AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA TELEKOMUNIKACJI

Praca dyplomowa magisterska

Opracowanie algorytmu wyszukiwania tras dla rowerzystów na podstawie heterogenicznych zbiorów danych geo-przestrzennych

Development of the search algorithm of routes for cyclists on the basis of heterogeneous geo-spatial data sets.

Autor:	*Paweł Milota, Jan Posz*
Kierunek studiów:	*Elektronika i Telekomunikacja*
Opiekun pracy:	*dr hab. inż Mikołaj Leszczuk*

Kraków, 2019

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Serdecznie dziękujemy dr hab. Mikołajowi Leszczukowi za nadzór i pomoc podczas tworzenia pracy.

Spis treści

1. Cel Pracy

Celem pracy było opracowanie algorytmu wyszukiwania optymalnych tras rowerowych w Krakowie na podstawie bazy danych ulic udostępnianej przez serwis http://rowery.zikit.pl. Algorytm ma na celu wyznaczenie najkrótszej trasy przebiegającej po istniejących ścieżkach rowerowych, a w przypadku ich braku, ulicach o możliwie najniższym ograniczeniu prędkości. Opierając się na uzyskanych wynikach została opracowana aplikacja mobilne, umożliwiająca pobranie trasy pomiędzy dwoma wprowadzonymi adresami oraz nawigowanie użytkowników podczas poruszania się do celu, a także aplikacja internetowa umożliwiająca pobranie trasy i zaznaczenie jej przebiegu na mapie. W celu weryfikacji działania algorytmu zostały przeprowadzone badania mające na celu porównanie opracowanego rozwiązania z istniejącym oferowanym przez Google Maps. Zostały także przeprowadzone badania w celu weryfikacji optymalnego dla tego zastosowania algorytmu wyszukiwania najkrótszych ścieżek w grafie.

2. Wstęp

3. Nawigacje dla rowerzystów

3.1. Specyfika tematu

3.2. Istniejące rozwiązania

3.3. Aplikacje webowe

3.4. Aplikacje mobilne

4. Charakterystyka środowisk

4.1. Urządzenia mobilne z systemem iOS

4.2. Środowisko webowe

4.2.1. Przeglądarki web na popularne systemy operacyjne

4.3. Dane geoprzestrzenne

4.3.1. Dane urzędu miasta krakowa na temat ścieżek rowerowych (CartoDB)

4.3.2. Dane społeczności rowerowej (DaroPlan)

4.3.3. OpenStreetMap oraz Apple Maps

4.4. Użyte algorytmy

4.4.1. Algorytm Dijkstry

4.4.2. Algorytm A*

4.4.3. Metoda Haversine

4.5. Wykorzystane języki programowania

4.5.1. Język programowania JavaScript

4.5.2. Biblioteka ReactJS

4.5.3. Język programowania Swift

4.6. Użyte narzędzia

4.6.1. Środowisko programistyczne Xcode

4.6.2. Narzędzie do testowania API Postman

4.6.3. System kontroli wersji Git

5. Tworzenie oprogramowania

5.1. Opis użytej technologii i bibliotek

W celu implementacji aplikacji serwerowej użyto biblioteki Node.js, jest to stosunkowo nowa technologia, jej pierwsza wersja ujrzała światło dzienne w roku 2009. Node.js umożliwia wykonywanie kodu Javascript poza przeglądarką użytkownika, po stronie serwerowej, składa się z silnika V8 zaimplementowanego przez firmę Google oraz zestawu bibliotek dołączonych przez twórcę języka w celu stworzenia wieloplatformowego środowiska do uruchamiania skryptów napisanych w kodzie Javascript. Biblioteka Node.js jest zaimplementowana w oparciu o architekturę event-driven i asynchroniczne operacje wyjścia i wejścia, oznacza to że biblioteka wydajnie wykorzystuje wielowątkowość i nie zostaje zablokowana wykonywaniem długich operacji przez równolegle działające procesy. Zważając na opisane wcześniej charakterystyki, aplikacje stworzone przy użyciu Node.js wyróżniają się dużą wydajnością oraz są optymalnym wyborem dla aplikacji oczekujących szybkiego przetwarzania lub komunikacji w czasie rzeczywistym. Dodatkową zaletą biblioteki Node.js jest fakt że wykorzystuje ten sam język programowania który używa się do programowania stron internetowych, z racji tego bariera wejścia do tworzenia pełnych systemów jest znacznie zmniejszona. Aplikacja serwerowa korzysta z wielu pomocniczych bibliotek które umożliwiają szybsze tworzenie kodu. Standardowym oprogramowaniem służącym do zarządzania zewnętrznymi bibliotekami jest aplikacja npm, podczas tworzenia użyto jednak aplikacji yarn która działa w bardzo przybliżony sposób, zabezpiecza jednak przed usunięciem przez twórcę oprogramowania kodu źródłowego każdej z bibliotek. W poniższych punktach opisano kilka wybranych bibliotek użytych do stworzenia aplikacji wraz z krótkim opisem i przykładem użycia:

Express - biblioteka służąca do pisania aplikacji serwerowych przy użyciu biblioteki Node.js. Udostępnia prosty interfejs do tworzenia punktów końcowych i jest określony jako jeden ze standardów podczas tworzenia aplikacji serwerowych przy użyciu Node. Eslint - biblioteka służąca jako tzw. linter , czyli proces na bierząco sprawdzający jakość kodu pisanego przez użytkownika. Dzięki użyciu eslint programista nie musi martwić się o niejednorodność we wcięciach, nieużywane zmienne czy niepoprawne ilości znaków białych, jest to sprawdzane przez program. Eslint korzysta z pliku .eslintrc który definiuje wszystkie zasady które mają być spełnione w projekcie aby kod został uznany za bezbłędny. Axios - biblioteka umożliwiająca proste interakcje z interfejsami serwerowymi innych serwisów. Dzięki stworzeniu globalnego klienta axios zawierającego podstawową konfigurację nie ma potrzeby powielać tych samych konfiguracji w wielu zapytaniach, wystarczy jedynie zdefiniować rzeczy które się różnią jak

na przykład URL czy metoda HTTP. Cors - w celu komunikacji aplikacji serwerowej i strony internetowej znajdujących się pod jednym adresem IP należy skonfigurować „Cross Origin Resource Sharing". Ma to zapobiec możliwości przesyłania złośliwych skryptów w formularzach które zostaną wykonane przez aplikację. Do umożliwienia Cross Origin Resource Sharing użyto biblioteki cors która po inicjalizacji automatycznie dodaje nagłówek HTTP Access-Control-Allow-Origin z odpowiednią domeną dla której tego typu zapytania mają być możliwe.

Aplikacja w celu wyznaczania tras korzysta z dwóch zewnętrznych serwisów - ZiKIT Carto oraz openstreetmap. Carto jest to platforma do zarządzania danymi geoprzestrzennymi która umożliwia pobranie, wizualizację oraz przeszukiwanie dróg przy użyciu odpowiednich komend SQL. Kod carto jest otwarty i każdy może uruchomić go w swojej stronie internetowej w celu uproszczonej wizualizacji tras na mapie. Krakowski ZiKIT korzysta z własnej instancji carto w celu wizualizacji i przetrzymywania wszelkiego rodzaju danych jak zbiory różnych typów dróg oraz wydarzenia(na przykład wypadki samochodowe) na mapach. Opensteetmap jest to darmowy portal udostępniający usługę map oraz oferuje szeroki zakres innych zakresów danych geoprzestrzennych. Openstreetmap jest projektem otwartym, portal jest jest utrzymywany przez fundację o tej samej nazwie a swoje dane zawdzięcza użytkownikom którzy wprowadzają je w celu ich późniejszego wykorzystania lub jako wolontariusze.

5.1.1. Opis udostępnionych endpointów

W celu umożliwienia komunikacji aplikacji mobilnej oraz strony internetowej, przygotowana aplikacja serwerowa udostępnia pod zdefiniowanymi adresami URL end-point'y które po przekazaniu odpowiednich parametrów w kwerendzie umożliwia pobranie danych. Zaimplementowano je przy pomocy obiektu router, będącego częścią opisanej wcześniej biblioteki express. Poniżej przedstawiono dwa end-point'y udostępnione przez aplikację razem z dokumentacją oraz przykładową, skróconą, odpowiedzią w formacie JSON:

1. Endpoint findOptimized

   Używany przez aplikację mobilną w celu pobrania optymalnej trasy pomiędzy dwoma punktami przekazanymi w formie kwerendy. Wykorzystuję metodę HTTP GET. Możliwe parametry do przekazania w kwerendzie URL:

   - startLocation - parametr określający punkt początkowy wyznaczonej trasy w przypadku gdy jest on przekazany jako adres.
   - startLocationLatitude oraz startLocationLongitude - przez te parametry zostaje przekazana pozycja użytkownika w przypadku gdy zostaje ona zdefiniowana przez współrzędne geograficzne punktu początkowego. W przypadku gdy lokalizacja punktu początkowego zostaje przekazana w tej postaci, parametr startLocation zostaje zignorowany, niezależnie od tego czy został zdefiniowany w kwerendzie.

- endLocation - parametr określający punkt końcowy trasy w postaci adresu. Aplikacja nie daje możliwości przekazania punktu końcowego trasy w postaci współrzędnych geograficznych, dlatego też w przypadku punktu końcowego trasy przekazanie lokalizacji w tej postaci nie zostało zaimplementowane.
- routeType - określenie czy wyznaczona trasa ma być najkrótsza, czy optymalna z wagowego punktu widzenia. Przyjmuje parametr typu string o wartość BEST lub SHORTEST.

Przykładowe zapytanie HTTP GET w celu uzyskania odpowiedzi zawierającej optymalną trasę przejazdu można uzyskać pod adresem URL:

bazowy_adres_url/api/routes/findOptimized?startLocation=romera&endLocation=karmelicka&routeType=BEST

Przykładową odpowiedź, skróconą do postaci przedstawiającej jej formę przedstawiono poniżej.

// tutaj idzie przeklejona odpowiedź API, najlepiej w postacie screen-shot'a żeby było formatowanie

2. Endpoint visualizationPoints:

   Używany przez stronę internetową będącą częścią przygotowanego systemu w celu wizualizacji trasy na mapach openstreetmap. Wykorzystuje metodę HTTP GET. Możliwe parametry do przekazania w kwerendzie URL:

   - startLocation - parametr określający początkową lokalizację użytkownika w postaci adresu.
   - endLocation - parametr określający końcową lokalizację użytkownika w postaci adresu.
   - routeType - określenie czy wyznaczona trasa ma być najkrótsza, czy optymalna z wagowego punktu widzenia. Przyjmuje parametr typu string o wartości BEST lub SHORTEST.
   - algorithm - parametr określający jaki algorytm ma zostać użyty do wyszukania ścieżki. Przyjmuje parametr typu String o wartości ASTAR, AGREEDY lub NBA.

   Przykładowe zapytanie HTTP GET w celu uzyskania odpowiedzi zawierającej optymalną trasę przejazdu można uzyskać pod adresem URL:

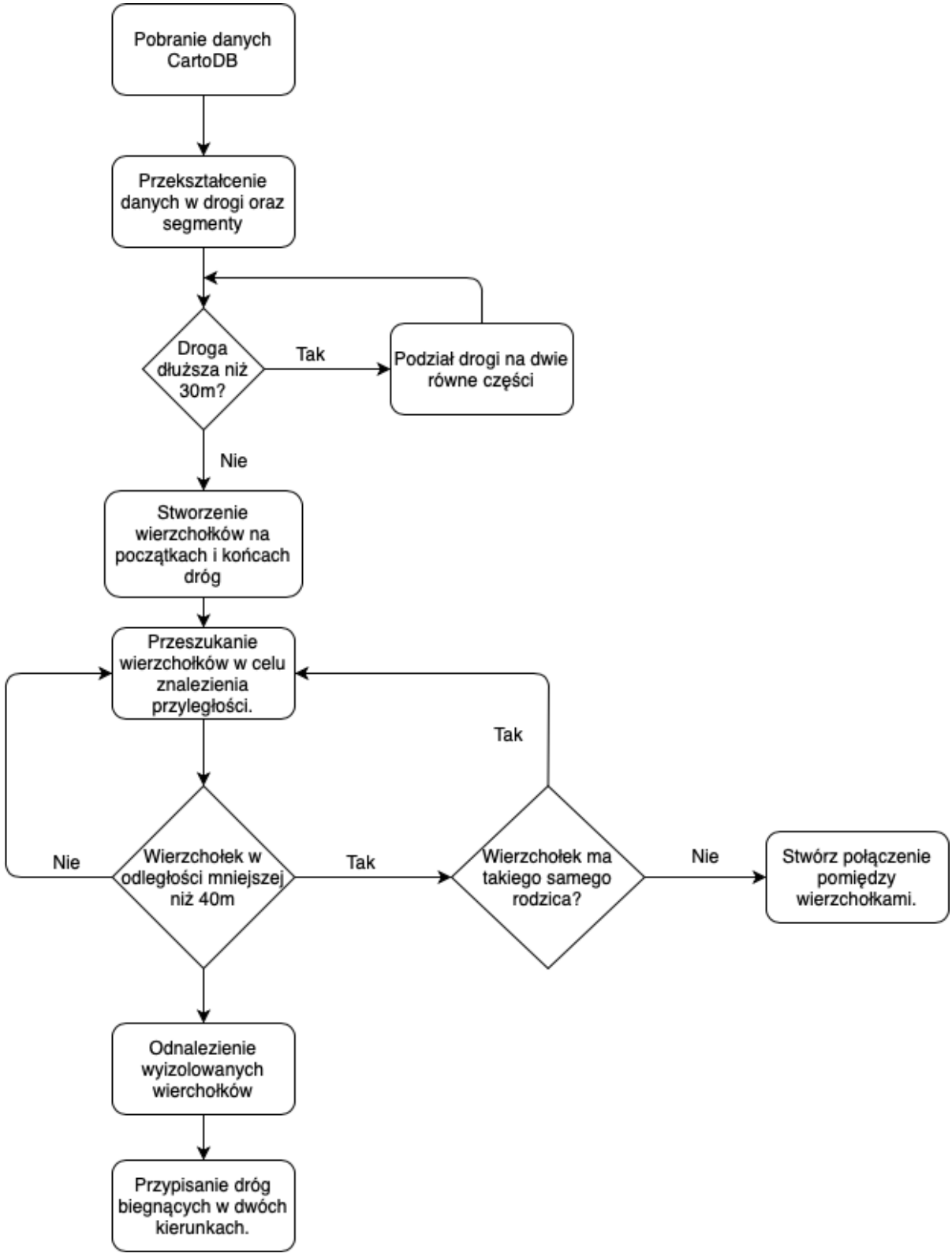
   bazowy_adres_url/api/routes/visualizationPoints?startLocation=romera&endLocation=karmelicka&routeType=BEST&algorithm=ASTAR

   Przykładową odpowiedź, skróconą do postaci przedstawiającej jej formę przedstawiono poniżej:

   // tutaj idzie przeklejona odpowiedź API, najlepiej w postacie screen-shot'a żeby było formatowanie

5.1.2. Proces wyznaczania grafu na podstawie pobranych danych

Na poniższym schemacie blokowym przedstawiono proces wyznaczania grafu na podstawie pobranych danych w celu dodatkowej wizualizacji jego przebiegu:



W celu optymalizacji czasu działania systemu, dane geoprzestrzenne są pobierane oraz przetwarzane przez aplikację tylko jeden raz, zaraz po włączeniu. Jest to czasochłonny proces który wymaga wielu iteracji po rozległym zbiorze danych którego rezultatem jest stworzenie grafu który może być w kolejnych etapach wykorzystywany jako gotowa dana wejściowa. Tworzenie grafu zostaje rozpoczęte przez pobranie zestawu danych geoprzestrzennych dla miasta Kraków z portalu https://zikit.carto.com. Zostaje stamtąd pobrana baza danych zarówno ścieżek rowerowych jak i dróg z ograniczeniem prędkości poniżej 50 kilometrów na godzinę, obydwie bazy są następnie scalane i przekształcane w zbiór odpowiednich dróg i segmentów. Droga jest logiczną reprezentacją krzywej poprowadzonej na mapie pomiędzy dwoma punktami, składa się z segmentów które zawierają jedynie punkt początkowy oraz końcowy i definiują zestaw linii prostych które mogą zostać użyte aby narysować linię drogi na mapie. Wykorzystując metodę Haversine, zostają wyznaczone długości każdej drogi przez zsumowanie długości wszystkich segmentów które się na nią składają. Kolejnym etapem jest proces przygotowywania dróg do procesu tworzenia grafu. Aby mieć pewność że wszystkie przyległości zostaną poprawnie odnalezione, w pierwszym kroku należy odpowiednio podzielić wszystkie pobrane drogi na takie o mniejszej długości. Trasy pobrane z serwisu Carto mają bardzo losowe długości oraz są zaznaczone w taki sposób, że nie występuje segmentacja w okolicach końców innych tras oraz przy przecięciach z nimi. Z tego powodu wyznaczenie poprawnego grafu z uzyskanych danych wymaga wcześniejszego przetworzenia. Pierwszym krokiem przetworzenia danych jest podzielenie segmentów uzyskanych z przetworzenia pobranych danych na takie które mają maksymalnie trzy metry długości. Dzieje się to przez podział każdego z segmentu na pół do momentu gdy osiągną one pożądaną długość. W następnej kolejności, gdy posiadamy już podzielone segmenty, następuje podzielenie tras. Dzieje się to na zasadzie podzielenia tablicy segmentów na części o określonej długości a następnie utworzenia zestawu nowych dróg z uzyskanych części. Każda droga stworzona przez podzielenie innej drogi na mniejsze części posiada także przypisanie swojego rodzica, pierwotnej drogi z której nastąpił podział. Droga będąca rodzicem, posiada za to tablicę zawierająca wszystkie drogi będące jego dziećmi. Na tym etapie następuje także przypisanie drodze odpowiedniej wagi oraz stwierdzenie czy droga przebiega po moście, jest to stwierdzane na podstawie wykrycia słów kluczowych „most" lub „kladka" w nazwie trasy. Niestety baza danych carto ZiKiT nie oferuje w swoich danych takich informacji, dlatego też występuje potrzeba przewidzenia tej danej w inny sposób. Proces tworzenia grafu rozpoczyna się od stworzenia wierzchołka w każdym miejscu będącym zakończenie lub rozpoczęciem drogi. W tym celu następuje iteracja po wszystkich drogach będących rodzicami, dla każdej z nich następuje iteracja po wszystkich ich dzieciach i tworzone są wierzchołki na początku pierwszej drogi oraz na zakończeniu każdej następnej. Każdy z utworzonych w tym procesie wierzchołków ma także przypisywaną trasę rodzica, który jest rodzicem drogie dla której wierzchołek został stworzony. Kolejnym etapem jest wyszukanie przyległości dla każdego z wierzchołków. W tym celu następuje zagnieżdżona iteracja po wszystkich stworzonych wierzchołkach i porównywana jest odległość pomiędzy nimi. W przypadku gdy odległość ta jest mniejsza niż 40 metrów, następuje połączenie wierzchołków drogą o typie standard link. W tym procesie ignorowane są wierzchołki posiadające takiego samego

rodzica, w celu uniknięcia łączenia wierzchołków już ze sobą połączonych w obrębie jednej drogi. Ignorowane są także wierzchołki których trasa-rodzic znajduje się na moście, wyodrębniając z tego wyjątek gdy aktualnie rozpatrywany wierzchołek jest początkiem lub końcem trasy do której przynależy. Zabieg ten ma na celu uniknięcie łączenia za sobą dróg znajdujących się na różnej wysokości. W kolejnym kroku działania algorytmu, obsługiwane są drogi dwukierunkowe, są to takie które w polu kategoria zawierają jeden z poniższych skrótów:

- dwr - droga rowerowa.
- cpr - ciąg pieszo rowerowy.
- c16t22 - chodnik z dopuszczonym ruchem rowerowym, skrót pochodzi od identyfikatora znaku drogowego.
- kontrapas i kontraruch - zakładamy że w tym przypadku użytkownik może poruszać się w przeciwną stronę po ulicy na której dana ścieżka się znajduje.
- skróty pokrewne zawierające kombinację powyższych lub dopisek w postaci cyfry.

Każda z dróg przypisanych do każdego z wierzchołków, w przypadku gdy jest oznaczona jako dwukierunkowa, musi zostać odpowiednio przetworzona aby algorytm wyszukiwania mógł przez nią prowadzić. W tym celu wszystkie takie drogi wychodzące z wierzchołka zostają dodane do dróg wchodzących do wierzchołka a następnie zostają odwrócone, czyli podmieniany jest identyfikator wierzchołka początkowego oraz końcowego, współrzędne końca i początku a także w analogiczny sposób zostaje odwrócony każdy segment przynależący do danej drogi. Ta sama procedura zostaje powtórzona w celu transformacji także dróg wchodzących do wierzchołka. Ostatnim z kroków przygotowywania grafu jest wyodrębnienie samotnych wierzchołków. Są to wierzchołki które znajdują się w mało zagęszczonym drogami obszarze i znajdują się zbyt daleko aby znaleźć dla nich przyległości, pomimo że powinny one zostać znalezione. Przykładem dla tego przypadku może być droga rowerowa na mało uczęszczanej ulicy, która w pewnym momencie z braku możliwości poprowadzenia zostaje przerwana na 100-150m po czym następuje jej kontynuacja w obrębie tej samej ulicy. W tym celu dla każdego z wierzchołków, który posiada tylko jedną drogę przychodząca, zostaje przeszukana okolica najbliższych 150 metrów. W przypadku znalezienia przyległości w odpowiedniej ilości, proces jest ignorowany gdy zostaje znalezione zbyt dużo, wierzchołek zostaje z nimi połączony drogą typu isolation link.

Po przeprowadzaniu całego powyższego procesu zostaje wygenerowany graf składający się z wierzchołków i w którym drogi pełnią rolę krawędzi. Ostatnim krokiem, przeprowadzanym podczas tworzenia docelowego grafu, jest przypisanie wagi każdej z dróg. Zważają na fakt że baza danych carto zawiera informacje odnośnie typu większości ścieżek, aplikacja stara się przewidzieć zysk użytkownika z przebycia nieznacznie dłuższej drogi znacznie bardziej komfortowymi drogami. W tym celu przypisywane są w zależności od kategorii wagi:

- Ddr - waga 0.7

– Cpr - waga 0.8

– Kontrapas, kontraruch - waga 0.8

– C16t22 - waga 0.9

– Standard link - w przypadku gdy krótszy niż 10 metrów, posiada wagę 1.5. W przeciwnym wypadku waga jest równa 2.

– Isolatation link - waga 4

– Droga rowerowa bez kategorii - waga 1

– Droga z bazy sugerowanych tras - waga 1.2

5.1.3. Opis procesu wyznaczania trasy dla użytkownika

Kolejnym etapem działania algorytmu jest wyszukiwanie tras na podstawie danych wejściowych przekazanych przez użytkownika. Gdy zostanie zarejestrowane zapytanie pod endpoint „findOptimized", aplikacja odpowiednio przetwarza dane w celu uzyskania współrzędnych geograficznych punktu początkowego oraz końcowego trasy. Wykorzystywany jest do tego serwis openstreetmap, odpowiednio skonfigurowany w celu dekodowania lokalizacji znajdujących się w Krakowie. W zależności od wybranej metody przeszukiwania brany jest także pod uwagę przekazany przez użytkownika parametr routeType. W przypadku przekazania drogi typu „BEST" przy tworzeniu odpowiedniego typu obiektu „graph", używanego później do przeszukiwania, do połączenia każdego z wierzchołków używana jest rzeczywista waga drogi, za to w przypadku przekazania drogi typu „SHORTEST" ignorowane są wagi dróg i brana pod uwagę jest tylko ich całkowita długość. W kolejnym etapie graf zostaje przeszukany w celu znalezienia wierzchołków znajdujących się najbliżej pozycji wybranych przez użytkownika. W celu zminimalizowania prawdopodobieństwa sytuacji w której znalezione wierzchołki znajdują się w dwóch grafach które są wobec siebie rozbieżne, są także wyszukiwane wszystkie wierzchołki znajdujące się w otoczeniu 300 metrów od wyznaczonego najbliższego. Metoda ma na celu w głównej mierze wykluczenie wierzchołków przynależących do odosobnionych dróg rowerowych, które ze względu na zbyt dużą odległość nie zostały włączone do żadnego większego grafu i są reprezentowane przez odrębny graf składający się jedynie z jednej krawędzie i dwóch wierzchołków, wyznaczających początek i koniec drogi. Kolejnym krokiem działania algorytmu jest wyznaczenie wszystkich możliwych ścieżek pomiędzy wszystkimi odnalezionymi punktami początkowymi i końcowymi. Wykorzystywany jest do tego algorytm A*. Każda z odnalezionych ścieżek ma przypisywaną zsumowaną wagę wszystkich dróg które do niej przynależą z wagą 1 oraz sumę odległości początku i końca ścieżki w stosunku do punktów początkowego i końcowego przekazanych przez użytkownika z wagą 20. Krok ten ma na calu eliminację dróg które są najkrótsze, ale są podzbiorem innej dłuższej drogi która prowadzi użytkownika tą samą trasą. Ze wszystkich wyznaczonych dróg zostaje wybrana ta która ma najmniejszą wagę. Następnie wyznaczona ścieżka musi zostać odpowiednio przekształcona aby zawierała jak najkrótsze połączenia

pomiędzy drogami które do niej przynależą. W tym celu zostają z niej wyeliminowane segmenty początkowe i końcowe każdej z dróg w przypadku gdy wykryto że prowadzą użytkownika okrężną drogą. Ostatnim etapem jest połączenie każdego punktu końcowego każdej drogi przynależącej do danej ścieżki z punktem początkowym kolejnej prostą drogą łącząca. Wyznaczona oraz przekształcona ścieżka zostaje zwrócona użytkownikowi w formie odpowiedzi w formacie JSON zawierającej współrzędne punktów początkowego i końcowego oraz zbioru dróg podzielonych na segmenty które pomiędzy nimi prowadzą.

5.2. Opis stworzonej aplikacji mobilnej

5.2.1. Opis technologii

Aplikacja mobilna została zaimplementowana pod system iOS przy użyciu SDK dostarczonego przez firmę Apple umożliwiającego implementację aplikacji. Implementację interfejsu użytkownika umożliwia biblioteka CocoaTouch będąca jego częścią. Z racji faktu że aplikacja wykorzystuje moduł GPS w celu wyznaczania aktualnego położenia użytkownika oraz bibliotekę MapKit wchodzącą w część biblioteki CocoaTouch, w poniższych akapitach opisano działanie obydwu bibliotek wraz z przykładami użycia wewnątrz zaimplementowanej aplikacji. CoreLocation jest biblioteką wchodzącą w skład iOS SDK. Oferuje interfejs do wyznaczania lokalizacji użytkownika ale odpowiada także za obsługę wielu innych sensorów obsługiwanych przez telefon jak na przykład magnetometru, barometru, a nawet Bluetooth - do wyznaczania położenia w stosunku do urządzeń w standardzie iBeacon. Przed uzyskaniem lokalizacji użytkownika, w celu ochrony prywatności, musi zostać wyświetlony odpowiedni komunikat, w którym użytkownik wyraża zgodę na udostępnienie swojej lokalizacji dla aplikacji. W następnej kolejności, przy użyciu delegacji klasy CLLocationManager aplikacja uzyskuje co sekundę odpowiedź zwrotną zawierająca aktualną lokalizację użytkownika a także jej dokładność w metrach. Biblioteka CoreLocation jest także używana do wyznaczania kierunku w którym użytkowni skierował swój telefon. Umożliwia to płynną animację ikony roweru podczas prowadzenia użytkownika po trasie, w interfejsie użytkownika jest on skierowany w tę samą stronę na mapie, co telefon w rzeczywistości. Dodatkową funkcją biblioteki wykorzystywaną w aplikacji jest geocoding, czyli wyznaczanie współrzędnych geograficznych na podstawie wpisanego tekstu adresu lub działanie do tego odwrotne. Użyta została do tego klasa CLGeocoder będąca częścią CoreLocation. Drugą biblioteką zapewniającą działanie aplikacji iOS jest MapKit - biblioteka umożliwiająca wyświetlanie map na ekranie telefonu. Została wybrana w stosunku do map Google z racji na prostszą implementację w aplikacjach pisanych na system iOS a także fakt że jest rozwiązaniem natywnym, zaimplementowanym przez firmę Apple, nie zewnętrzną biblioteką.

5.2.2. Schemat klas aplikacji mobilnej

Za zarządzanie danymi oraz stanem aplikacji odpowiada klasa NavigationManager, jest to swojego rodzaju łącznik pomiędzy widokiem głównym a wszelkimi serwisami takimi jak klient nawigacji lub serwis zapewniający lokalizację użytkownika lub klasa odpowiedzialna za wykonywanie zapytań HTTP.

Zarządza w odpowiedni sposób danymi wejściowymi oraz interakcjami użytkownika przekazanymi z ekranu głównego aby zareagować odpowiednimi akcjami i zmianą stanu który następnie przekazany do warstwy widoku przez delegację odpowiada za wyświetlenie odpowiednich danych dla użytkownika. Do odpowiedzialności klasy NavigationManager należą:

1. Obserwacja danych wpisywanych przez użytkownika do pól punktu startowego oraz końcowego.
2. Obsługa przycisku wyznaczenia aktualnego adresu użytkownika. Po jego wciśnięciu do klasy zostaje przekazane odpowiednie zdarzenie na które reaguje dekodowaniem aktualnej lokalizacji użytkownika na adres na mapie.
3. Pobranie trasy przy użyciu endpointu „findOptimized" po akcji użytkownika w postaci przyciśnięcia przycisku „Navigate" oraz zmianę stanu aplikacji na zaznaczenie trasy gdy zapytanie się powiedzie.
4. Włączenie oraz wyłączenie procesu nawigowania użytkownika w przypadku gdy została zatrzymana przez użytkownika lub znalazł się on w punkcie końcowym.
5. Na poniższej ilustracji przedstawiono schemat działania aplikacji razem z interakcjami pomiędzy jej komponentami:

// tutaj idzie schemat aplikacji.

5.2.3. Spis ekranów i stanów aplikacji

Zaimplementowana aplikacja mobilna składa się z zestawu ekranów które umożliwiają użytkownikowi intuicyjne wprowadzenie parametrów trasy oraz przejście przez każdy stan nawigacji aż do doprowadzenia do końca trasy. W poniższych akapitach zestawiono przykładowe ekrany aplikacji wraz z opisem ich celu i możliwych interakcji ze strony użytkownika. Bezpośrednio po wejściu do aplikacji, użytkownikowi zostaje przedstawiony ekran z mapą oraz dwoma polami tekstowymi, każde z nich posiada placeholder w celu zasugerowania którą z wartości należy wpisać. Po prawej stronie pola tekstowego służącego do pisania adresu początkowego znajduje się przycisk lokalizowania użytkownika. Po kliknięciu aplikacja pobiera z modułu GPS aktualną pozycję użytkownika, a następnie używając wbudowanej w iOS SDK klasy CLGeocoder wykonuje translację współrzędnych geograficznych na tekstową reprezentację adresu. Wyznaczony adres zostaje wpisany w pole tekstowe. Na poniższej ilustracji przedstawiono ekran początkowy w stanie gdy nie ma wprowadzonych danych.

// tutaj idzie screen pierwszego ekranu

Po wprowadzeniu punktów końcowych trasy, następuje jej wyszukanie, w tym celu aplikacji przygotowuje zapytanie pod endpoint „findOptimized" przekazując dane wpisane w polach tekstowych, a na czas ładowania odpowiedzi wyświetla indykator ładowania na górnym panelu oraz blokuje interakcje użytkownika ze wszystkimi elementami interfejsu użytkownika poza mapą. Po pobraniu trasy jest ona przekształcana na modele po stronie aplikacji oraz rysowana na mapie. Linią ciągłą zaznaczona

jest trasa, linią przerywaną odcinki poza trasowe które trzeba pokonać aby dostać się z aktualnej lokalizacji użytkownika do początku trasy oraz z końca trasy do punktu końcowego wprowadzonego przez użytkownika. W celu odpowiedniej wizualizacji, po narysowaniu mapa jest odpowiednio przybliżana aby pokazać użytkownikowi przebieg całej trasy od jej początku do końca. Ekran z narysowaną mapą przedstawiono na poniższej ilustracji.

// ekran z trasą narysowaną na mapce, przybliżony

Po wciśnięciu przycisku nawiguj zostaje włączony proces nawigacji użytkownika po trasie. W pierwszej fazie jest to prowadzenie do początku trasy, mapa zostaje odpowiednio przybliżona aby objąć trasę z punktu początkowego na początek trasy, lokalizacja użytkownika jest przedstawiona w postaci standardowego punkty na mapie obsługiwanego przez bibliotekę MapKit.

// ekran z zaznaczoną trasą użytkownika do początku trasy

Po dojściu użytkownika w obręb wyznaczonej trasy, interfejs użytkownika rozpoczyna nawigowanie go po trasie. Punkt wyznaczający aktualną lokalizację jest co sekundę animowanie przesuwany na pozycję na trasie odpowiadającą najbliższemu segmentowi w stosunku do rzeczywistej pozycji użytkownika na mapie. W przypadku wykrycia przez aplikację zakrętu w obrębie około najbliższych 200 metrów, na górnym panelu jest wyświetlana wskazówka zawierająca kierunek zakrętu oraz aktualną odległość po której wystąpi. W przypadku gdy użytkownik zjedzie z trasy, na górnym panelu jest przedstawiona instrukcja sugerująca powrót na trasę, zaś gdy oddali się od niej za daleko, instrukcja zmienia się w informację o wyznaczaniu nowej trasy do punktu końcowego.

// 2 ekrany, jeden z nawigacją gdy jest strzałka, drugi z reroutingiem

5.2.4. Opis działania nawigacji

System nawigacji w aplikacji został zaimplementowany w celu wizualizacji aktualnej lokalizacji użytkownika w stosunku do trasy która została wyznaczona. Działanie systemu składa się z zestawu stanów pomiędzy którymi algorytm może przechodzić w przypadku wykrycia określonych warunków. W poniższym akapicie została przedstawiona lista stanów na które składa się proces nawigacji a także dokładny opis warunków koniecznych do przejścia pomiędzy nimi. W celu wizualizacji procesu został także stworzony diagram stanów odwzorowujący cały proces.

Lista stanów w których może znaleźć się proces nawigacji użytkownika:

- Dojście z aktualnej lokalizacji do punktu startowego trasy
- Nawigacja użytkownika po trasie
- Dojście z końca trasy do punktu końcowego nawigacji
- Zejście z trasy - w przypadku gdy użytkownik znalazł się więcej niż 30m ale nie więcej niż 150m od trasy.
- Potrzeba wyznaczenia nowej trasy - gdy użytkownik znalazł się dalej niż 150m od najbliższego segmentu trasy.

W momencie gdy użytkownik rozpoczyna nawigację, zostaje włączony proces który co sekundę pobiera aktualną lokalizację użytkownika z klienta lokalizacji i wyznacza aktualny stan aplikacji. Startowym stanem jest prowadzenie użytkownika do początku trasy. Jako że pokrycie Krakowa ścieżkami rowerowymi oraz drogami z niskim ograniczenie prędkości jest względnie małe, w większości przypadków użytkownik na początku będzie musiał przemieścić się kilkaset metrów poza trasami wspieranymi przez aplikację. W tym celu jest rysowana prosta linia pomiędzy punktem początkowym, wpisanym przez użytkownika, a początkiem trasy a mapa jest odpowiednio przybliżana aby umożliwić użytkownikowi proste dotarcie na drogę. W czasie fazy prowadzenia użytkownika na start trasy, co każdy cykl odświeżenia aktualnego stanu, jest sprawdzane czy użytkownik znalazł się w otoczeniu 10 metrów w stosunku do początku któregokolwiek z segmentów składających się na trasę. Sprawdzanie jedynie segmentu będącego punktem początkowym trasy nie daje w tym wypadku oczekiwanego rezultatu ze względu na częste błędy w wyznaczeniu pozycji użytkownika przez moduł GPS a także przez fakt że często na trasę wjeżdżamy nie dokładnie w punkcie jej początku a przykładowo dopiero po pierwszych stu metrach. Posiadając dostępny zbiór danych aplikacja nie jest w stanie niestety wyznaczyć optymalnej trasy użytkownika pomiędzy jego aktualnym położeniem a punktem końcowym trasy. Jesteśmy w stanie jedynie oszacować że najbardziej prawdopodobnym punktem będzie najbliższy wierzchołek grafu zakładając ich odpowiednią granulację otrzymaną podczas procesy jego tworzenia. W przypadku wykrycia wejścia użytkownika w obręb wyznaczonej dla niego trasy aplikacja przechodzi w stan nawigowania po trasie. Aby uniknąć błędów w wyznaczaniu pozycji użytkownika i prowadzenia go 10-20 metrów obok zaznaczonej drogi, został zaimplementowany mechanizm dociągania do wyznaczonej ścieżki. W tym celu w każdym kroku filtrowane są wszystkie drogi w celu znalezienia tej znajdującej się najbliżej użytkownika, zastosowano do tego porównanie sumy odległości od początków z każdej tych dróg, podzielonej przez jej długość. Dzięki temu rozwiązaniu algorytm wyeliminował znajdywanie jedynie krótkich dróg dla których suma odległości do początku oraz do końca była najmniejsza. Po odnalezieniu najbliższej drogi, algorytm przeszukuje wszystkie jej segmenty w celu znalezienia tego znajdującego się najbliżej. Jako że długości segmentów zostały ujednolicone w jednym z kroków tworzenia grafu, na tym etapie możemy zastosować prostsze porównanie które sprawdza jedynie sumę odległości użytkownika od początku oraz końca każdego z segmentów. W następnym kroku, uzyskany segment zostaje podzielony na 20 równych sobie odcinków, najbliższy aktualnej pozycji użytkownika zostaje przypisany jako najbardziej odpowiedni a aplikacja sztucznie dociąga lokalizację użytkownika do tego właśnie punktu. Powyżej opisana metoda działa w przypadku gdy wyznaczona pozycja użytkownika znajduje się nie dalej niż 50 metrów od trasy. W przypadku gdy użytkownik znajdzie się w odległości większej niż 50 metrów, algorytm przestaje zwracać pozycję użytkownika dociągniętą do trasy a zamiast tego zwraca rzeczywiste położenie użytkownika na mapie. Ten tryb został zaimplementowany w celu pokazania użytkownikowi informacji że znalazł się poza trasą i powinien na nią wrócić. Ogranicza to także niepotrzebne zapytania do strony serwerowej w przypadku gdy użytkownik celowo zszedł z trasy aby na przykład wejść do sklepu. W przypadku gdy wykryte położenie znajduje się dalej niż 150m od trasy zaznaczonej na mapie, algorytm zakłada że należy dla użytkownika wyznaczyć nową trasę, o czym informuje przez wysłanie

określonego sygnału oraz zakończenie działania. W tym momencie zostaje wysłane zapytanie do serwera o nową trasę zawierające aktualną pozycję użytkownika oraz miejsce docelowe określone na początku procesu nawigacji. W przypadku gdy zapytanie się powiedzie, algorytm zostaje zrestartowany i wraca do stanu początkowego, czyli prowadzenia użytkownika do początku trasy. W każdym kroku działania algorytmu jest także sprawdzana odległość aktualnej pozycji użytkownika w stosunku do końca wyznaczonej trasy. Jeśli ta jest mniejsza niż 50 metrów, zakładamy że użytkownik dotarł do końca i nawigacja przechodzi w tryb prowadzenia użytkownika do punktu końcowego trasy, pokazując jednocześnie odpowiednio przybliżony obszar na mapie. W tym kroku założona odległość od punktu końcowego musi być znacznie większa niż ta która stanowi o momencie wejścia na trasę ponieważ użytkownik może znajdywać się w najbliższym otoczeniu punktu końcowego jedynie przez krótką chwilę.

5.2.5. Opis testów

Do testów działania nawigacji zostały wykorzystane pliki GPX(GPS Exchange Format) które, sformatowane w odpowiedni sposób umożliwiają symulację lokalizacji użytkownika zarówno przy użyciu symulatora jak i na rzeczywistym urządzeniu z systemem iOS. Plik GPX jest to plik w formacie XML który składa się. Ze zbioru punktów oraz czasów skorelowanych z każdym z punktów. System odczytując ten plik animuje lokalizację użytkownika tak aby przeszła pomiędzy wszystkimi określonymi punktami w ściśle wyznaczonym czasie. Do stworzenia plików użyto strony http://www.gpsies.com/createTrack.do która udostępnia graficzny interfejs do zaznaczania punktów na mapie oraz określenia z jaką prędkością chcielibyśmy aby użytkownik się pomiędzy nimi przemieszczał. Pobrany stamtąd plik następnie trzeba poddać obróbce w postaci usunięcia kilku linii aby był wspierany przez symulację lokalizacji w systemie iOS.

5.3. Opis stworzonej strony internetowej

5.3.1. Opis technologii i użytych bibliotek

W ramach projektu została zaimplementowana także strona internetowa, która w intuicyjny sposób pozwala użytkownikowi wyszukać trasę pomiędzy wpisanymi adresami oraz wyświetla ją na mapie. Aplikacja została stworzona w oparciu o bibliotekę ReactJS która pozwala w szybki sposób zbudować prototyp strony internetowej przy użyciu języka Javascript oraz predefiniowanych elementów strony takich jak przyciski czy pola tekstowe. Do wyświetlenia map użyto map udostępnionych przez openstreetmap przez bibliotekę React Leaflet. Leaflet jest to biblioteka umożliwiająca bardzo proste renderowanie map przy użyciu Javascript, React Leaflet dodaje do tego mapy w postacie gotowych komponentów React. Wybór został podyktowany faktem, że prostsze w użyciu i wydajniejsze mapy Google w celu wyświetlenia przy użyciu języka Javascript wymagają płatnej subskrybcji. W projekcie strony internetowej, do zarządzania zewnętrznymi zależnościami użyto programu yarn, a do zapytań http użyto biblioteki

axios. Narzędzia te nie różnią się od tych dla aplikacji serwerowej, dlatego też w tym akapicie pominięto ich opis.

5.3.2. Spis ekranów, opis działania

Aplikacja jest bardzo prosta, składa się z dwóch ekranów. Na pierwszym z nich są przedstawione dwa pola tekstowe oraz przycisk umożliwiający wyszukanie trasy, jego wygląd przedstawiono na poniższej ilustracji.

// tutaj idzie screen pierwszego ekranu frontendu

Kolejny z ekranów ma zaimplementowaną logikę pobierania trasy w zależności od stanu przekazanego do niego z ekranu wpisywania danych. Wykorzystując bibliotekę axios wykonuje zapytanie pod endpoint „visualizationPoints" zawierając w kwerendzie wymagane dane. Zwrócona odpowiedź w formacie JSON zawiera zbiór wszystkich punktów na mapie który należy ze sobą połączyć aby przestawić użytkownikowi wizualizację wyznaczonej ścieżki. Ścieżka jest przedstawiona na mapie obejmującej całość ekranu.

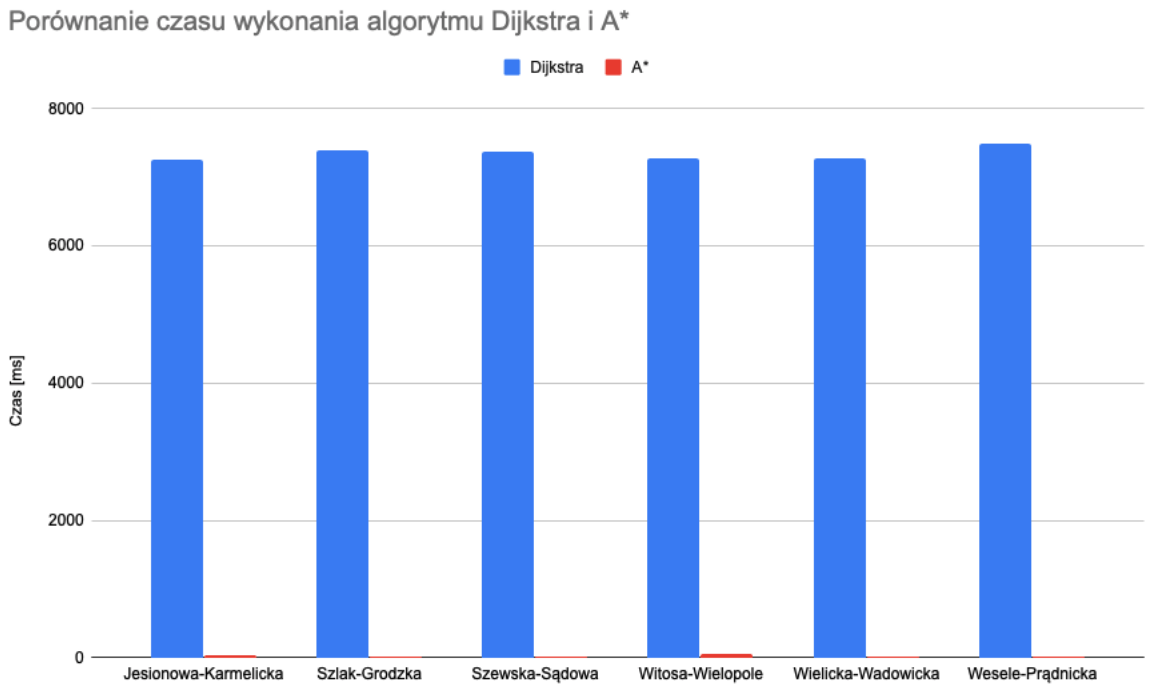
6. Analiza otrzymanych wyników

6.1. Analiza działania użytych algorytmów

W celu implementacji algorytmu wyszukania optymalnych tras, w aplikacji została zaimplementowana obsługa kilku algorytmów wyszukiwania ścieżek w grafie. Standardowo aplikacja korzysta tylko z jednego z nich, implementacja ma na celu tylko ich porównanie. W poniższych akapitach przedstawiono testy porównawcze prędkości działania algorytmów podczas wyznaczania tras pomiędzy zestawem wybranych losowo punktów na mapie Krakowa. W celu heterogenizacji wyznaczonych tras, punkty początkowe i końcowe każdej z nich zostały dobrane w taki sposób aby pokryły odpowiedni obszar Krakowa a także przebiegały w miejscach gęsto i luźno obstawionych przez ścieżki zawarte w posiadanym zbiorze danych. Z tego powodu niektóre z nich przebiegają w okolicach rynku, a inne przez Wolę Justowską gdzie jedyną ścieżką w posiadanym zbiorze jest droga po wale rzeki Rudawy. Obydwa z zaimplementowanych algorytmów gwarantują każdorazowo wyznaczenie optymalnej trasy, nie kończą przeszukiwania po uzyskaniu pierwszej znalezionej trasy, stąd porównanie można uznać za miarodajny wynik złożoności obliczeniowej każdego z algorytmów.

6.1.1. Porównanie działania algorytmu z wykorzystaniem algorytmu Dijkstra i A*

Na poniższym wykresie zestawiono czasy działania obydwu algorytmów wraz z opisem trasy dla której każdy z czasów został wyznaczony.



Z wykresu można odczytać że algorytm A* sprawdza się nieporównywalnie lepiej w stosunku do algorytmu Dijkstra w kontekście czasu wykonania. Z wykresu ciężko nawet odczytać różnice w czasie działania ze względu na minimalny wkład czasów osiągniętych przez A*. W najgorszym z zaprezentowanych przypadków jego działanie było około 200 krotnie szybsze, w najlepszym z przypadków różnica w prędkości działania była prawie 500 krotna. Z powyższego zestawienia jasno wynika, że do produkcyjnego zastosowania w aplikacji w porównaniu z algorytmem Dijkstra nadaje się tylko algorytm A*. Jest on za to znacznie bardziej czasochłonny w implementacji, stąd jego zastosowanie do zdecydowanie mniej złożonych grafów może być korzystniejsze. W celu zminimalizowania czasu wykonywania algorytmu Dijkstra można było zastosować odpowiednie zmniejszenie grafu na podstawie filtracji wierzchołków które znajdują się wewnątrz obszaru na mapie zawartego przez punkt końcowy i początkowy. Jednak ze względu na brak potrzeby optymalizacji i odpowiednią wizualizację różnic czasu działania zaniechano tej modyfikacji. W przypadku przeszukania grafu w którym wierzchołki mogą być określone jako punkty na mapie, algorytm A* jest zdecydowanie szybszy ze względu na prostotę wyznaczenia heurystyki. W tym wypadku przewidywanie czy droga zbliża się do końca może być określone przez wyznaczenie odległości pomiędzy ma złożoność obliczeniową O(1) i polega na wyznaczeniu odległości pomiędzy kolejnym sprawdzanym wierzchołkiem a punktem końcowym trasy. Dzięki temu, w przeciwieństwie do algorytmu Dijkstra, A* nie prowadzi przeszukania całego grafu przed wyznaczeniem optymalnej ścieżki a ogranicza się jedynie do przeszukania szeregu dróg które prowadzą pomiędzy punktem końcowym i początkowym. Ogólną złożoność obliczeniową algorytmu A* przedstawiono poniższym wzorem.

// tutaj wzór na złożoność ogólną A*

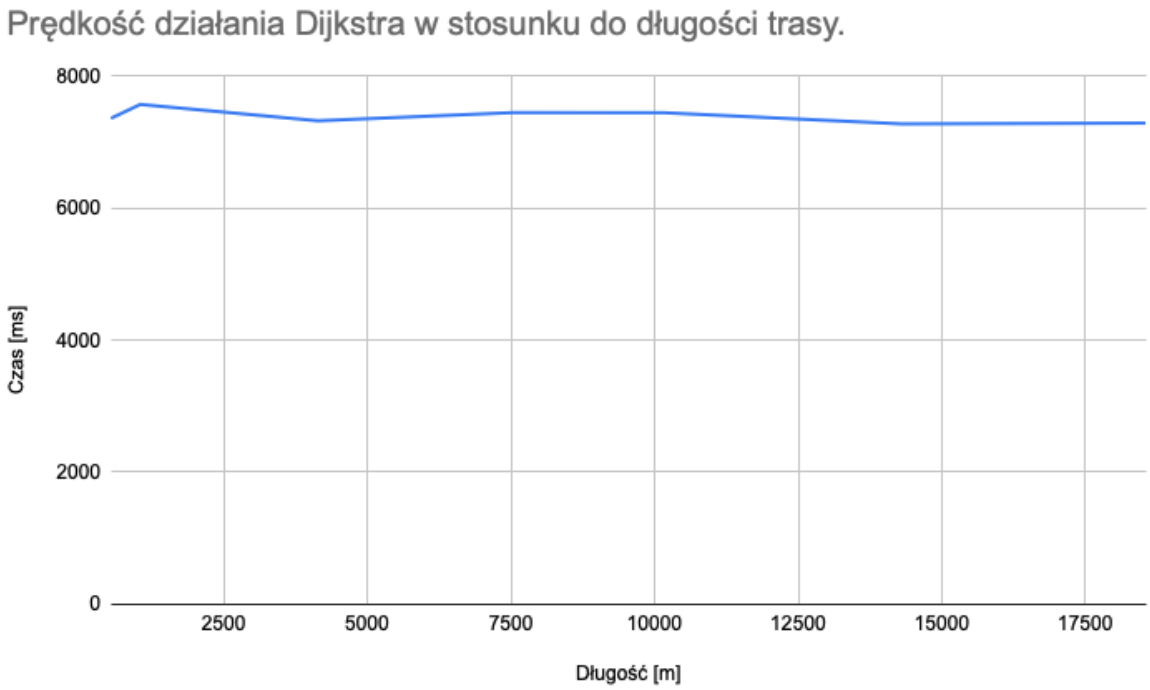
Dla przykładu algorytmu użytego w aplikacji może zostać wyznaczona wzorem:

// tutaj wzór na złożoność A* w tym przypadku.

Na podstawie wzoru widać że złożoność algorytmu A* w dużej mierze zależy od złożoności obliczeniowej wyznaczenia heurystyki. W przypadku gdy proces ten przedstawiał by się złożonością obliczeniową O(n2), algorytm działał by z prędkością porównywalną do algorytmu Dijkstra.

6.1.2. Czas działania algorytmu A* i Dijkstra w zależności od długości trasy

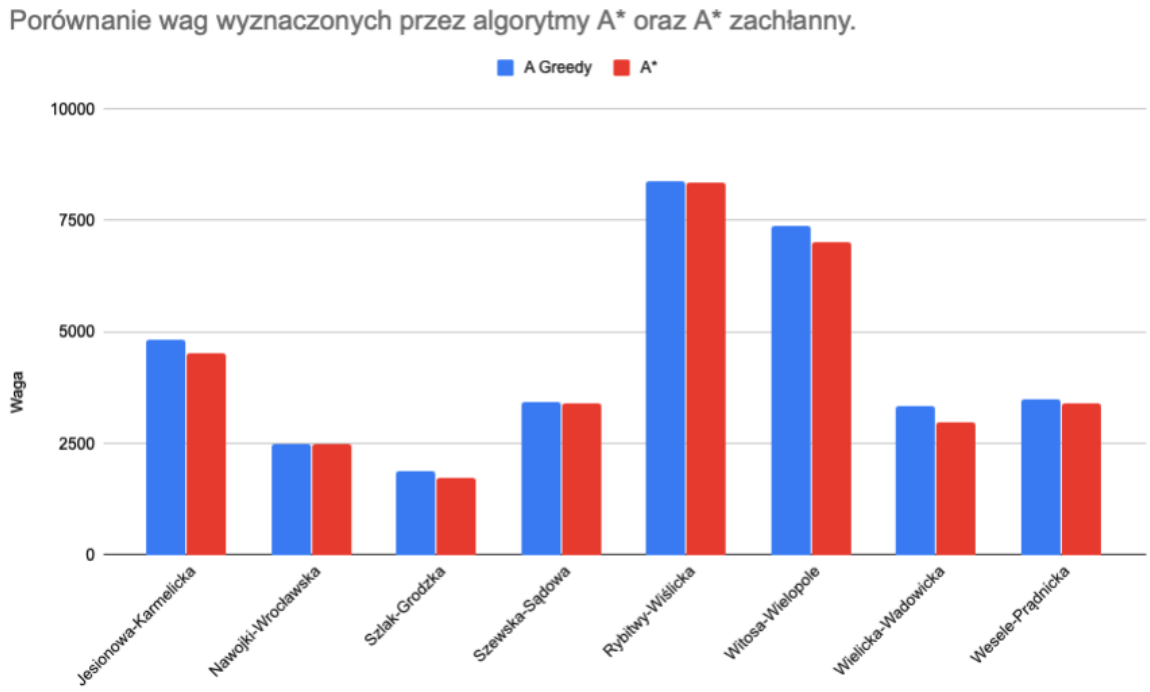
Na poniższych wykresach przedstawiono czas działania algorytmu A* i Dijkstra w zależności od obszaru objętego przeszukiwaniem. W tym celu na terenie Krakowa wyznaczono zestaw tras w zakresie od bardzo krótkich, obejmujących tylko kilkaset metrów do takich obejmujących teren całego Krakowa.



Z przedstawionych wykresów można odczytać że czas działania algorytmu Dijkstra nie zależy od długości wyznaczonych tras. Delikatne fluktuacje czasu wykonania zależą od aktualnego użycia pamięci i procesora komputera podczas wykonywania testu. Jest to spowodowane zasadą działania algorytmu Dijkstra, który w każdym przypadku w pierwszej kolejności wykonuje obliczenia najkrótszych ścieżek pomiędzy wszystkimi wierzchołkami grafu. Przeciwny przypadek obowiązuje dla algorytmu A* który dzięki zastosowaniu heurystyki jest w stanie wykryć gdy najkrótsza ścieżka została już odnaleziona i zatrzymać przeszukiwanie w odpowiednim przypadku. Z tego powodu w zależności od obszaru grafu obejmowanego przez przeszukanie, algorytm A* wykazuje znaczny, liniowy wzrost czasu wykonania.

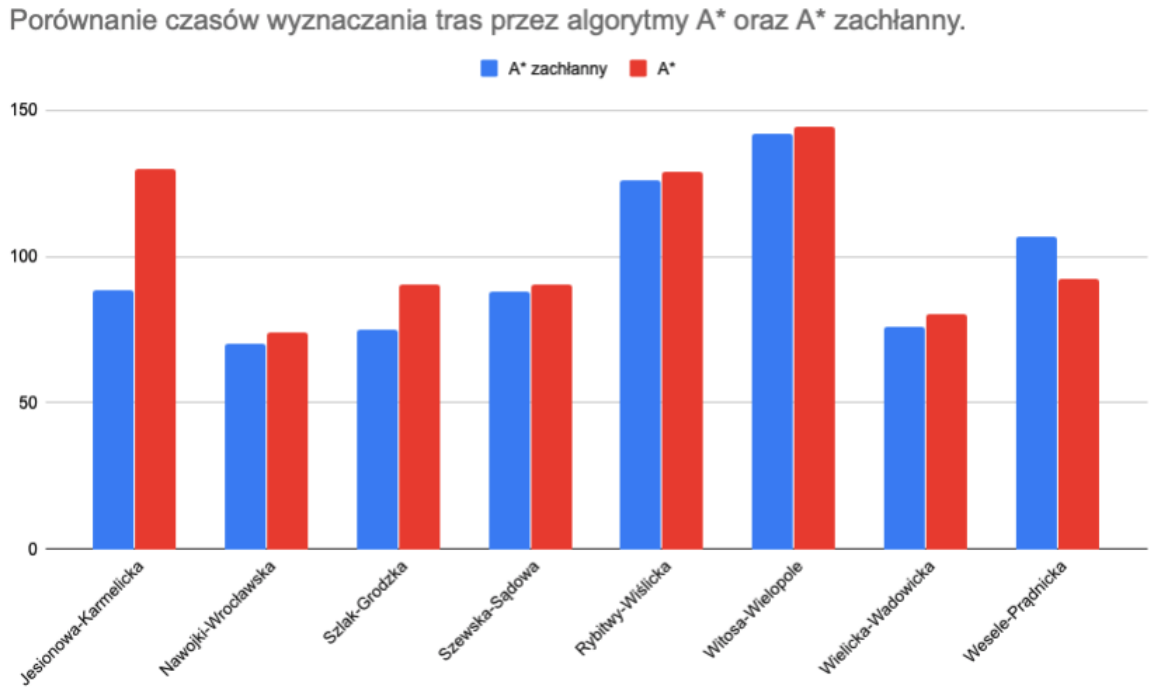
6.1.3. Porównanie algorytmu zachłannego A* i standardowej implementacji A*

Kolejnym etapem analizy jest określenie czasu oraz jakości działania algorytmu zachłannego A* w stosunku do podstawowej wersji A*. W przeciwieństwie do standardowej implementacji, algorytm zachłanny, zyskując na czasie wykonania, nie gwarantuje wyznaczenia optymalnej ścieżki pomiędzy wierzchołkiem początkowym i końcowym. Bazując na przekazanej heurystyce, stara się wyznaczyć najlepsze lokalne rozwiązania podzbiorów składających się na wynikową trasę, następnie łączy uzyskane podzbiory w celu wyznaczenia trasy. W poniższych akapitach zawarto analizę czasu wykonania oraz wyznaczonych wag tras w zależności od typu algorytmu. Na poniższym wykresie przedstawiono zestawienie wag tras wyznaczonych przez obydwa algorytmy dla uprzednio wyznaczonego zestawy ścieżek testowych.



Korzystając z wartości przedstawionych na powyższym wykresie można odczytać że poza przypadkiem bardzo krótkiej drogi pomiędzy ulicami Nawojki i Wrocławską, zachłanna odmiana algorytmu A* w każdym wypadku wyznaczyła trasę która z punktu widzenia waga jest gorsza niż ta wyznaczona przez standardową odmianę algorytmu A*.

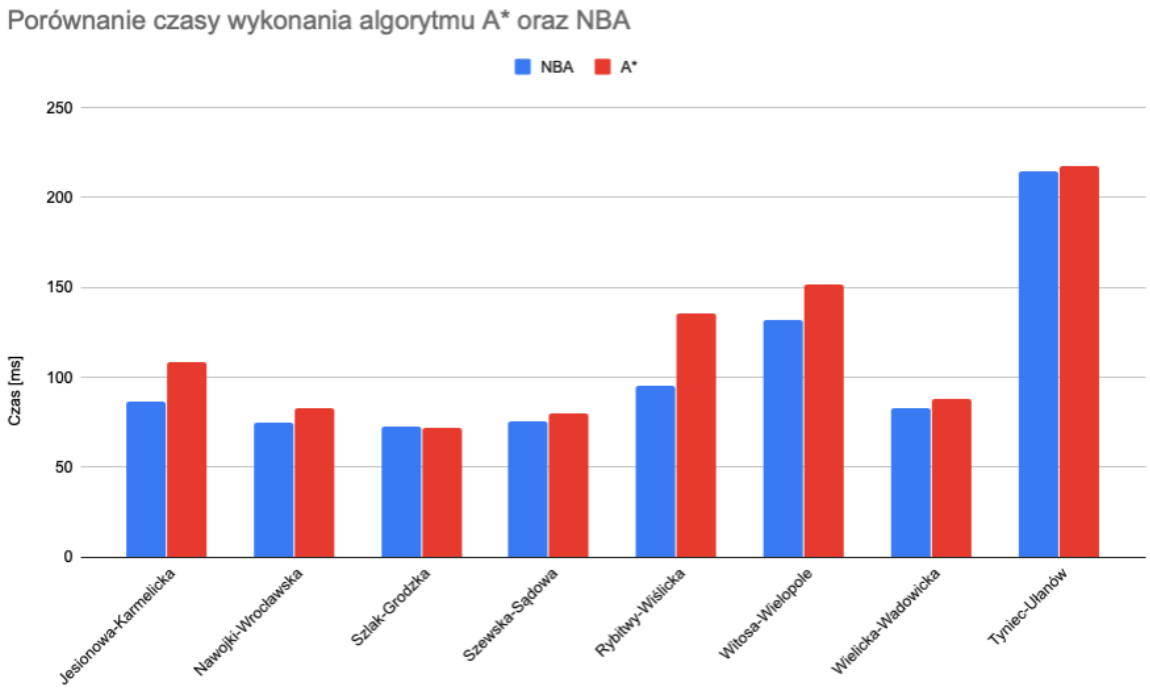
Na poniższym wykresie przedstawiono zestawienie czasu działania dla uprzednio wyznaczonego zestawu testowych ścieżek.



Za wyjątkiem jednego przypadku, który może być spowodowany chwilową mniejszą wydajnością maszyny na której działały testy, algorytm zachłanny uzyskał lepszy średni czas wyznaczania trasy. Jest to spodziewane, ponieważ algorytm ten w swojej zasadzie działania stara się optymalizować czas wykonania kosztem jakości. Różnice w czasie działania dla większości przypadków są bardzo niewielkie, sumując je z brakiem gwarancji wyznaczenia trasy optymalnej, algorytm zachłanny A* nie znajduje zastosowania do użycia w stworzonym algorytmie.

6.1.4. Porównanie algorytmu NBA i standardowej implementacji A*

W celu dalszej analizy możliwych usprawnień dla czasu działania algorytmu, został zaimplementowany i porównany także algorytm NBA*(New Bidirectional A*). Jest to odmiana algorytmu A* który w celu optymalizacji czasu działania jednocześnie rozpoczyna swoje wykonywanie z punktu końcowego do punktu początkowego oraz w przeciwną stronę. Złożoność obliczeniowa wykonania obydwu algorytmów pozostaje taka sama, jednak istnieje możliwość wykonywania dwóch algorytmów przy wykorzystaniu dwóch rdzeni procesora, dzięki czemu czas obliczeń ulega znacznej poprawie. W przeciwieństwie do algorytmu zachłannego, algorytm NBA gwarantuje każdorazowe wyszukanie optymalnej trasy w grafie. Na poniższym wykresie przedstawiono zestawienie czasów działania algorytmów. W celu otrzymania porównywalnych wyników, porównania zostały także przeprowadzone dla tego samego zestawu danych testowych.



Z uzyskanych wyników można odczytać że algorytm NBA daje możliwość uzyskania nieznacznie mniejszych czasów obliczeń. Same różnice nie są duże, ale znikomym nakładem pozwalają nieznacznie obniżyć wykorzystanie zasobów serwera w przypadku gdy aplikacja jest używana przez znaczną ilość

ludzi. Z wykresu można także odczytać nieznaczny trend w kierunku zwiększenia różnic wykonania algorytmu w przypadku gdy trasy są dłuższe. Jest to spowodowane faktem że zasoby czasowe potrzebne na alokację oraz rozpoczęcie procedury przeszukiwania grafu z obydwu stron mogą być skompensowane przez rzeczywisty zysk uzyskany przez samą procedurę przeszukiwania. Czas obliczeń dla tras o długości 800-2000m jest w przypadku obydwu algorytmów prawie taki sam.

6.2. Porównanie wyników w stosunku do tras wyznaczonych przez Google Maps

7. Wnioski i możliwe dalsze usprawnienia