Sztuczna inteligencja i inżynieria wiedzy - Lista 1

Nikodem Świerkowski

May 1, 2024

Abstract

Algorytmy przeszukiwania grafów stanowią kluczowy obszar w informatyce, znajdując zastosowanie w wielu dziedzinach, w tym w analizie sieci komunikacyjnych. W tej pracy zaprezentowane zostaną implementacje algorytmów przeszukiwania grafów w celu znalezienia najoptymalniejszej podróży za pomocą komunikacji miejskiej Wrocław. Celem pracy jest zbadanie wydajności oraz trudności implementacji takowych algorytmów. W ramach wypełnienia celu badania, zaimplementowane zostały następujące algorytmy:

- Algorytm Dijkstry
- Algorytm A*
- Algorytm Tabu

Głównym zadaniem badania jest zaprezentowanie praktycznego spojrzenia na napisanie w języku programowania działających rozwiązań, przygotowany na rzeczywistych danych, tak aby wykorzystać ich pełen potencjał.

1 Wprowadzenie

Zadaniem algorytmów przeszukiwania grafu jest znalezienie najmniejniej kosztownej ścieżki prowadzącej z jednego węzła do innego. W przypadku komunikacji miejskiej rozpatrywanymi przez nas węzłami zostaną przystanki, przez które przejeżdzają tramwaje oraz autobusy MPK Wrocław, natomiast jako koszt może zostać przyjęty w zależności od potrzeby:

- czas, który należy poświęcić by pokonać trase
- liczba przesiadek, w czasie podróży

W zależności od zapotrzebowania należy wybrać odpowiednią funkcję kosztu. Sąsiedztwo w grafie komunikacji jest natomiast rozumiane, jako zdolność bezpośredniego przejechania z jednego przystanku, na drugi, a więc bez przystanków pośrednich. W celu przygotowania rozwiązania problemu został wykorzystany dokument "connection_graph.csv", który zawiera rozkład jazdy komunikacji miejskiej.

2 Implementacja grafu w programie

2.1 Reprezentacja grafu w programie

Aby przetwarzać przygotowane w pliku csv rzeczywiste dane, należy zaprezentować je w postaci grafu, czyli jako macierz lub jako powiązane referencjami obiekty. W opisywanym rozwiązaniu zdecydowano się na tą drugą opcje, z uwagi na pominięcie problemu przechodzenia po ogromnej macierzy, w której większość pól byłaby pusta - większość przystanków ma niewielu sąsiadów w skali liczby wszystkich przystanków we Wrocławiu. Konsekwencją działania na prawdziwych danych wynikających z rozkładu jazdy jest potrzeba identyfikacji przystanków. We Wrocławiu w przeciwieństwie do wielu miast, przystanki nie mają unikalnych nazw. Efektem tego jest np. istnienie dwóch przystanków o nazwie "Dworzec Główny". Teoretycznie możemy rozróżniać przystanki po współrzędnych geograficznych, należy jednak pamiętać że przystanki o nieunikalnej nazwie znajdują się blisko siebie, czego efektem jest niewielka różnica w lokalizacji. Należałoby także wyodrębnić, wtedy przystanki po dwóch stronach drogi, które różni jedynie zwrot kierunku przejeżdzających tramwajów - zazwyczaj domyślnie traktujemy je, jako

jeden ten sam przystanek. Stąd zostało przyjęte założenie o unikalności nazw przystanków. Jeśli istnieją dwa przystanki, o takiej samej nazwie to w grafie będzie reprezentowany jako jeden węzeł, o współrzędnych pierwszego znalezionego przystanku z tą nazwą.

2.2 Reprezentacja węzła i krawędzi

Węzeł został przedstawiony jako klasa "Node" o atrybutach:

- "stop" - nazwa przystanku
- "lat" - szerokość geograficzna przystanku
- "lon" - długość geograficzna przystanku
- "neighbours" - słownik, którego kluczem są nazwy przystanków - sąsiadów obecnego węzła, a wartością lista krotek, składających się z numeru linii, czasu odjazdu, czasu przyjazdu
- "lines" - słownik, którego kluczem są nazwy linii, przejeżdzające przez ten przystanek, a wartością lista krotek, składających się z nazwy sąsiada, do którego prowadzi ta linia, czasu odjazdu, czasu przyjazdu. Krotkę tą będę w dalszej części nazywał "przejazdem".

Może wydawać się niepotrzebne dublowanie informacji w atrybucie "neighbours" i "lines", ale okaże się to przydatne w momencie implementacji różnych funkcji kosztu.

2.3 Stworzenie grafu

W celu utworzenia grafu o opisanej strukturze, należało wczytać plik "connection_graph.csv". Została użyta do tego biblioteka "pandas". Każdy rekord został przeanalizowany. Jeśli węzły, o występujących w rekordzie nazwach nie istniały, to zostawały tworzone. Następnie do pierwszego przystanku, zostawał dodawany nowy sąsiad wraz z dostępnym przejazdem lub do listy istniejącego sąsiada zostawał dodany nowy przejazd. Podobnie zostało to zrealizowane w przypadku atrybutu "lines". Po zakończeniu wczytywania, każdemu sąsiadowi i każdej linii w każdym węźle posortowano listę przyjazdów, aby szybciej móc znajdywać następny przejazd między przystankami.

2.4 Serializacja grafu

Oczywiście stworzenie się takiego grafu jest czasochłonne, dlatego w celu przyspieszenia działania programu został on zserializowany do postaci pliku: "node_dict.pkl". Za każdym więc razem, kiedy ktoś odwołuje się do stworzonego grafu jest on wczytywany z pliku, jeśli takowy istnieje. Jeśli nie, powtarzana jest procedura tworzenia grafu, wraz z jego serializacją. Czas wczytania zserializowanego pliku jest stały, a także na tyle mały, że zostanie pominięty przy obliczaniu czasu działania algorytmów.

3 Implementacja algorytmu Dijkstry

3.1 Zapisywanie ścieżki przejścia po grafie

W celu implementacji algorytmu Dijkstry, należy zdecydować w jaki sposób będzie przechowywana ścieżka przechodzenia po grafie. Początkowo wybrany został zapis z pomocą słownika, o kluczu-węźle i wartości, będącej innym węzłem, z którego dostano się do klucza. Szybko okazało się, że prowadzi to do zapętlenia w przypadku próby odtworzenia ścieżki do postaci listy. Czasami algorytm decydował, że poprzednikiem zostanie węzeł, który odwiedziliśmy już w drodze. W ten sposób odkryta ścieżka zostawała utracona. Rozwiązanie problemu było kosztowne pod względem optymalizacji pamięci - dla każdego węzła, zostaje teraz przechowywana lista przystanków, które zostały odwiedzone, by do niego trafić. W ten jednak sposób, odkrywając inną ścieżke, nigdy nie tracimy dotychczasowych rozwiązań.

3.2 Zapisywanie kosztu dojazdu do poszczególnych węzłów

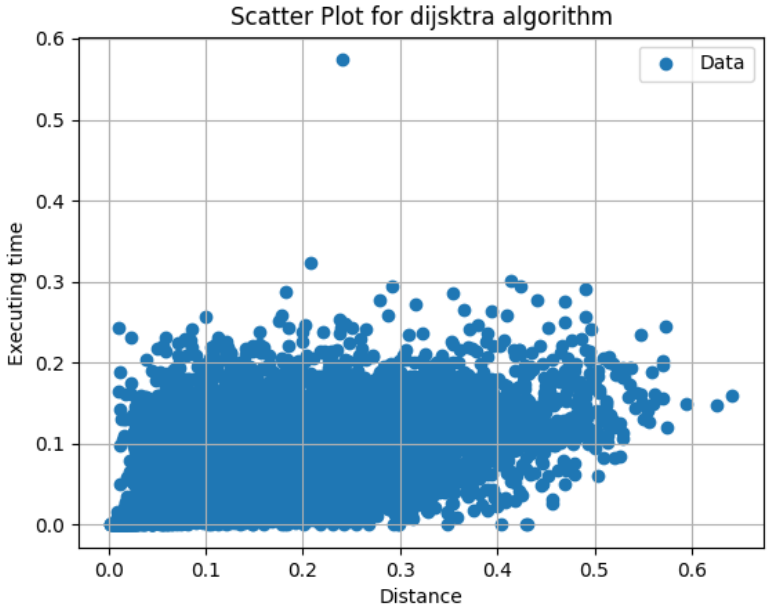
Zapamiętywanie najmniejszego kosztu, zostało zrealizowane podobnie - jako słownik węzła do minimalnego kosztu dotarcia do niego od punktu startowego. W ten sposób łatwo możemy sprawdzić, jaki był dotychczasowy koszt i czy znalezione alternatywne przejście do przystanku jest bardziej opłacalne. Koszt jest też oczywiście priorytem kolejki, która przekazuje nam kolejne przystanki.

3.3 Funkcja kosztu

Stworzona implementacja operuje jedynie na koszcie, jako czasie podróży. Możliwe jest więc wyliczenie kosztu, jako różnicy między godziną pojawienia się na przystanku, a godziną dojazdu na kolejny przystanek. Stąd funkcją celu została "travel_between" z pliku "exercise1_additional_function.py", który przechowuje funkcje, z których korzystają inne algorytmu z zadania pierwszego. Zadaniem tej funkcji jest znalezienie najszybszego przejazdu, między obiektami klasy Node. Aby tego dokonać wykorzystuje wyszukiwanie binarne - jest to możliwe, ponieważ w czasie tworzenia grafu, każdemu węzłowi posortowaliśmy przejazdy dla każdego sąsiada. Wybierany jest przejazd, który prowadzi do przekazaniego przystanku oraz jest to pierwszy możliwy odjazd po przekazanym czasie początkowym. Przyjęto tu uproszczenie. Ponieważ opieramy się na odjeździe, ignorujemy fakt, że pomimo tego że pierwszy możliwy przejazd, nie musi wcale dojechać szybciej do docelowego przystanku, niż kolejny przejazd. Łatwo możemy wyobrazić sobie sytuacje, że pierwszy środek komunikacji, który przyjeżdza na przystanek to autobus, który jedzie kilka minut dłużej, niż przyjeżdzający za minute tramwaj. Należy jednak pamiętać, że w obrębie danych, które posiadamy nie jest to zjawisko występujące bardzo często, a nawet gdy zazwyczaj się wydarzy spowoduje różnicy maksymalnie kilku minut względem najoptymalniejszego rozwiązania, który to czas i tak może być niwelowany przez postój w oczekiwaniu na przesiadkę. Ważnym zagadnieniem w przypadku funkcji kosztu - czasu, jest ustalenie zachowania modelu w momencie, gdy trasa rozpoczeła się innego dnia niż zakończyła. W takim przypadku nie możemy po prostu odjąć czasu końcowego od początkowego - należy uwzględnić że dzień dobiegł końca. W tym celu, jeśli godzina końca trasy jest wcześniejsza niż godzina początkowa, do różnicy będziemy dodawać dwadzieścia cztery godziny. Istotnym problem w większym grafie mogłaby być podróż na przestrzeni kilku dni, jednak w przypadku naszego zbioru danych nie istnieje możliwość wybrania takich dwóch przystanków, aby optymalny czas podróży między nimi trwał więcej niż doba, więc nie musimy rozwiązywać tego problemu. Wróci on jednak w przypadku implementacji algorytmu tabu search.

3.4 Wynik rozwiązania

Aby zweryfikować skuteczność rozwiązania, został stworzony skrypt, który wielokrotnie mierzył czas wykonywania się algorytmu. Skrypt wybierał losowo dwa różne przystanki, a następnie wyznaczał trasę między nimi. Efektem jest widoczny wykres:



Wykres jest mało czytelny. Widzimy że czas (wyrażony w sekundach) jest zróżnicowany dla przystanków oddalonych o taką samą odległość. Stąd wynik został uśredniony dla przystanków oddalonych o równą odległość. Aby zoobrazować lepiej zależności, argumenty zostały podzielone na tzw. kubełki, o stałej wielkości np. każdy rekord, w których odległość między dwoma przystankami była w zakresie od 0 do 0.01 został zgrupowany do jednego kubełka. Następnie każdy rekord, z odległością między 0.01 a 0.02 do kolejnego kubełka itd. Dla każdego kubełka jako wartość wzięta została mediana wartości. Należy jednak pamiętać, że pomiar został przygotowany w warunkach nielaboratoryjnych, stąd gdy mówimy o tak małym czasie, bardzo na pomiar wpływa działanie systemu operacyjnego, który wykonuje dodatkowe operacje w tle.

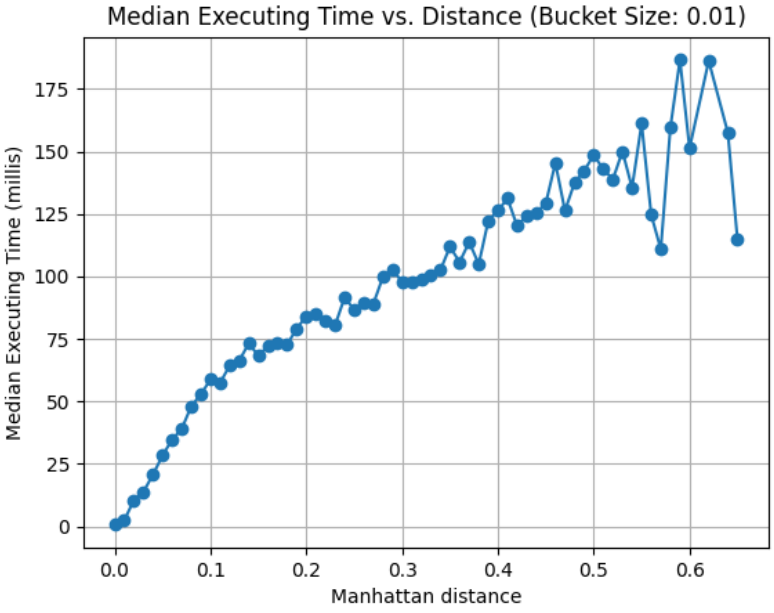


Figure 1: Wykres mediany czasu wykonania programu w zależności od dystansu

Widzimy że krzywa przypomina z lekka funkcje logarytmiczną, pozostawałbym jednak sceptyczny

co do tej kwestii. Algorytm użyty ma pesymistyczną złożoność czasową, około $n^2 * log(n)$ z uwagi na użycie wyszukiwania binarnego. Aby dobrze sklasyfikować tą krzywą należałoby powtórzyć badanie, na przygotowanym środowisku testowym, dla większych wartości czasu wykonywania, tak aby móc dokładnie przeanalizować tą krzywą. Należy też pamiętać, że przyjęta zależność odległość przystanku do czasu wykonania programu nie należy traktorać jako dobry model reprezentujący złożoność algorytmu, bardziej jako reprezentacji zmiany działania programu w zależności od wyboru odległych przystanków.

4 Implementacja algorytmu A* (kryterium czasu)

4.1 Różnice miedzy algorytmem A* a algorytmem Dijkstry

Główna różnica dotyczy implementacji przechowywania kosztu dotarcia do węzła. Teraz zamiast prostego słownika klucz wartość, jako węzeł do kosztu, został użyty słownik węzęł do krotki, składającej się z wartości G, H oraz F tego przystanku, rozumiach w sposób następujący:

- G - czas, jaki zajmuje przyjechanie do tego przystanku.
- H - czas od tego węzła do węzła końcowego, wyznaczony za pomocą heurystyki
- F - suma G oraz H

Każdy z tych trzech parametrów w implementacji jest obiektem klasy timedelta. Tym razem zamiast kosztu, oznaczającego wartość czasu dotarcia do węzła, wykorzystujemy koszt rozumiany, jako łączny czas przejazdu (od przystanku początkowego do końcowego), przez ten węzeł, czyli wykorzystujemy wartość F przystanku.

4.2 Wykorzystane heurystyki

4.2.1 Heurystyka oparta na prędkości

Najprostrzym sposobem próby przewidywania czasu podróży jest próba oszacowania prędkości. Z tego powodu po utworzenia grafu, zostaje obliczona wartość "tram_speed", która jest maksymalną prędkością, jaką może jechać tramwaj między dwomą sąsiadującymi przystankami. Stała ta jest wyliczona w następujący sposób: Dla każdego sąsiada wszystkich przystanków, wyliczana jest prędkość rozumiana jako odległość manhattan między przystankami podzielona przez czas przejazdu. Następnie spośród wszystkich wyliczonych prędkości wybieramy największą. Jest to o tyle istotne, by była to największa prędkość, tak aby stworzona heurystyka była optymistyczna.

4.2.2 Heurystyka oparta na modelu regresji

Alternatywnym sposobem przewidywania czasu trasy może być zbudowanie prostego modelu, którego zadaniem będzie estymacja czasu przejazdu. Model zostałby zserializowany i wczytywany przy próbie użycia algorytmu. Z uwagi na stworzenie heurystyki optymistycznej, wynik modelu będzie korygowany przez średni błąd absolutny, tak aby nigdy nie zaniżał prawdziwej wartości. Najlepszym rozwiązaniem byłoby tutaj zwracanie przez model przedziału ufności - wtedy mielibyśmy prawie stuprocentową pewność, że pobrana wartość jest niemniejsza niz prawdziwa.

Na podstawie wygenerowanych danych na potrzeby wniosków na temat algorytmu dijkstry możemy spróbować znaleźć dane, które pozwolą nam estymować czas.

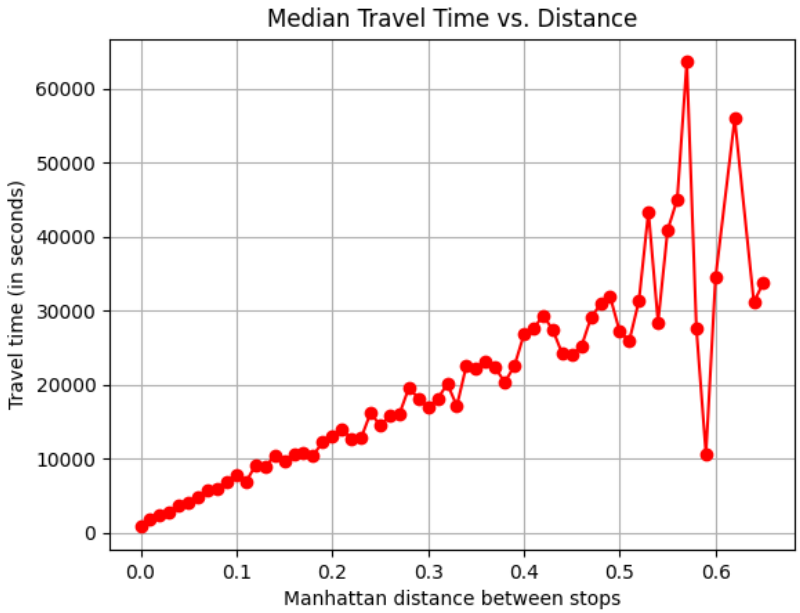
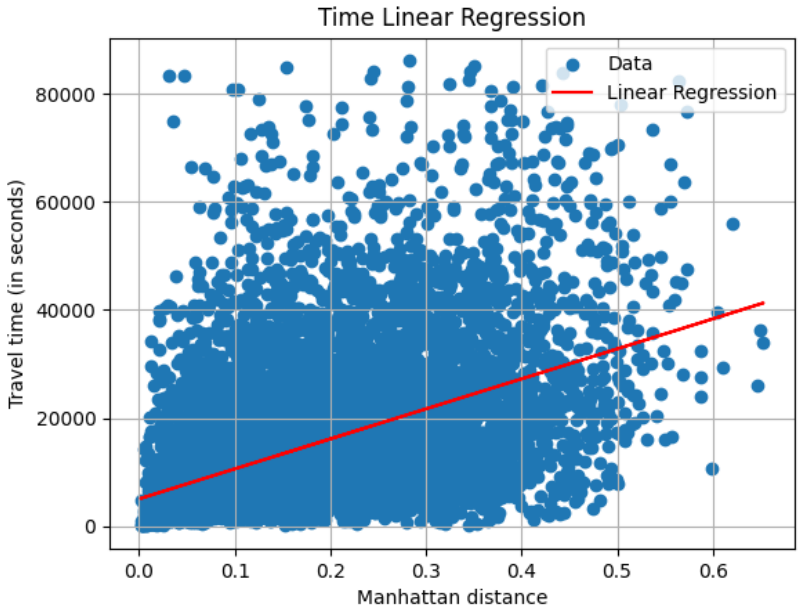


Figure 2: Powyższy wykres prezentuje zależność mediany czasu przejazdu od odległości manhattanowej przystanku początkowego i końcowego.

Zależność czas od dystansu jest niemal idealnie liniowa. Atrybut odległość, może więc zostać użyty do budowy modelu. Z uwagi na obecność jedynie dwóch wartości i zależności liniowej, zdecydowano o utworzeniu modelu regresji liniowej, który prawdopodobnie najlepiej się sprawdzi do przedstawionego problemu.

Graficzna reprezentacja utworzonego modelu:



Wyniki modelu	
Średni błąd bezwzględny (MAE)	8281.76614264802
Średni błąd kwadratowy (MSE)	136564946.42173597
Współczynnik determinacji R^2	0.2299710317397886

Table 1: Statystyki utworzonego modelu

4.3 Potencjalna optymalizacja - podejście kombinacji heurystyk

Szansą na uzyskanie lepszego wyniku może być wykorzystanie kombinacji heurystyk, czyli użycie kilku algorytmów heurystycznych, a następnie wybranie spośród nich heurystyki o maksymalnej wartości. Istnieje w ten sposób szansa, że tak stworzona heurystyka będzie szybciej znajdowała rozwiązanie, jednak sam fakt potrzeby wyliczania obu heurystyk zwiększy czas oczekiwania na wynik.

4.4 Wyniki rozwiązań

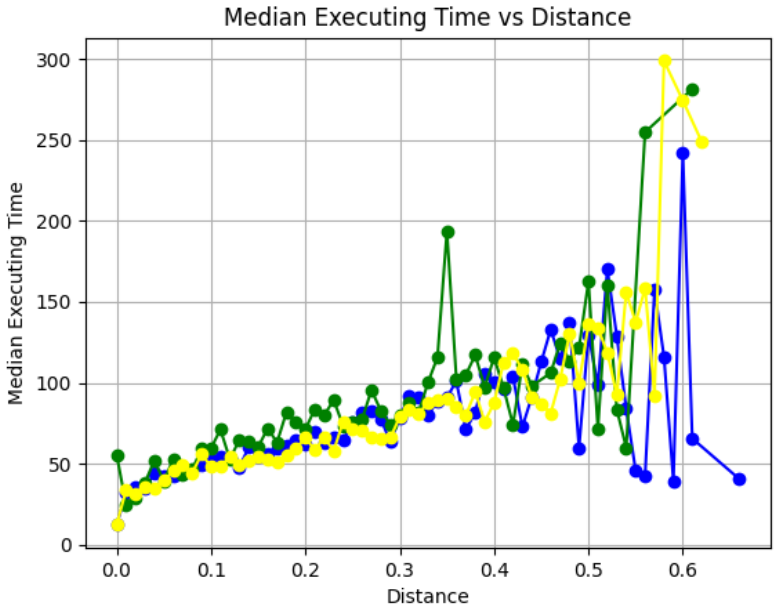


Figure 3: Porównanie działania różnych heurystyk z użyciem algorytmu A*. Kolor niebieski reprezentuje model regresji liniowej, zielony wykorzystanie prędkości, a żółty to wynik heurystyki kombinacji

Po porównaniu heurystyk możemy wywnioskować, że ich wyniki są bardzo zbliżone. Każda z tych heurystyk, mogłaby być używana skutecznie w algorytmie A*. Najgorzej poradziła sobie najprostrza heurystyka, czyli prędkość, jednak zważywszy na prostote jej wyliczenia, wciąż jest świetną alternatywą jako heurystyka, wyraźnie przyspieszającą działanie algorytmu. Trudniej natomiast stwierdzić, czy lepiej poradził sobie model regresji, czy też kombinacja heurystyk. Aby to roztrzygnąć należałoby sprawdzić, jak często kombinacja opierała swoją estymacje na modelu regresji. Po podobieństwu krzywych możemy wnioskować, że było to częste zjawisko. Nawet jeśli przyjmiemy, że kombinacja heurstyk poradziła sobie lepiej, to różnice są na tyle mało, że to podejście okazało się niezbyt satysfakcjonujące.

5 Implementacja algorytmu A* (kryterium przesiadek)

5.1 Implementacja atrybutów G, H i F

Z uwagi na zmiane rodzaju kosztu, G H i F nie mogły dłużej pozostać obiektem klasy timedelta.

- G - zostało liczbą całkowitą, oznaczającą liczbę przesiadek odbytych do momentu dotarcia do przystanku
- H - zostało liczbą zmiennoprzecinkową, oznaczającą estymacje liczby przesiadek, jakie odbyją się do momentu dotarcia do węzła końcowego
- F - zostało liczbą zmiennoprzecinkową, sumą G i H

Nieintuicyjny może wydawać się typ H - wkońcu nie można odbyć np. półtora przesiadki. Jednak w zależności od typu użytej heurystyki, może ona zwracać liczbę zmiennoprzecinkową, stąd decyzja o niezaokrąglaniu jej.

Obliczanie atrybutu G jest następujące: Jeśli rozpatrywany węzeł jest sąsiadem, poprzedniego przystanku, to w zależności od tego czy wymaga przesiadki do innej linii, czy też nie, dodaj jeden lub zero do wartości G poprzedniego sąsiada. Jeśli nie jest sąsiadem to G niech będzie nieskończonością (w opisywanym rozwiązaniu maksymalną wartością liczby całkowitej).

Jeśli podróż do węzła końcowego z węzła obecnego jest możliwa to zamiast, jak w przypadku kryterium czasu wybierać pierwszy możliwy odjazd z przystanku do następnego przystanku, wprowadzono modyfikacje poprawiającą działanie algorytmu. Mianowicie spośród linii, które przejeżdzają przez aktualny przystanek, szukamy takiej, która przejedzie zarówno przez rozpatrywany kolejny węzeł, ale także węzeł, który oznacza koniec podróży. Jeśli takowa nie została znaleziono, to standardowo wyszukiwany jest pierwszy możliwy odjazd. Jeśli zostanie znaleziona to wyszukujemy binarnie najszybszy odjazd tej linii z tego przystanku i to nim trafiamy do kolejnego przystanku.

5.2 Heurystyka

Przeciwnie do kryterium czasu, nie mamy już tak prostego sposobu na estymacji jak wyznaczenie prędkości. Stąd decyzja o użyciu modelu regresji liniowej. Podobnie jak w przypadku kryterium czasu model zostanie wytrenowany na podstawie korelacji między odległością manhattanową i tym razem nie czasem, a liczbą przesiadek.

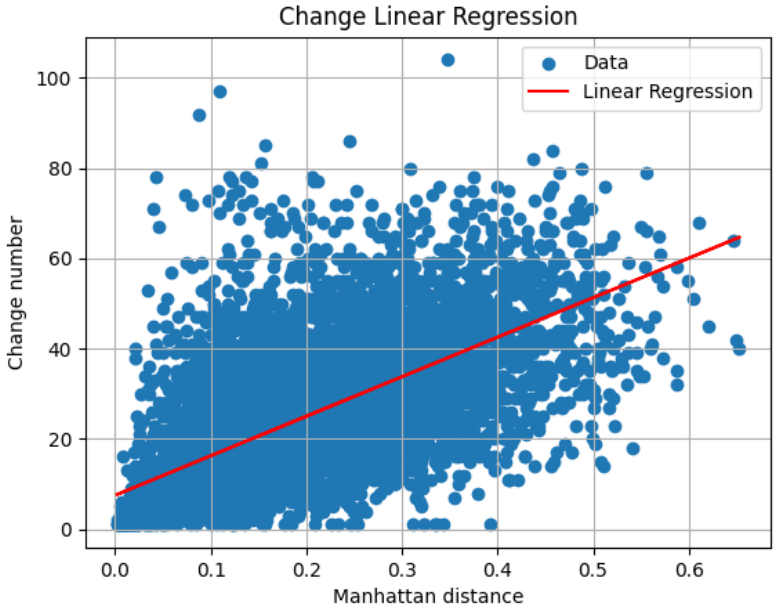


Figure 4: Graficzna reprezentacja modelu, na zebranych tle danych

Wyniki modelu	
Średni błąd bezwzględny (MAE)	7.063238298945144
Średni błąd kwadratowy (MSE)	101.5090038697551
Współczynnik determinacji R^2	0.47344964816370994

Table 2: Statystyki utworzonego modelu

5.3 Wynik rozwiązania

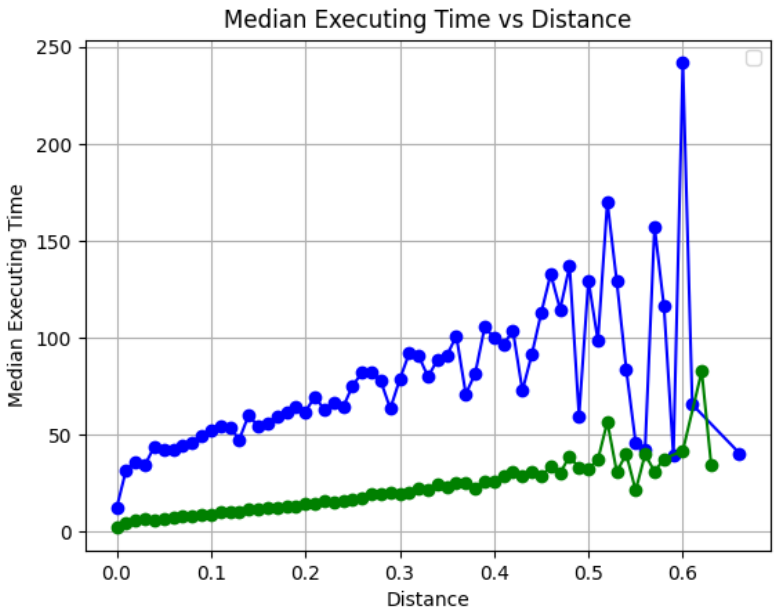


Figure 5: Prezentacja i porównanie działania modelu regresji dla kryterium czasu (niebieski) oraz modelu dla kryterium przesiadek (zielony)

Wyniki dla rozwiązania opartego o kryterium przesiadek jest szybsze niż rozwiązania opartego na kryterium czasu. Może to być spowodowane specyfiką kosztu. W przypadku czasu w każdej iteracji tworzone było wiele obiektów deltatime, obciążając działanie komputera, dodatkowo dla każdego sąsiada za każdym razem następowało wyszukiwanie odpowiedniego przejazdu. W kryterium przesiadek w momencie znalezienia linii, która łączy przystanek obecny i końcowy, algorytm nie musiał tracić czasu na przeszukiwanie listy przejazdów. Widzimy że krzywa jest takiego samego typu, co potwierdza, że mimo zmiany typu kosztu jest to ten sam algorytm.

6 Implementacja algorytmu Tabu

6.1 Wykorzystanie algorytmu Tabu w kontekście komunikacji miejskiej

Znalezienie najoptymalniejszej trasy, przez wybrane przystanki jest nietrywialnym problemem. Sam fakt wykorzystania algorytmu A* do wyznaczenia trasy między kolejnymi przystankami, nie rozwiązuje problemu, ponieważ wymagałoby użycia na algorytmu wyznaczania ścieżki między dwoma przystankami na każdej możliwej konfiguracji odwiedzania przestanków. Jest to bardzo kosztowne. Stąd do rozwiązania problemu zdecydowano, aby zaimplementować algorytm Tabu.

6.2 Rozwiązanie naiwne

Wprowadzone rozwiązanie opiera się na założeniu, że czas podróży między dwoma przystankami jest stały w czasie. Nie jest to dokońca prawda, ponieważ częstotliwość przejazdów między dwoma przys-

tankami zmienia sie w czasie. Należy jednak zwrócić uwage, że jeśli koszt przejazdu między przystankami 'A' i 'B' jest najkrótszy spośród wszystkich innych przejazdów z zadanej listy przystanków do odwiedzenia o godzinie 12:00 to prawdopodobnie czas dojazdu z 'A' do 'B' o godzinie 22:00 i tak będzie najkrótszy spośród wszystkich innych przejazdów z zadanej listy przystanków do odwiedzenia o godzinie 22:00. Wpierw generujemy wszystkie możliwe permutacje przekazanej listy przystanków, a następnie za pomocą algorytmu A* wyliczamy koszt podróży dla następującej permutacji. Za każdym razem zapisujemy iteracje dla której koszt był jak dotychczas najmniejszy. Czynność jest powtarzana, gdy przeiterowujemy się po wszystkich permutacjach lub do momentu dotarcia do T-tej iteracji, gdzie T to przekazana liczba oznaczająca maksymalną liczbę rozpatrywanych iteracji. Już na tym etapie jednak widzimy, że znalezienie optymalnego rozwiązania w liczbie T iteracji jest uzależnione od kolejności permutacji. W tym rozwiązanie nie dbamy w żaden sposób o ich kolejność i wymaga to poprawy. Natomiast w celu minimalizacji liczby użytych wywołań funkcji implementującej algorytm A*, memoizujemy wyniki wyliczonego kosztu drogi między dwoma przystankami, za pomocą słownika.

6.3 Rozwiązanie dokładniejsze

W tym podejściu nie generujemy wszystkich możliwych permutacji przekazanej listy przesiadek do odwiedzenia. Z każdą iteracją będziemy tworzyć nowe rozwiązanie, które jeszcze nie zostało rozpatrywane (które nie jest tabu). Orginalna lista zostaje traktowana jako lista główna. Z każdą iteracją uruchamiamy funkcje "next_lexicographic_permutation", która jest implementacją algorytmu generowania leksykograficznej permutacji. Zapewnia nam ona, że za każdym razem gdy odwołamy się do niej zwróci nam kolejną permutacje listy. Wymaga jednak posortowania listy wcześniej. W ten sposób upewniamy się że w nieskończonej liczbie kroków zawsze znajdziemy najlepszą ścieżke. Jednak zazwyczaj jednak będziemy ograniczać działanie algorytmu za pomocą maksymalnej dopuszczalnej liczby iteracji. Aby zwiększyć prawdopodobieństwa natrafienia na najlepszą ścieżke, przy ograniczonej liczbie iteracji tworzymy tzw. populacje, czyli liste ścieżek, które przeciwnie do głównej listy z każdą iteracją będą losowo zamieniać dwa elementy w swoim rozwiązaniu (w taki sposób by nie naruszyć tabu).

Następnie dla list w populacji i głównej listy następuje wyliczenie łącznej odległości do przebycia (to jest sumy odległości wszystkich kolejnych przystanków w liście). Z wyników wybieramy ten, o najmniejszej odległości, wykorzystując wiedzę, że istnieje zależność w zbiorze danych między odległością przystanków, a kosztem przejazdu. Dla niego z pomocą algorytmu A* wyliczamy koszt przejazdu i jeśli jest mniejszy niż dotychczasowy najmniejszy koszt, zostaje zapisany. W tym podejściu staramy się generować permutacje bazując na przewidywanym koszcie.

7 Podsumowanie

Realizacja algorytmów przeszukiwania w prawdziwym kontekście pozwala inaczej spojrzeć na trudności w implementacji. Umożliwia to na skorzystanie z wiedzy dotyczącej problemu, ale też pokazuje trudność przeniesienia abstrakcyjnego modelu matematycznego, na płaszczyzne świata rzeczywistego. Dane, które nie zostały przewidziane, powodujące błędy, takie jak przystanki, na które da się tylko przyjechać, a nie wyjechać albo symbolika oznaczania czasu, która dopuszcza istnienie godzin większych niż dwadzieścia cztery. Zrozumienie danych jest kluczowe w celu tworzenia programów odpornych na błędy, ale też takich, które mają wykorzystywać w sobie heurystyke. Przykład implementacji algorytmu A*, o kryterium czasowym, pokazuje że rozwiązanie oparte na modelu statystycznym, czyli takim który opiera swoje działanie na znalezieniu korelacji w zbiorze danych, może być zarówno dokładniejszy oraz szybszy niż alternatywne wykorzystane heurystyki. Tak samo zrozumienie podstawowej zależności i stworzenie w ten sposób stałej uznawanej za prędkość także okazało się zwracać przyzwoite wyniki więc mógłby to być preferowany sposób heurystyki, gdy chcemy minimalizować zasoby pamięciowe (nie wymaga przetrzymywania modelu w programie).

Ciekawym wnioskiem na podstawie wykonanych algorytmów jest wpływ funkcji kosztu na działanie algorytmu. Kryterium przesiadek, które początkowo wydawało się zdecydowanie trudniejsze, okazało się znacznie szybsze i prostrze w implementacji, pozwalające ominąć problem właściwego liczenia czasu. Każde kryterium wymaga oddzielnego podejścia i zrozumienia.

W przypadku działania pojawia się też trudność w przypadku testowania modelu na prawdziwych, niegenerowanych danych. Dane testowe, wymagałaby ręcznego przygotowania, pełną pewnością co do

rozwiązań co w przypadku drugiego zadania, które w najbardziej pesymistycznej wersji zamienia się w problem komiwojażera okazuje się niemal niemożliwe w krótkim przedziale czasowym.

W przyszłości przedstawiona implementacja mogłaby zostać znacząco poprawiona. Algorytmy o kryterium czasowym bazowałyby na czasie w sekundach, zamiast obiektach datetime, co znacząco poprawiłoby ich efektywność. Algorytm A* o kryterium przesiadek, powinien też zostać zaimplementowany z dodatkową heurystyką wyliczaną podobnie do prędkości, jednak zamiast dzielenia odległości na czas, współczynnik byłby wyliczany po przez podzielenia odległości na liczbę przesiadek, na podstawie danych zebranych z działania algorytmu dijkstry.

Dobrze byłoby też wytrenowanie różnych rodzajów modeli, tak aby ustalić, który sprawdza się najlepiej do powierzonego zadania. Być może należałoby spróbować znaleźć inne zależności w zbiorze danych, tak aby umożliwić zbudowanie wielowymiarowego zbioru argumentów. Potencjalnie czas przejazdu oraz liczba przesiadek jest uzależniona od godziny, w której wyszkiwana jest trasa. Częstotliwość przyjazdów jest zmienna wciągu dnia.

Interesujące wydaje się także zaimplementowanie algorytmu tabu, z użyciem algorytmu genetycznego, który decydowałby o kolejnych przesunięciach, permutacjach. Prawdopodobnie znacząco zwiększyłoby to efektywność działania algorytmu.