

# SPDB

## Projekt Znajdowanie najlepszej trasy

Jakub Dzięgielewski  
Jakub Śliwa

Politechnika Warszawska

31 maja 2024

### Spis treści

<b>Wstęp</b> . . . . .	2
<b>Założenia wstępne</b> . . . . .	2
Baza Danych . . . . .	2
Geomapowanie . . . . .	2
Tworzenie Grafu . . . . .	2
Wyznaczenie optymalnej kolejności odwiedzanych punktów . . . . .	2
Algorytm wyznaczania najszybszej trasy . . . . .	2
Wyznaczanie skrętów w lewo . . . . .	2
Minimalizacja liczby skrętów w lewo . . . . .	3
Reprezentacja wyników . . . . .	3
Porównanie z innymi rozwiązaniami . . . . .	3
Środowisko . . . . .	3
<b>Dokumentacja końcowa</b> . . . . .	4
Opis metody wyznaczania najlepszej trasy . . . . .	4
Rozwiązywanie problemu komiwojażera . . . . .	4
Wyszukiwanie najlepszej ścieżki między dwoma punktami . . . . .	4
Zniechęcanie do wykonywania skrętów w lewo . . . . .	4
Wyznaczanie skrętów w lewo . . . . .	4
Opis modelu danych . . . . .	5
Node . . . . .	5
Way . . . . .	5
Opis architektury aplikacji . . . . .	5
Core . . . . .	5
Frontend . . . . .	6
Parametry startowe aplikacji . . . . .	8
Wykorzystywane biblioteki . . . . .	8
Core . . . . .	8
Frontend . . . . .	9
Opis i wyniki testów . . . . .	10
Wpływ wysokości kary za skręt w lewo na kształt wyznaczanej trasy . . . . .	10
Dopuszczalność stosowanej heurystyki w algorytmie A* . . . . .	12
Optymalność znalezionej trasy - przypadek pesymistyczny . . . . .	13
Porównanie algorytmu Nearest Neighbor oraz brutalnego . . . . .	14
Wnioski z realizacji projektu . . . . .	16
<b>Bibliografia</b> . . . . .	17

## **Wstęp**

Projekt z przedmiotu SPDB polega na znajdowaniu najszybszej trasy pomiędzy kilkoma wybranymi adresami. Rozwiązaniem projektu będzie aplikacja, której użytkownik będzie podawał adres początkowy oraz listę adresów, które musi odwiedzić. Program będzie wyznaczał trasę najbardziej optymalną pod względem oczekiwanej czasu przejazdu, równocześnie minimalizując liczbę wykonywanych skrętów w lewo.

## **Założenia wstępne**

W celu zapewnienia kompletności dokumentacji projektowej pozostawiono tę sekcję bez zmian w stosunku do jej stanu sprzed rozpoczęcia implementacji aplikacji. Niemniej jednak wiele z założeń ostatecznie uległo pewnym modyfikacjom m.in. zrezygnowano z postawienia bazy danych PostGIS na rzecz pobierania danych za pośrednictwem biblioteki pyrosm, a graf reprezentujący sieć drogową uzyskiwany jest również z wykorzystaniem tej biblioteki w miejsce proponowanego wcześniej narzędzia osm2po.

## **Baza Danych**

Do rozwiązania problemu wykorzystane zostaną dane z bazy OpenStreetMap dla województwa mazowieckiego, które będą przechowywane w bazie danych PostGIS.

## **Geomapowanie**

Użytkownik będzie podawał listę adresów, a stworzony program będzie wykorzystywał narzędzie Nominatim API w celu przetłumaczenia tych adresów na współrzędne geograficzne. Każdy z podanych przez użytkownika punktów zostanie utożsamiony z geograficznie najbliższym znajdującym się w grafie węzłem.

## **Tworzenie Grafu**

Graf modelujący sieć drogową zostanie utworzony na starcie aplikacji i w czasie jej wykonywania pozostanie wczytany w pamięci operacyjnej. Do stworzenia grafu planowane jest użycie narzędzia osm2po, które na podstawie surowych danych przestrzennych w formacie .osm.pbf generuje skrypt SQL tworzący strukturę grafową. Po wczytaniu grafu z bazy krawędziom zostaną nadane odpowiednie wagи zgodnie ze wzorem ( $\text{distance-in-meters} / (\text{speed-limit-kmph} / 3.6)$ ), czyli szacowany czas przejazdu w sekundach. W przypadku, gdy dla jakiejś drogi nie będzie informacji o dopuszczalnej maksymalnej prędkości, waga zostanie wyznaczona na podstawie kategorii (standardu) drogi.

## **Wyznaczenie optymalnej kolejności odwiedzanych punktów**

Problem komiwojażera ma złożoność  $O(n!)$ , toteż w rozwiązaniu brutalnym dla większej liczby miejsc do odwiedzenia czas oczekiwania byłby bardzo długi. Zaplanowano, że do wyznaczenia optymalnej kolejności odwiedzania poszczególnych punktów zostanie wykorzystana heurystyka podróży do geograficznie najbliższego, nieodwiedzonego jeszcze punktu (Nearest Neighbour). Nie wykluczono jednak możliwości, że w trakcie implementacji heurystyka ta zostanie zastąpiona przez inną.

## **Algorytm wyznaczania najszybszej trasy**

Zaplanowano wykorzystanie algorytmu A\* z heurystyką mierzącą odległość euklidesową pomiędzy danym węzłem, a węzłem docelowym i optymistycznie transformującą ją na czas przejazdu przy założeniu maksymalnej dopuszczalnej prędkości poruszania się (osobno w mieście i poza nim).

## **Wyznaczanie skrętów w lewo**

Skręty w lewo wyznaczane będą na podstawie wartości kata pomiędzy wektorami. Wektorami będą krawędzie wchodząca i wychodząca z danego skrzyżowania (węzła). Korzystając z odpowiednich wzorów wyznaczone zostaną cosinus i sinus tego kąta. Wartość kąta w radianach wyznaczona zostanie dzięki funkcji arccos, a następnie za pomocą znaku sinusa odpowiednio skorygowana. Posiadając kąt można określić, czy znajduje się on

w ustalonym wcześniej przedziale (należy uniknąć sytuacji, że lekkie odbicie drogi w lewo traktowane jest jako skręt) np.  $[5\pi/4, 2\pi]$ .

### **Minimalizacja liczby skrętów w lewo**

W celu minimalizacji liczby skrętów w lewo dodane zostaną parametry kary w momencie, gdy wykonywany jest taki skręt. Dla każdej z trzech dostępnych opcji zostanie stworzony osobny parametr: skręt z drogi o wyższym standardzie w drogę o niższym standardzie, skręt w drogę o takim samym standardzie oraz skręt w drogę o wyższym standardzie. Wartość odpowiedniego parametru dodawana będzie do całkowitego dotychczasowego kosztu ścieżki (w zależności od postaci grafu nastąpi to albo na poziomie danych albo na poziomie algorytmu A\*).

### **Reprezentacja wyników**

Wynikiem działania programu będzie lista węzłów ustawiona zgodnie z sugerowaną kolejnością przejścia. Dzięki współrzednym geograficznym każdego węzła, możliwe będzie graficzne przedstawienie ścieżki np. za pomocą narzędzia geojson.io

### **Porównanie z innymi rozwiązaniami**

W celu porównania stworzonego algorytmu z innymi rozwiązaniami, zostaną przeprowadzone testy dla znajdowania ścieżki kilku przykładowych problemów z różną liczbą miejsc do odwiedzenia. Przede wszystkim sprawdzone zostanie, czy uzyskana w wyniku optymalizacji trasa znacząco różni się od tej zwracanej przez znane narzędzie optymalizujące GraphHopper Route Optimization API. Porównane zostaną również czasy oczekiwania na znalezienie rozwiązania przez obie implementacje. Zbadany zostanie również wpływ zmian wartości parametrów kary za skręt w lewo na kształt zwracanej trasy.

### **Środowisko**

Ostatecznie podjęto decyzję, że projekt realizowany będzie w języku Python.

## Dokumentacja końcowa

Kod źródłowy dostępny jest w repozytorium w repozytorium QuickestPath.

### Opis metody wyznaczania najlepszej trasy

Celem naszego projektu była implementacja aplikacji służącej do znajdowania ścieżki o możliwie najkrótszym przewidywanym czasie przejazdu między wieloma zadanimi przez użytkownika punktami, przy dodatkowym kryterium minimalizacji liczby skrętów w lewo. Poniżej zamieszczamy opis rozwiązania podproblemów składających się na całość zadania projektowego.

### Rozwiązańe problemu komiwojażera

Zadany problem jest znany pod nazwą problemu komiwojażera, z tą drobną modyfikacją, że w naszym wydaniu po odwiedzeniu wszystkich punktów nie powracamy do punktu startowego. Brutalny algorytm znajdujący optymalne rozwiązanie ma złożoność czasową  $O(n!)$ , przez co już dla niewielkiej liczby punktów czas oczekiwania na wynik staje się niemożliwie długi, a aplikacja stosująca takie podejście - niepraktyczna. Z tego względu zdecydowaliśmy się na zastosowanie algorytmu aproksymacyjnego Nearest Neighbor polegającego na kolejnym wyznaczaniu ścieżki do najbliższego geograficznie i dotychczas nieodwiedzonego punktu. Algorytm ten działa w racjonalnym czasie i w większości przypadków znajduje akceptowalne rozwiązanie, lecz zazwyczaj nie jest ono optymalne, szczególnie w przypadku złożliwie spreparowanych danych. Implementacja tej warstwy algorytmu znajduje się w klasie TravelSalesmanSolver.

### Wyszukiwanie najlepszej ścieżki między dwoma punktami

Zgodnie z założeniami wstępnyimi, do wyszukiwania najlepszej trasy pomiędzy dwoma zadanimi punktami korzystamy z algorytmu A\* z optymistyczną heurystyką zakładającą, że pomiędzy kolejnym rozpatrywanym punktem a punktem docelowym istnieje droga w linii prostej o maksymalnej dopuszczalnej prędkości. Klasą definiującą przebieg tej części algorytmu jest BestPathFinder, trzymający kolejne węzły do rozpatrzenia w kolejce priorytetowej.

### Zniechęcanie do wykonywania skrętów w lewo

Spełnienie dodatkowego kryterium minimalizacji skrętów w lewo ma zapewnić odpowiednia kara za wykonanie takiego skrętu. Realizowana jest ona poprzez dodanie odpowiedniej kary czasowej do przewidywanego czasu trwania pokonywania danej ścieżki, jeżeli takowy skręt miał miejsce. Wysokości kar są parametrami startowymi aplikacji i dodawane są na poziomie algorytmicznym, nie zaś na poziomie grafu. Wysokość kary różni się w zależności od względnej różnicy pomiędzy kategoriami drogi, z której następuje skręt oraz drogi, do której on następuje. Domyślnie wyższa kara towarzyszy skrętowi z drogi o niższym priorytecie w drodze o wyższej kategorii. Wyznaczaniem wysokości kar za skręt w lewo zajmuje się klasa LeftTurnHandler. Efektem zastosowania kar ma być wykonanie skrętu w lewo tylko wtedy, gdy jest to rzeczywiście konieczne lub gdy przyniesie to odpowiednio wysoki zysk w całkowitym oczekiwany czasie przejazdu.

### Wyznaczanie skrętów w lewo

Istotnym elementem algorytmu jest zdeterminowanie, czy dany element drogi jest skrętem w lewo, czy też nie. W tym celu zawsze badamy 3 kolejne węzły na danej drodze oraz 2 łączące je krawędzie. Czasami jedna ulica, mimo braku skrzyżowań, reprezentowana jest przez wiele węzłów. Z tego względu, jeżeli środkowy węzeł ma tylko 2 sąsiednie węzły to wiadomo, że na pewno nie jest on skrętem w lewo, a rozpatrywany fragment może być zwykłym zakrętem jednej drogi. Z drugiej strony często zdarza się, że 2 węzły są połączone dwiema różnymi krawędziami, gdyż każda z nich reprezentuje możliwość ruchu w jedną ze stron - wówczas takie zwracanie traktujemy jako skręt w lewo. W mniej skrajnych przypadkach sposób postępowania jest następujący - wyznaczamy wektory pomiędzy punktami pierwszym i środkowym oraz środkowym i trzecim, dokonując przekształcenia współrzędnych geograficznych na płaszczyznę 2D. Następnie korzystając ze współrzędnych powstały wektorów oraz ich norm wyznaczamy sinus i cosinus kąta pomiędzy nimi. Na ich podstawie, przy użyciu funkcji arctan wyznaczamy wartość kąta pomiędzy wektorami. Jeżeli wartość kąta w stopniach jest ujemna, jest to skręt w prawo. Jeżeli jest ona dodatnia i większa niż zadany parametr startowy aplikacji determinujący, od jakiego kąta klasyfikujemy skręty jako skręty w lewo (domyślnie 45 stopni), wówczas rozpatrywany skręt jest uznawany za skręt w lewo. Za rozpoznawanie skrętów w lewo odpowiada klasa LeftTurnHandler.

## Opis modelu danych

Do działania naszej aplikacji wykorzystujemy dane z OpenStreetMap, a dokładniej o dane strukturach *Node* i *Way*.

### Node

W *OpenStreetMap* Node (Węzeł) jest podstawowym sposobem reprezentacji punktu na mapie. Każdy node posiada numer ID oraz szerokość i długość geograficzne. Węzły mogą być używane do definiowania wolno stojących punktów, ale częściej są używane do definiowania kształtu linii.[1]

### Way

Way (Linia) reprezentuje zwykłe liniowy element na ziemi (taki jak droga, mur lub rzeka). Technicznie rzecz biorąc, linia jest uporządkowaną listą węzłów, która zwykle ma również co najmniej jeden znacznik (para klucz-wartość służąca do opisania elementów, np. maksymalnej dozwolonej prędkości na drodze) lub jest częścią relacji. Linia zazwyczaj ma od 2 do 2000 węzłów. Linia może być otwarta lub zamknięta.[2]

## Opis architektury aplikacji

Aplikację podzielono na 2 komponenty składowe, co umożliwiło lepszy podział obowiązków oraz rozbicie zadania na podproblemy. Główną częścią przygotowanej aplikacji jest Core (znajdujący się w katalogu *src*), czyli moduł udostępniający funkcjonalność znajdowania najlepszej ścieżki pomiędzy zadanymi przez użytkownika adresami. Za warstwę interakcji z użytkownikiem odpowiada Frontend, który to umożliwia mu podanie adresów w formie tekstowej, pomiędzy którymi należy znaleźć odpowiednią ścieżkę.

### Core

W celu uzyskania oczekiwanej funkcjonalności zaimplementowano szereg klas, mających na celu rozwiązywanie pewnych podproblemów. Wszystkie te klasy zostały opakowane w ostatecznie udostępnianą na frontend klasę App, oferującą możliwość znajdowania najkrótszej ścieżki dla zadanych adresów. Poniżej przedstawiony został zwięzły opis każdej z klas oraz motywacja stojąca za ich wyodrębnieniem.

#### 1. GraphProvider

Klasa ta ma za zadanie dostarczyć gotowy graf reprezentujący sieć drogową na starcie aplikacji. W zależności od parametrów wejściowych aplikacji graf ten może zostać zbudowany "od zera", tzn. z wykorzystaniem biblioteki pyrosm, jednak jest to proces dość czasochłonny - dla Warszawy i okolic zbudowanie takiego grafu trwa ok. 7 minut. Alternatywnym podejściem jest zatem zapisanie tak stworzonego grafu do pliku binarnego (w przypadku stworzonej aplikacji za pomocą modułu pickle), w celu wczytywania grafu przy kolejnych uruchomieniach aplikacji, co zajmuje znacznie mniej czasu, bo ok. 20 sekund. Efektem działania klasy jest dostarczenie gotowego grafu w postaci obiektu MultiDiGraph z biblioteki networkx, który można wykorzystać do znajdowania najkrótszej ścieżki.

#### 2. GeoMapper

Klasa ta ma za zadanie dokonać geomapowania, tzn. przekształcenia zadanych przez użytkownika adresów w formie tekstowej na współrzędne geograficzne. Wykorzystywana w tym celu jest biblioteka osmnx, która pod spodem korzysta ze wspomnianego w dokumentacji wstępnej Nominatim API. Dodatkowo, klasa ta umożliwia znalezienie dla punktu o zadanych współrzędnych geograficznych najbliższego węzła w zbudowanym wcześniej grafie.

#### 3. InputValidator

Klasa ta ma na celu walidację danych wprowadzonych przez użytkownika. Sprawdza ona, czy liczba podanych na starcie adresów nie przekracza maksymalnej dopuszczalnej wartości punktów. Ponadto upewnia się, czy każdy z zadanych przez użytkownika adresów mieści się w bbox stworzonego grafu.

#### 4. LeftTurnHandler

Klasa mająca na celu udostępnienie funkcjonalności rozpoznawania skrętów w lewo w grafie oraz wyznaczania odpowiedniej kary dla rozpoznanego skrętu. Zachowanie tej klasy zależy od parametrów wejściowych aplikacji, takich jak wysokość kary dla różnych rodzajów skrętów w lewo (na drogę o wyższej, równej lub niższej kategorii) oraz wartość progowa kąta pomiędzy wektorami, po przekroczeniu której skręt traktowany jest jako skręt w lewo.

#### 5. BestPathFinder

Klasa umożliwia funkcjonalność znajdowania najszybszej trasy przejazdu w grafie pomiędzy dwoma zadanymi węzłami. Wykorzystywany w tym celu jest algorytm A\* z opisaną wcześniej heurystyką. Heurystykę można jednak modyfikować poprzez podanie w parametrze aplikacji wartości dopuszczalnej prędkości jazdy na teoretycznej drodze prowadzącej po linii prostej z badanego węzła do celu.

#### 6. TravelSalesmanSolver

Klasa ta ma za zadanie rozwiązanie problemu komiwojażera dla zadanego przez użytkownika adresów. W celu uproszczenia problemu wykorzystywany jest algorytm aproksymacyjny Nearest Neighbor. Wielokrotnie korzysta z funkcjonalności oferowanej przez BestPathFindera i znajduje najszybsze trasy pomiędzy kolejnymi punktami w wyznaczonej przez siebie kolejności. Efektem działania tej klasy jest zwrócenie znalezionej ścieżki.

#### 7. App

Klasa definiująca działanie głównej funkcjonalności naszej aplikacji. Określa ona kolejność wykonywanych operacji, upewnia się, że wszystkie elementy składowe odpowiednio będą ze sobą współpracować oraz że każdy z nich otrzyma dane w wymaganej przez siebie postaci. Udostępnia na frontend prosty interfejs umożliwiający znalezienie najlepszej ścieżki między wieloma zadanymi adresami. By korzystać z aplikacji, obiekt tej klasy musi zostać zainicjalizowany z odpowiednimi parametrami (którym przyporządkowano wartości domyślne), a następnie musi zostać na nim wywołana metoda inicjalizująca jego stan na potrzeby obsługi przyszłych przychodzących zapytań.

#### 8. GraphUtils

Nie jest to wprawdzie klasa sama w sobie, lecz w pliku tym zdefiniowano wszelkie potrzebne metody pomocnicze wykorzystywane w innych miejscach. Zdefiniowana tutaj funkcjonalność to m.in. obliczanie wartości heurystyki, obliczanie odległości między dwoma punktami na podstawie ich współrzędnych geograficznych, porównywanie kategorii dróg na podstawie ich zdefiniowanego liniowego porządku, obliczanie sinusa, cosinusa oraz kąta pomiędzy dwoma wektorami, wyznaczanie wektora pomiędzy dwoma punktami na podstawie ich współrzędnych geograficznych, obliczanie bbox dla zbudowanego grafu reprezentującego sieć drogową, uzupełnianie brakujących wartości maksymalnej dopuszczalnej prędkości jazdy na podstawie kategorii drogi czy czyszczenie zbioru danych przed zbudowaniem grafu.

### Frontend

Aplikacja składa się z dwóch widoków:

- widok domyślny, który umożliwia dodanie adresu
- widok *success*, który umożliwia uruchomienie algorytmu odnajdywania najszybszej drogi, a także usuwanie wcześniej dodanych adresów

## Enter an Address

Text:

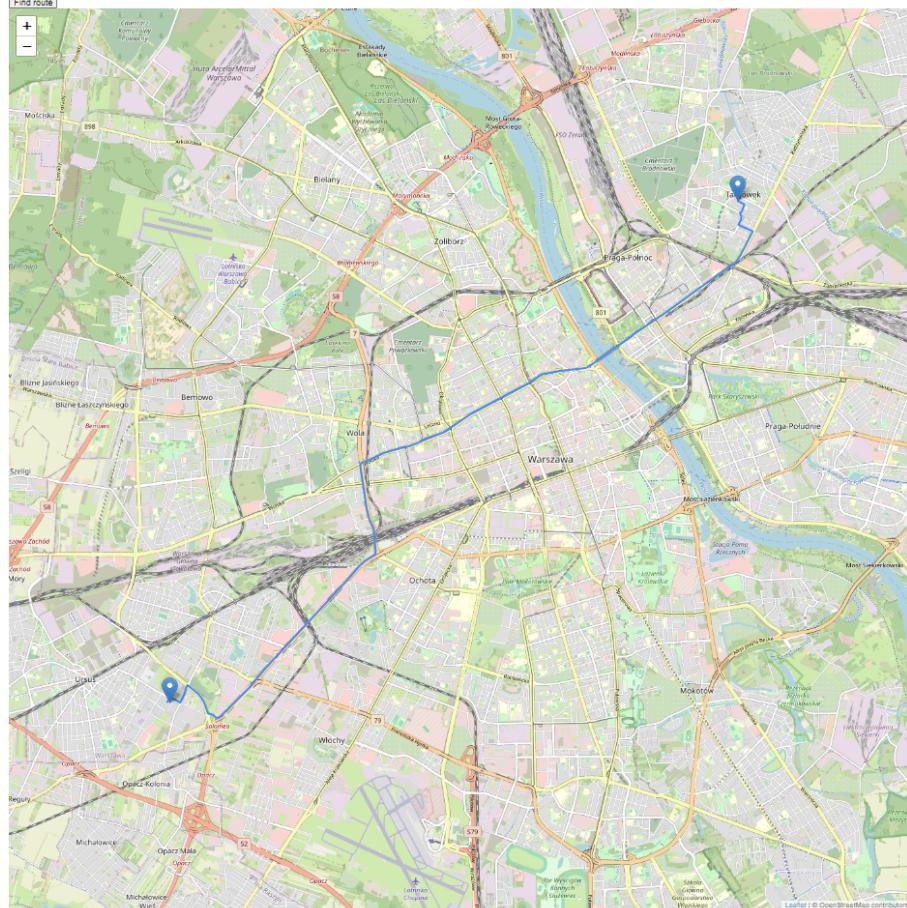
Rys. 1. Widok domyślny aplikacji

### Entered addresses

- Mokra 33, Warszawa [Delete](#)
- Danei Warszawy 23b, Warszawa [Delete](#)

[Add another address](#)

[Find route](#)



Rys. 2. Widok *success*

W widoku domyślnym dodawane są adresy, które są następnie widoczne w widoku *success*. W momencie kliknięcia przycisku **Find route**, aplikacja wysyła adresy do Core, który przetwarza dane, a następnie zwraca listę numerów id, należących do node'ów, które znajdują się na odnalezionej ścieżce. Następnie te dane są tłumaczone na listę współrzędnych geograficznych, za pomocą metod znajdujących się w skrypcie services.py. Ta lista jest przekazywana do skryptu odpowiedzialnego za wygląd mapy znajdującej się w widoku *success*. Javascript następnie generuje linie, która leży na obliczonej trasie.

## Parametry startowe aplikacji

Sposób działania zaimplementowanej aplikacji zależy od wielu podawanych parametrów startowych. Poniżej zamieszczony został opis każdego z parametrów.

1. `read_graph_from_pickle` (domyślna wartość: False)  
Flaga boolowska określająca, czy graf ma zostać wczytany z pliku binarnego .pkl (True), czy nie (False)
2. `pickle_filepath` (domyślna wartość: "")  
Ścieżka do pliku .pkl zawierającego zapisany wcześniej graf
3. `region` (domyślna wartość: "Warsaw")  
Region geograficzny, dla którego budujemy graf - w naszym zastosowaniu 'Warsaw'
4. `max_points_allowed` (domyślna wartość: 7)  
Maksymalna dopuszczalna liczba punktów, pomiędzy którymi użytkownik może wyznaczyć ścieżkę
5. `min_angle_left_turn` (domyślna wartość: 45.0)  
Minimalny kąt w stopniach, od którego zakręt klasyfikowany jest jako skręt w lewo
6. `penalty_to_better_road` (domyślna wartość: 30.0)  
Dodatkowa kara w sekundach za skręt w lewo w drogę o wyższej kategorii
7. `penalty_to_equal_road` (domyślna wartość: 20.0)  
Dodatkowa kara w sekundach za skręt w lewo w drogę o takiej samej kategorii
8. `penalty_to_worse_road` (domyślna wartość: 10.0)  
Dodatkowa kara w sekundach za skręt w lewo w drogę o niższej kategorii
9. `heur_maxspeed` (domyślna wartość: 140.0)  
Maksymalna dopuszczalna prędkość jazdy na teoretycznej drodze na potrzeby heurystyki

## Wykorzystywane biblioteki

Poniżej prezentujemy krótki opis zewnętrznych bibliotek, z których korzystano przy implementacji aplikacji z podziałem na Core i frontend.

### Core

1. `pyrosm`  
Umożliwia wczytywanie danych z plików .osm.pbf do obiektów GeoDataFrame z biblioteki geopandas. Wykorzystano ją do pobrania danych z OSM dla badanego regionu (Warszawa) oraz przetworzenia surowych danych na sieć drogową reprezentowaną przez graf.
2. `networkx`  
Umożliwia wydajną pracę ze strukturami grafowymi. Obiekt klasy MultiDiGraph z tej biblioteki został wykorzystany do reprezentowania sieci drogowej, w której następowało wyszukiwanie najlepszej trasy.

### 3. osmnx

Oferuje szereg funkcjonalności usprawniających pracę z danymi z OpenStreetMap. Wykorzystana została głównie do geomapowania adresów na współrzędne geograficzne oraz znajdowania najbliższego węzła w grafie dla punktu o zadanych współrzędnych geograficznych.

### 4. geopandas

Ułatwia pracę z surowymi danymi przestrzennymi. Została wykorzystana do wstępnego czyszczenia danych przed utworzeniem grafu oraz wyliczeń oczekiwanej czasu przejazdu dla poszczególnych fragmentów dróg.

### 5. numpy

Wykorzystano ją do wykonywania obliczeń numerycznych, szczególnie przy wyznaczaniu skrętów w lewo oraz odległości geograficznej pomiędzy punktami o zadanych współrzędnych.

## Frontend

Postanowiono, że serwis zostanie zrealizowany w formie aplikacji webowej. W tym celu wykorzystano framework Django oraz języka Javascript i biblioteki Leaflet.

Django to framework webowy oparty na języku Python, który ułatwia tworzenie aplikacji internetowych. Biblioteka Django zapewnia strukturę dla budowania aplikacji, oferując narzędzia do zarządzania bazami danych, routingu URL, autoryzacji użytkowników oraz generowania widoków i szablonów.

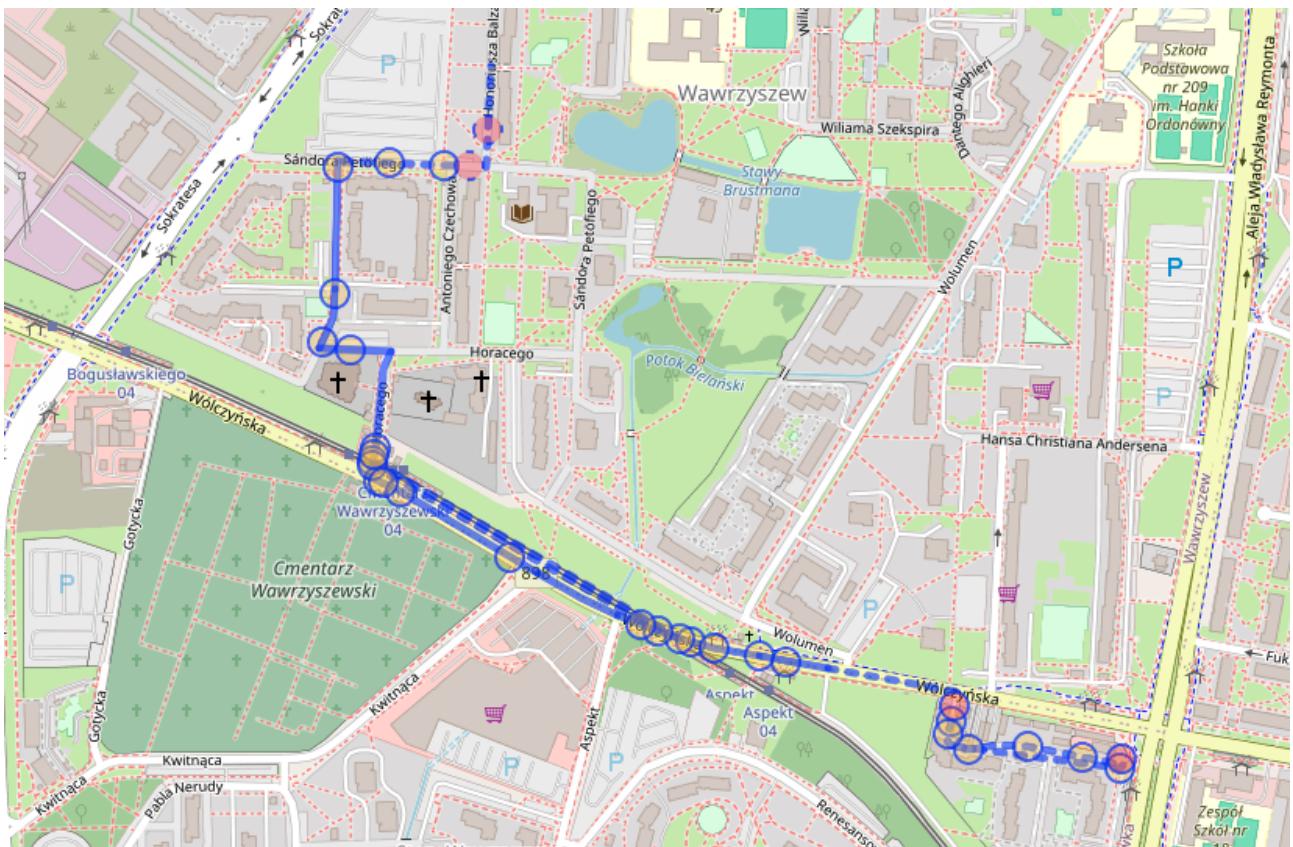
Leaflet to lekka, otwarta biblioteka JavaScript służąca do tworzenia interaktywnych map internetowych. Umożliwia łatwe osadzanie map na stronach internetowych, oferując funkcje takie jak zoomowanie, przesuwanie, dodawanie markerów oraz warstw zewnętrznych danych. Leaflet integruje się z różnymi źródłami danych mapowych, takimi jak OpenStreetMap.

## Opis i wyniki testów

### Wpływ wysokości kary za skręt w lewo na kształt wyznaczanej trasy

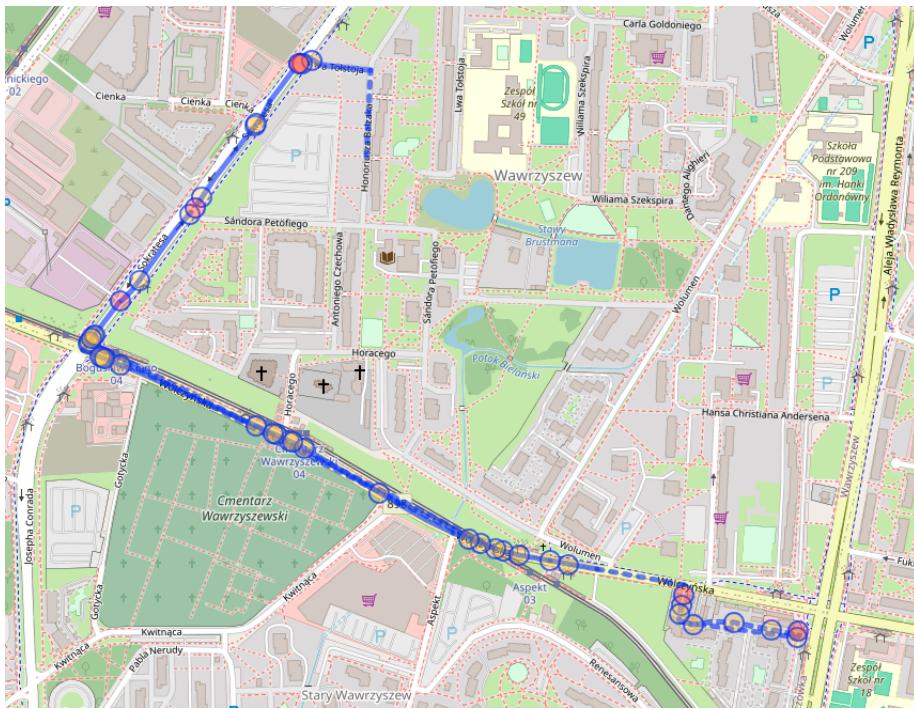
W celu sprawdzenia poprawności naliczania kary za skręty w lewo przeprowadzony został prosty eksperyment - wyznaczono najlepszą trasę pomiędzy dwoma punktami - startowym "Wólczyńska 3, Warszawa" oraz docelowym "Balzaka 2, Warszawa" - w wersji bez naliczania kary, z jej stosunkowo niską wartością (20 s) oraz z jej bardzo wysoką wartością (240 s).

Na rysunku 1 przedstawiono znalezioną trasę bez żadnej kary za skręt w lewo. Wyznaczona trasa prowadzi prosto do celu, a po drodze trzykrotnie wykonywany jest skręt w lewo - pierwszy przy wejściu na ul. Wólczyńską, drugi za kościołem na ul. Horacego oraz trzeci pod koniec trasy z ul. Petőfiego w ul. Balzaka. Jak obrazuje zatem ten przykład, przy braku kary za skręt w lewo algorytm nie unika ich stosowania.



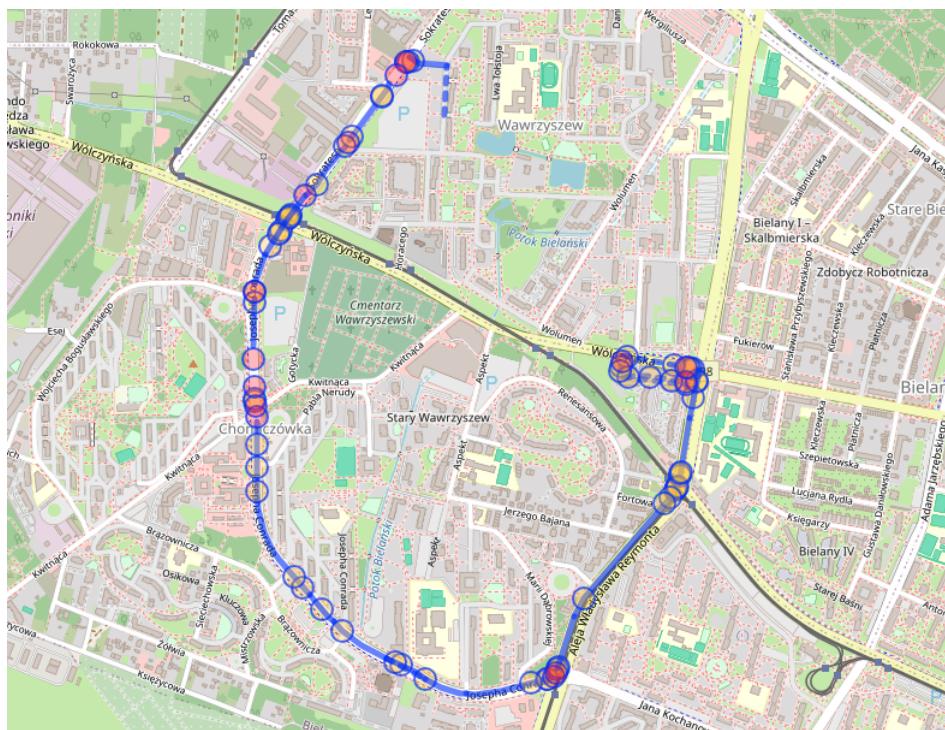
Rys. 3. Wyznaczona trasa bez kary za skręt w lewo

Na rysunku 2 przedstawiono znalezioną trasę ze średnią wartością kary za skręt w lewo (20 s). W odróżnieniu od poprzedniej znalezionej trasy, chcąc uniknąć dwóch skrętów w lewo, algorytm nie wybiera skrętu w prawo w ul. Horacego, lecz kontynuuje jazdę ul. Wólczyńską, aż do skrętu w prawo w ul. Sokratesa. Wybrana wartość parametru kary jest zatem wystarczająco wysoka, by uniknąć dwóch skrętów, lecz równocześnie wystarczająco niska, by zdecydować się na skręt w lewo w ul. Wólczyńską na początku trasy.



Rys. 4. Wyznaczona trasa z niską karą za skręt w lewo

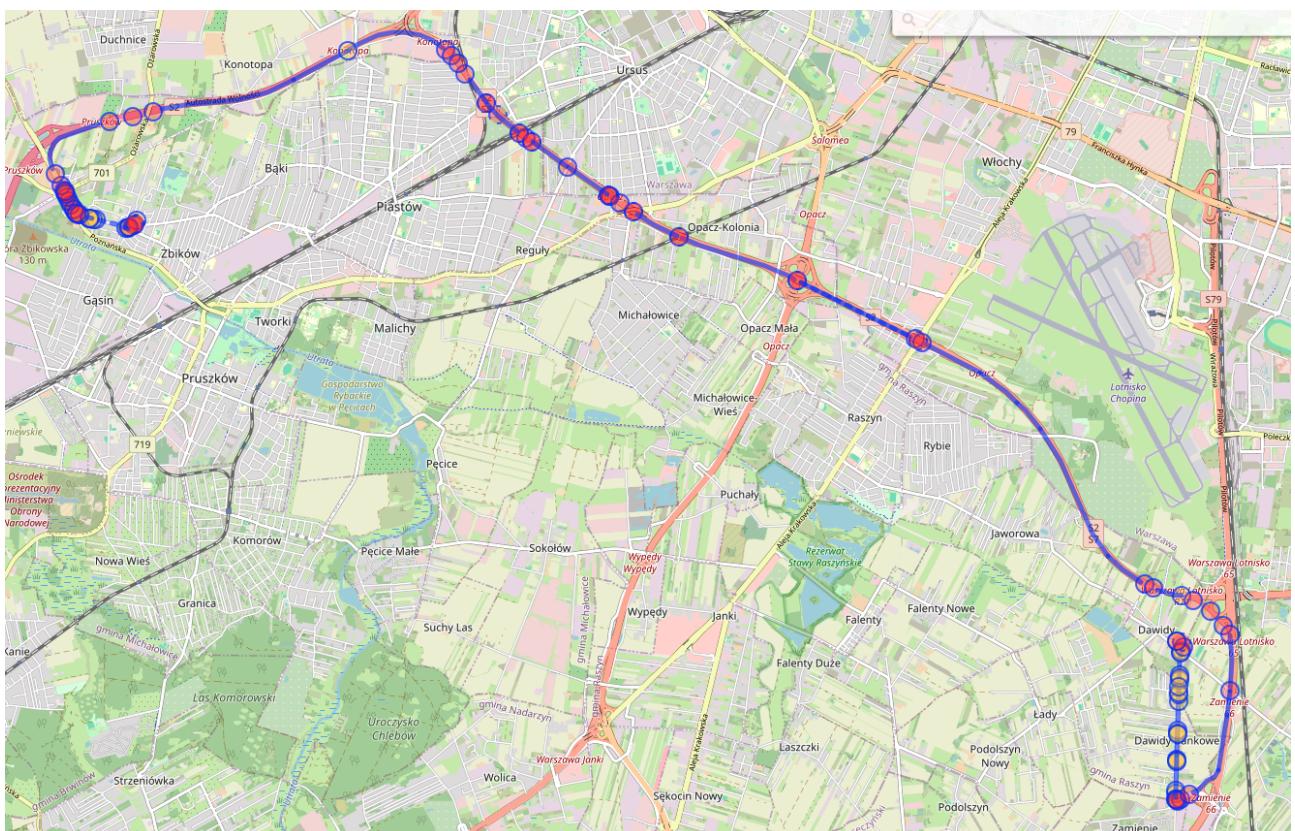
Na rysunku 3 przedstawiono znalezioną trasę z wysoką wartością kary za skręt w lewo (240 s). Wartość ta okazała się na tyle wysoka, że algorytm unika również skrętu w lewo w ul. Wólczyńską i preferuje drogę prowadzącą bardzo dookoła (przez al. Reymonta i ul. Conrada). Wyznaczona przy wysokiej wartości parametru kary trasa całkowicie eliminuje skręty w lewo dla zadanych punktów startowego i docelowego.



Rys. 5. Wyznaczona trasa z wysoką karą za skręt w lewo

## Dopuszczalność stosowanej heurystyki w algorytmie A\*

Istotnym elementem zapewnienia poprawności rozwiązania w wyszukiwaniu najszybszej trasy między dwoma zadanymi punktami jest dopuszczalność wykorzystywanej heurystyki. W celu zobrazowania dopuszczalności wyznaczono trasę pomiędzy punktem startowym "Granitowa 9, Żbików" a punktem docelowym "Śliska 9, Dawidy". Algorytm nie zwraca ścieżki najkrótszej pod względem odległości, lecz estymowanego czasu przejazdu - jest on gotowy eksplorować drogę w przeciwnym kierunku geograficznym (w poniższym przykładzie na północy zachód) w nadzieję na znalezienie drogi o wyższej maksymalnej dopuszczalnej prędkości jazdy. Znalezioną najszybszą trasę między dwoma powyżej zadanymi punktami, wiodącą przez drogę ekspresową S2, obrazuje rysunek 4.

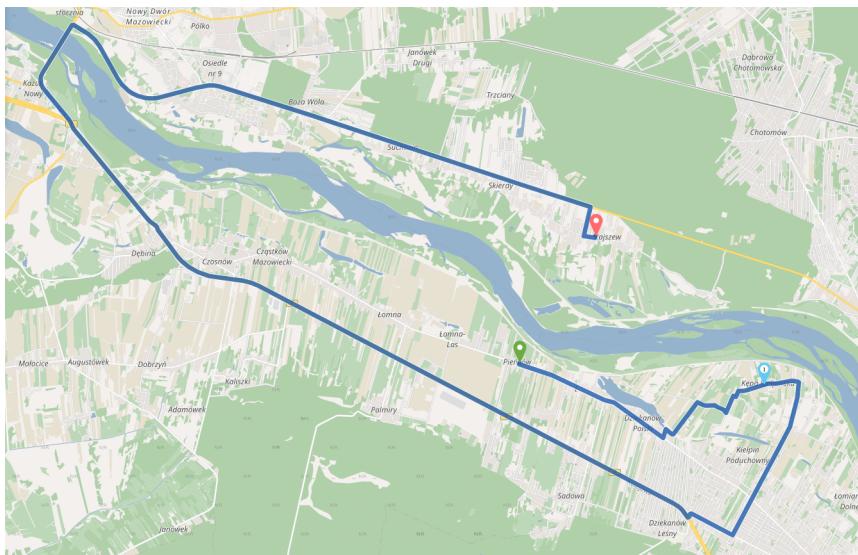


Rys. 6. Poprawnie działająca heurystyka algorytmu A\*

## Optymalność znalezionej trasy - przypadek pesymistyczny

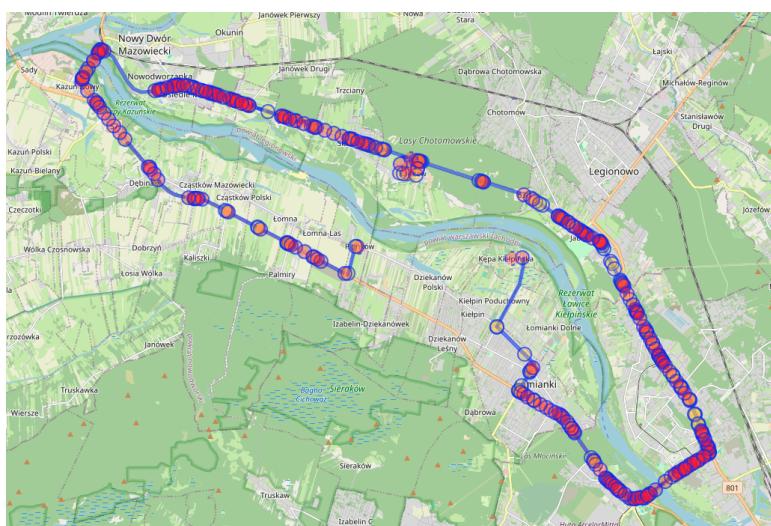
Niestety, wykorzystywany algorytm aproksymacyjny Nearest Neighbor zwraca trasę daleką od optymalnej dla złośliwie spreparowanych danych. Przypadek taki zilustrowany może być na przykładzie problemu wyznaczenia najkrótszej ścieżki pomiędzy trzema punktami, gdzie zadanym punktem startowym jest "Pieńków 34", a pozostałe dwa punkty do odwiedzenia to "6 Pułku Piechoty, Kępa Kiełpińska" oraz "Rajszew".

Na rysunku 5 przedstawiono optymalną trasę wyznaczoną przez narzędzie GraphHopper. Jak widać, z punktu startowego (zielony znacznik) należy udać się najpierw do Kępy Kiełpińskiej (niebieski znacznik), a dopiero później na drugą stronę Wisły do Rajszewa (czerwony znacznik).



Rys. 7. Optymalna trasa wyznaczona przez narzędzie GraphHopper

Na rysunku 6 przedstawiono daleką od optymalnej trasę zwracaną przez wykorzystywany w aplikacji algorytm Nearest Neighbor. Ze względu na fakt, iż Rajszew znajduje się w bliższej odległości w linii prostej od punktu startowego w Pieńkowie, algorytm uda się w pierwszej kolejności właśnie tam, a dopiero później do Kępy Kiełpińskiej. Skutkuje to trasą daleką od optymalnej, która dwukrotnie pokonuje Wisłę. Algorytm Nearest Neighbor ma zatem dużą szansę na zwrócenie nieoptymalnej trasy, gdy pomiędzy punktami znajdują się pewne przeszkody (np. rzeka).

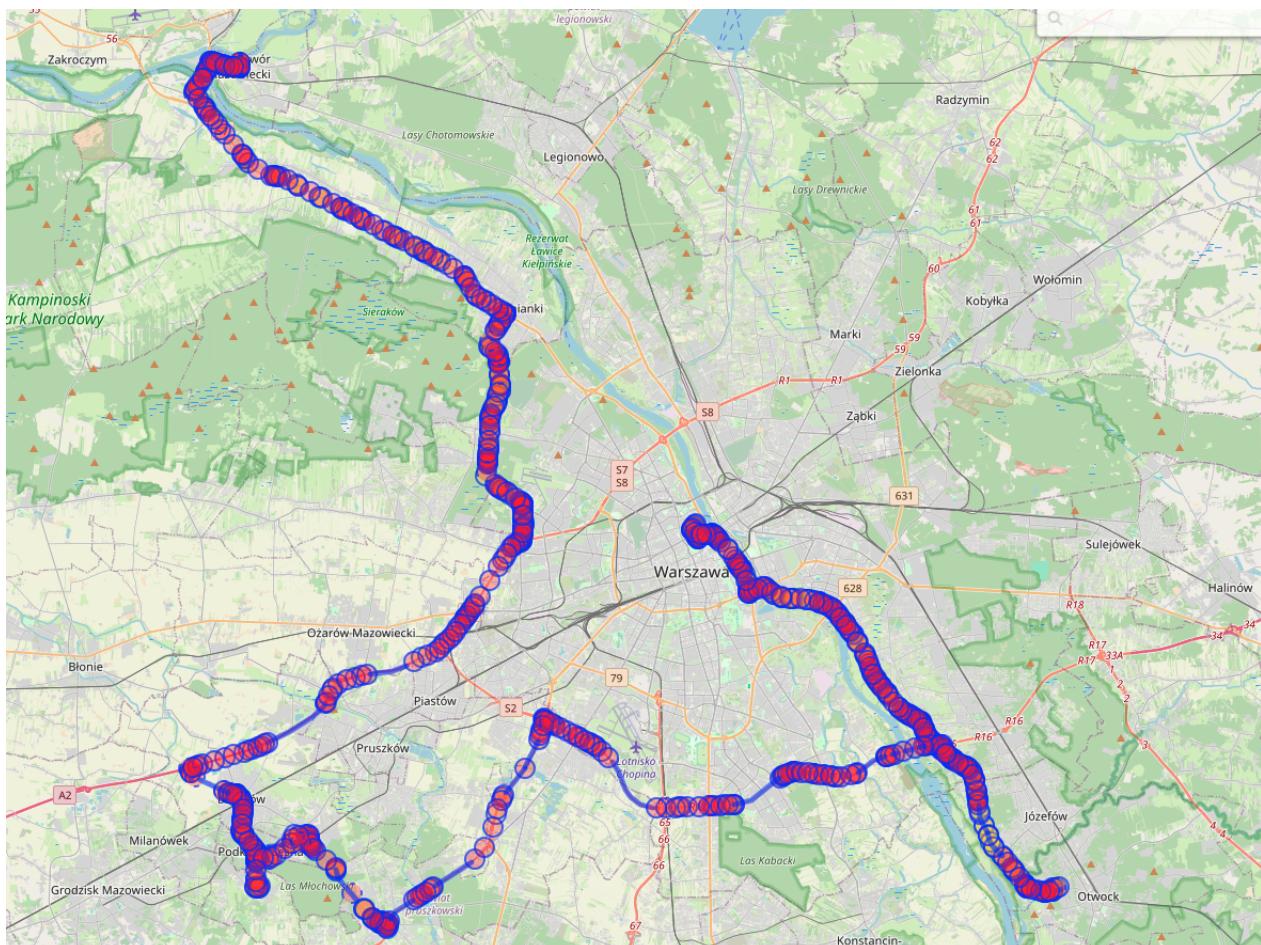


Rys. 8. Nieoptymalna trasa zwracana przez algorytm Nearest Neighbor

## Porównanie algorytmu Nearest Neighbor oraz brutalnego

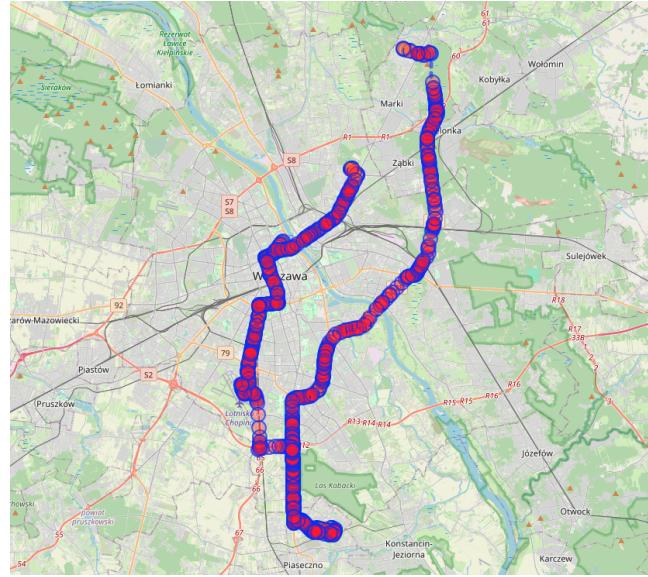
W celu porównania wyników uzyskiwanych przez algorytm Nearest Neighbor oraz brutalny przeprowadzone zostały dwa eksperymenty. W pierwszym z nich wyznaczono trasę pomiędzy czterema zadanymi punktami, możliwe rozrzuconymi po mapie Warszawy i okolic - były to: "Freta 10, Warszawa" jako punkt startowy, "Klonowa 6, Żółwin", "Podleśna 14, Otwock" oraz "Lotników 18, Nowy Dwór Mazowiecki" jako punkty do odwiedzenia. W drugim eksperymencie złożliwie wybrano 5 punktów w taki sposób, aby zademonstrować różnicę w jakości zwracanego rozwiązania - te punkty to: "Samotna 4, Warszawa" jako punkt startowy, "Żwirki i Wigury 1, Warszawa", "Kopciuszka 5, Julianów", "Freta 10, Warszawa" oraz "Tęczowa 4, Marki" jako punkty do odwiedzenia. Poniżej prezentujemy wyniki obu eksperymentów. W przypadku algorytmu brutalnego zastosowano przycinanie, tzn. gdy całkowity oczekiwany czas obecnie wyznaczanej trasy przekroczył czas dotychczas najlepszej znalezionej trasy, jej dalsze wyznaczanie było przerywane. By skrócić czas działania algorytmu brutalnego w drugim eksperymencie, jako początkowy najlepszy dotychczas znaleziony czas podano przewidywany czas trasy zwróconej uprzednio przez algorytm NN.

W pierwszym eksperymencie z 4 zadanymi punktami zarówno algorytm NN, jak i brutalny zwróciły tę samą najlepszą znalezioną trasę, która została zilustrowana na rysunku 7. Przewidywany czas pokonania tej trasy był naturalnie w obu przypadkach jednakowy i wyniósł 1 godz 57 min i 6 sek. Czas pracy algorytmu wyniósł 2 min 8 sek w przypadku algorytmu NN oraz 16 min 15 sek w przypadku algorytmu brutalnego. Biorąc pod uwagę fakt, że zwrócono jednakowe trasy, a algorytm NN wykonał to 8 razy szybciej niż brutalny, można uznać, że użycie algorytmu aproksymacyjnego dla zadanego przykładu okazało się zasadne.



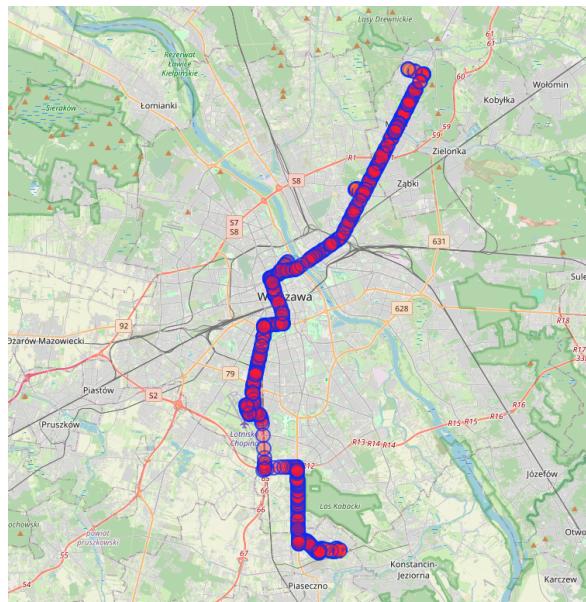
Rys. 9. Wyniki eksperymentu dla 4 zadanych punktów

W drugim eksperymencie najlepszą trasę zwróconą przez algorytm Nearest Neighbor przedstawiono na rysunku 8. Przewidywany czas przejazdu trasy wyniósł 1 godz 18 min i 40 sek, a czas pracy algorytmu to 1 min 59 sek. Zwrócona trasa na pierwszy rzut oka wydaje się nieoptimalna - znacznie lepiej byłoby najpierw odwiedzić Tęczową 4 w Markach (punkt położony najdalej na północny wschód), a dopiero resztę punktów. Algorytm NN kierując się wyznaczeniem drogi do najbliższego nieodwiedzonego sąsiada, decyduje się wpierw odwiedzić wszystkie punkty na południu, a do Marek pojechać na koniec.



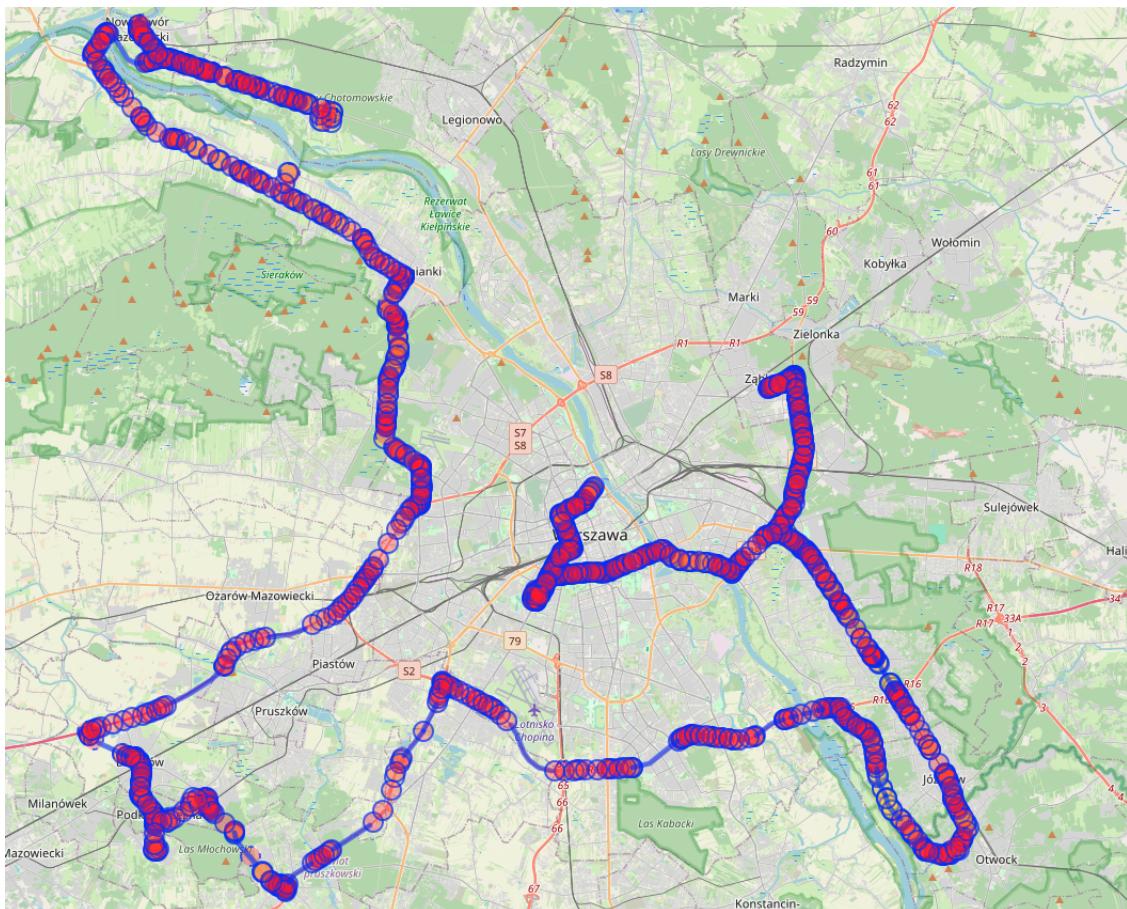
Rys. 10. Wyniki algorytmu NN dla 5 zadanych punktów

W przypadku algorytmu brutalnego z przycinaniem znaleziona zostaje optymalna trasa, czyli Marki odwiedzane są w pierwszej kolejności, a dopiero potem wszystkie kolejne punkty na południu. Estymowany czas pokonania zwróconej trasy, przedstawionej na rysunku 9, wyniósł 1 godz 2 min i 37 sek, czyli o ok. 16 minut mniej niż w przypadku algorytmu NN. Czas pracy algorytmu brutalnego wyniósł jednak 48 min 23 sek, czyli 24 razy więcej niż w przypadku NN. Oznacza to, że przycinanie w powyższym przykładzie nie przyniosło zadowalających rezultatów, gdyż 24 to właśnie liczba permutacji kolejności punktów do odwiedzenia (4!).



Rys. 11. Wyniki algorytmu brutalnego dla 5 zadanych punktów

Podjęta została również próba przeprowadzenia eksperymentu z 8 zadanymi punktami. Do czterech punktów z eksperymentu pierwszego dodano 4 kolejne punkty do odwiedzenia: "Opaczewska 13, Warszawa", "Spacerowa 7, Ząbki", "Pieńków 34" i "Fiolkowa 2, Rajszew". W próbie tej algorytm NN znalazł ścieżkę przedstawioną na rysunku 10. Wyznaczona kolejność odwiedzania punktów jest następująca: "Freta 10, Warszawa", "Opaczewska 13, Warszawa", "Spacerowa 7, Ząbki", "Podleśna 14, Otwock", "Klonowa 6, Żółwin", "Pieńków 34", "Fiolkowa 2, Rajszew" i na końcu "Lotników 18, Nowy Dwór Mazowiecki". Oczekiwany czas przejazdu wyznaczonej trasy wyniósł 2 godz 48 min 57 sek, a czas pracy algorytmu NN to 5 min 17 sek. Na pierwszy rzut oka widać, że nie jest to rozwiązanie optymalne - lepsze można by uzyskać chociażby przez zamianę kolejności odwiedzenia dwóch ostatnich punktów. Niemniej jednak rozwiązanie wydaje się być racjonalne, a czas jego uzyskania - akceptowalny. Warto jednak zwrócić uwagę, iż podany przykład nie został dobrany z nadmierną złośliwością. W przypadku algorytmu brutalnego, nie udało się uzyskać rezultatów - po ok. 6 godzinach pracy algorytmu przerwano jego działanie.



## Bibliografia

- [1] *Węzel*, [Online; accessed 30-May-2024], 2024. [Online]. Available: <https://wiki.openstreetmap.org/wiki/Pl>.
- [2] *Linia*, [Online; accessed 30-May-2024], 2024. [Online]. Available: <https://wiki.openstreetmap.org/wiki/Pl:Linia>.