesiadkę

Dodaje karę za przesiadkę, ale stara się znaleźć złoty między czasem a liczbą przesiadek.

4.4.2 Funkcja kary oparta o dystans

Dodaje karę proporcjonalną do odległości do końcowego przystanku, zwiększając koszt tras które się oddalają od końcowego przystanku.

4.4.3 Funkcja kary oparta o kąt

Jeśli przesiadka prowadzi w znacząco innym kierunku, dodaje dodatkową karę, by unikać nieefektywnych tras.

4.4.4 Hybryda funkcji kary

Łączy karę odległości i kąta, by preferować trasy krótsze i o bardziej naturalnym kierunku jazdy.

- **Start:** Wilczyce - Sosnowa
- **Cel:** Kołobrzeska

Linia	Przystanek początkowy	Przystanek końcowy
911	Wilczyce - Sosnowa (07:10:00)	C.H. Korona (07:25:00)
131	C.H. Korona (07:26:00)	Brücknera (07:27:00)
128	Brücknera (07:29:00)	KROMERA (07:35:00)
116	KROMERA (07:36:00)	Kasprowicza (07:39:00)
118	Kasprowicza (07:42:00)	most Milenijny (07:55:00)
104	most Milenijny (07:56:00)	Milenijna (Hala Orbita) (07:57:00)
134	Milenijna (Hala Orbita) (08:02:00)	Na Ostatnim Groszu (08:06:00)
129	Na Ostatnim Groszu (08:07:00)	Kołobrzeska (08:15:00)

- **Liczba przesiadek:** 7
- **Koszt trasy:** 75.0 min
- **Czas obliczeń:** 0.2325 sek

5 Porównanie wyników

Wyniki są na podstawie 20 losowo wygenerowanych tras

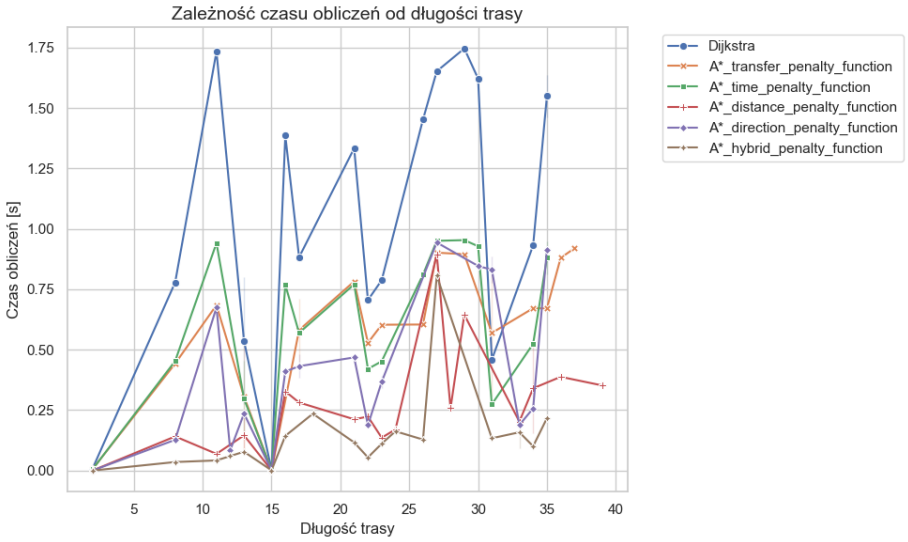
Tabela 1: Średnie wyniki dla algorytmów

Algorytm	**Koszt**	**Czas**	**Przesiadki**	**Węzły**	**Krawedzie**
A* For Transfer	75.40	18.847225	1.35	4890.65	17124.00
A* Direction Penalty Function	57.00	0.430020	3.15	244.40	320.95
A* Distance Penalty Function	56.10	0.263795	3.05	121.75	180.85
A* Hybrid Penalty Function	56.95	0.144365	3.35	67.35	106.35
A* Time Penalty Function	55.25	0.603015	2.80	359.05	460.25
A* Transfer Penalty Function	56.05	0.575375	2.80	344.20	467.50
Dijkstra	55.25	1.060155	2.80	364.40	466.45

Analiza wyników:

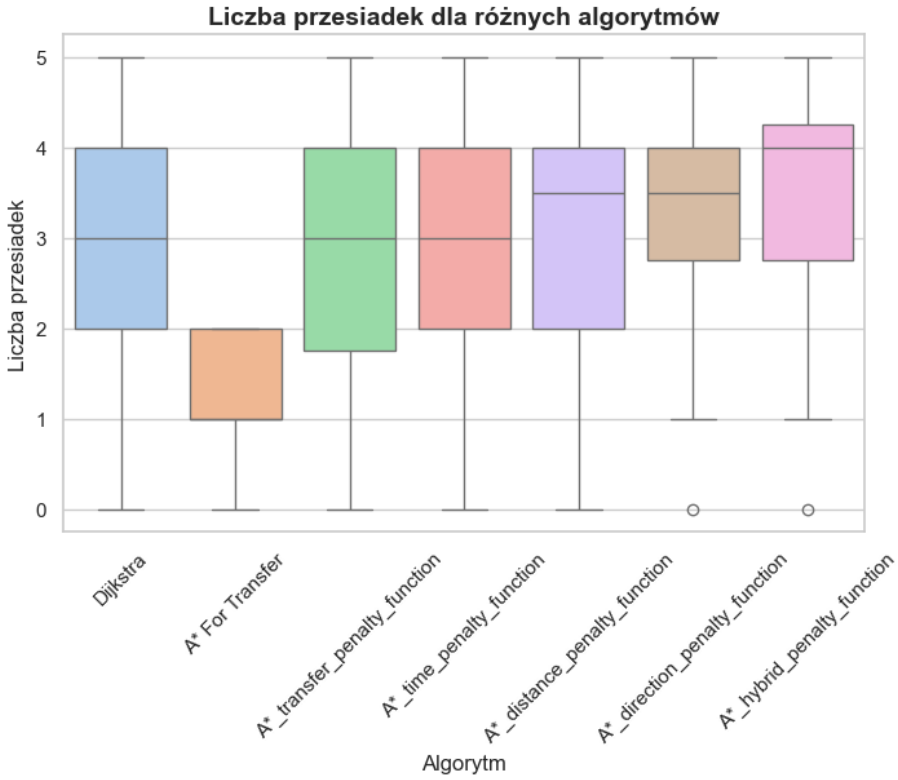
- Najbardziej optymalnym algorytmem wydaje się **A* Hybrid Penalty Function**, który osiąga dobre wyniki pod względem kosztu i liczby przesiadek, a jednocześnie ma najkrótszy czas obliczeń.
- **A* Distance Penalty Function** również wypada dobrze, zapewniając stosunkowo niski koszt i krótki czas wykonania.
- Algorytm **A* For Transfer** zdecydowanie minimalizuje liczbę przesiadek (1.35), ale odbywa się to kosztem znacznie wyższego kosztu i bardzo długiego czasu obliczeń.
- Algorytmy wykorzystujące funkcje kary (penalty functions) pozwalają na lepsze dostosowanie wyników do konkretnych potrzeb, np. minimalizacji kosztu lub liczby przesiadek. Widać również, że mają mniejszą ilość odiwedzonych węzłów i krawedzi niż Djikstra.
- **A* Time Penalty Function** oraz **A* Transfer Penalty Function** uzyskują podobne wyniki do Dijkstry, ale działają szybciej, co czyni je bardziej efektywnymi w praktycznych zastosowaniach.
- Wybór algorytmu zależy od priorytetów – jeśli kluczowe jest zminimalizowanie przesiadek, **A* For Transfer** może być odpowiedni, ale dla równowagi między czasem i kosztem lepiej sprawdzają się **A* Hybrid** i **A* Distance**.

Algorytm **A* For Transfer** znacząco wyróżnia się pod względem całkowitego kosztu trasy oraz liczby odwiedzonych węzłów i krawędzi. Z tego powodu nie będziemy uwzględniać go w niektórych wykresach, aby lepiej uwidocznić różnice pomiędzy pozostałymi metodami.



Rysunek 1: Zależność czasu obliczeń od długości trasy

Wykres pokazuje, że Dijkstra liczy najdłużej, zwłaszcza dla dłuższych tras. Najlepiej wypadają **A* distance penalty function** i **A* hybrid penalty function**, a **A* direction penalty function** czasem się wyróżnia, co sugeruje, że dobrze działa dla tras liniowych. Im trasa dłuższa, tym większe różnice między algorytmami. W niektórych przypadkach wszystkie metody liczą podobnie, co może oznaczać, że była dostępna tylko jedna sensowna trasa. To pokazuje, że w pewnych sytuacjach wybór algorytmu nie ma dużego znaczenia. Dla **A* direction penalty function** widizmy spory rozrzut, gdy jest dostępna szybka trasa w lini prostej do celu to jest optymalny, ale w przypadkach gdy trzeba odbijać od celu może sobie nie poradzić.



Rysunek 2: Liczba przesiadek dla różnych algorytmów

- **A* For Transfer ma najmniejszy rozrzut** – wyniki są mocno skupione wokół 1-2 przesiadek, co oznacza, że algorytm skutecznie minimalizuje ich liczbę.
- **Dijkstra i inne warianty A* mają szerszy rozkład** – w zależności od przypadku liczba przesiadek może być wyższa, co wpływa na wyniki algorytmu.
- **A* hybrid_penalty_function ma najwyższą medianę liczby przesiadek** – sugeruje to, że łączenie różnych kar może skutkować większą liczbą przesiadek, być może kosztem innych optymalizacji (np. czasu czy dystansu).
- **A* distance_penalty_function i A* direction_penalty_function mają bardziej symetryczne rozkłady** – wskazuje to na ich bardziej zrównoważone działanie w różnych scenariuszach.
- **A* transfer_penalty_function ma medianę podobną do Dijkstry, ale mniejszy rozrzut** – co sugeruje, że skutecznie ogranicza ekstremalne przypadki z dużą liczbą przesiadek.

- **Niektóre warianty A* mają więcej wartości odstających** – np. A* hybrid_penalty_function i A* direction_penalty_function wykazują obecność wartości odstających w postaci punktów poniżej dolnego wąsa, co może oznaczać sporadyczne przypadki znacznego zmniejszenia liczby przesiadek.

Podsumowując, algorytmy A* wykazują dużą różnorodność pod względem liczby przesiadek, a ich skuteczność zależy od wybranej funkcji kary. Algorytm A* For Transfer wydaje się najlepszym wyborem do minimalizacji przesiadek, natomiast inne warianty mogą wprowadzać większe zróżnicowanie wyników.

6 Tabu search

6.1 Wprowadzenie

Tabu Search to zaawansowana metoda optymalizacyjna, która pozwala na efektywne przeszukiwanie przestrzeni rozwiązań problemu. W odróżnieniu od prostych algorytmów lokalnego przeszukiwania, Tabu Search wykorzystuje mechanizm "pamięci"(listę tabu), który zapobiega powrotowi do niedawno odwiedzonych rozwiązań, co pomaga uniknąć utknięcia w lokalnych minimach.

W naszej implementacji Tabu Search stosujemy do optymalizacji tras w transporcie publicznym, co pozwala na znalezienie optymalnej sekwencji odwiedzania przystanków.

6.2 Implementacja algorytmu

Zaimplementowaliśmy algorytm Tabu Search w klasie `TSPSolver`, która służy do rozwiązywania problemu komiwojażera (TSP) w kontekście transportu publicznego. Nasza implementacja zawiera kilka kluczowych modyfikacji.

6.2.1 Dynamiczny dobór długości listy tabu

Zaimplementowaliśmy mechanizm dynamicznego dostosowywania długości listy tabu w zależności od rozmiaru problemu. Takie podejście pozwala na:

- Automatyczne dostosowanie intensywności przeszukiwania do złożoności problemu,
- Uniknięcie zbyt restrykcyjnej listy tabu dla małych problemów,
- Zapewnienie wystarczającej pamięci dla dużych problemów, co zapobiega powracaniu do tych samych rozwiązań.

6.2.2 Mechanizm aspiracji

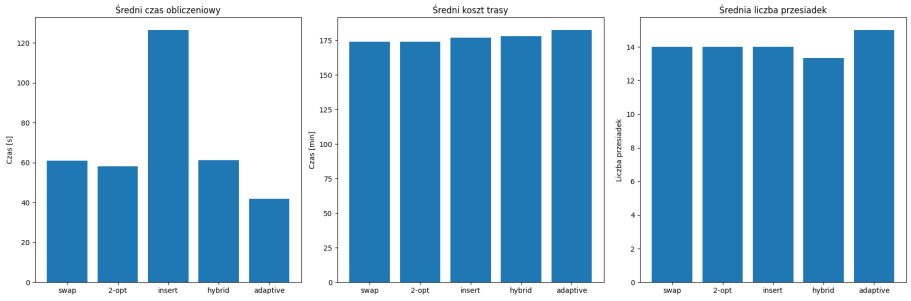
Zaimplementowaliśmy kryterium aspiracji, które pozwala na akceptację ruchów znajdujących się na liście tabu, jeśli prowadzą do rozwiązań lepszych niż dotychczas znalezione najlepsze rozwiązanie. Dzięki temu mechanizmowi:

- Algorytm może "przełamać"ograniczenia nałożone przez listę tabu, jeśli prowadzi to do znaczącej poprawy rozwiązania,
- Zwiększa się elastyczność przeszukiwania, co często prowadzi do znalezienia lepszych rozwiązań.

6.2.3 Strategie próbkowania sąsiedztwa

Nasza implementacja zawiera różnorodne strategie próbkowania sąsiedztwa, co pozwala na wybór najlepszej metody generowania sąsiednich rozwiązań. Zaimplementowane strategie obejmują:

- **Swap** - zamiana pozycji dwóch przystanków,
- **2-opt** - odwrócenie segmentu trasy,
- **Insert** - przeniesienie przystanku na inną pozycję,
- **Hybrid** - mieszanka różnych strategii dla zwiększenia dywersyfikacji,
- **Adaptive** - dostosowanie strategii w zależności od rozmiaru problemu.



Rysunek 3: Porównanie strategii próbkowania w tabu search

Analizując przedstawione wykresy, można wyciągnąć następujące wnioski:

- Metoda **2-opt** wykazuje korzystny balans między czasem obliczeniowym a kosztem trasy – oferuje drugi najkrótszy czas obliczeń przy stosunkowo niskim koszcie trasy.
- Metoda **adaptive** osiąga najlepsze wyniki pod względem czasu obliczeniowego, ale generuje najwyższy koszt trasy.
- Metoda **insert** jest zdecydowanie najwolniejsza (ponad dwukrotnie dłuższy czas obliczeń niż pozostałe metody), nie oferując przy tym znaczącej przewagi kosztowej.
- Pod względem liczby przesiadek, metoda **hybrid** wypada najkorzystniej, podczas gdy **adaptive** generuje najwięcej przesiadek.

Podsumowując, metoda **2-opt**, biorąc pod uwagę zarówno koszt trasy, jak i czas obliczeń, wypada najlepiej spośród wszystkich analizowanych metod, choć jej przewaga nie jest bardzo znacząca w porównaniu do metod **swap** czy **hybrid**.

7 Napotkane problemy

- **Problem z optymalizacją czasu przesiadki** - trudność w dostosowaniu optymalnego czasu potrzebnego na przesiadkę w różnych lokalizacjach.
- **Dobranie idealnej funkcji kary** - zbyt duża lub zbyt mała wartość kary dawała istotnie różne wyniki; znalezienie balansu było wyzwaniem.
- **Wiele przystanków o tych samych nazwach** - przystanki o identycznych nazwach zlokalizowane blisko siebie utrudniały jednoznaczną identyfikację i podejście do problemu.
- **Długi czas wykonywania się Tabu Search** - problemy z wydajnością algorytmu i trudność w dobraniu optymalnych parametrów dla większej ilości przysntaków, czy większej ilości iteracji..

8 Wykorzystane biblioteki

- **import pandas as pd** - do przetwarzania danych, analizy i manipulacji strukturami danych, konwersji typów i efektywnego operowania na ramkach danych.

- **import heapq** - wykorzystany do implementacji kolejki priorytetowej niezbędnej w algorytmach Dijkstry i A*, zapewniając efektywne wybieranie wierzchołków o najniższym koszcie.
- **import time** - do pomiaru czasu wykonania algorytmów, umożliwiając porównanie ich efektywności.
- **from datetime import datetime, timedelta** - do obsługi danych czasowych, obliczania różnic czasowych między odjazdami i przyjazdami oraz zarządzania harmonogramami transportu.
- **from math import radians, sin, cos, sqrt, atan2, degrees** - funkcje matematyczne używane do obliczania odległości geograficznych między przystankami (wzór haversine) oraz kątów między punktami na trasie, co było kluczowe dla funkcji heurystycznych.
- **import seaborn as sns** - do tworzenia wykresów
- **import matplotlib.pyplot as plt** - do tworzenia wykresów
- **import networkx as nx** - do stworzenia wizualizacji trasy