Algorytmy przeszukiwania w poszukiwaniu połączeń między przystankami

Karolina Łukasik

21 kwietnia 2023

Spis treści

1 Wstęp

Poniższy raport zawiera teoretyczne opisy sposobów, metod i algorytmów za pomocą których podeszłam do problemu poszukiwania połączeń między przystankami. Jest on uzupełnieniem **notatnika Jupyter**, w którym dokładnie okomentowałam i opisałam funkcje wraz przykładami ich użycia.

Zaczniemy od przyjrzenia się dostarczonym danym i przedstawienia ich za pomocą grafu. Pierwszym omawianym algorytmem będzie klasyczny algorytm Dijkstry, za pomocą którego znajdziemy najszybsze połączenie między przystankami. Następnie rozszerzymy go o estymację kosztu przejścia z wierzchołka do celu, otrzymując tym samym algorytm A*. Dodamy możliwość wyboru kryterium: czasu lub liczby przystanków w zależności od preferencji użytkownia. By Algorytm A* w poszukiwaniu połączenia z najmniejszą ilością przystanków dawał satysfakcjonujące wyniki, zmodyfikujemy sposób patrzenia na graf i tym samym mechanizm przeszukiwania sąsiednich wierzchołków. Na koniec zastanowimy się, jak znaleźć najkrótszą ścieżkę między listą przystanków. Do tego problemu, zwanego inaczej problemem komiwojażera, podejdziemy za pomocą przeszukiwania Tabu.

2 Reprezentacja danych za pomocą grafu

Dzięki podstawowej analizie jesteśmy w stanie wybrać najlepszą strategię radzenia sobie z naszym zbiorem danych. Pierwsze początkowe rekordy prezentują się następująco:

	Unnamed: 0.1	Unnamed: 0	company	line	departure_time	arrival_time	start_stop	end_stop	start_stop_lat	start_stop_lon	end_stop_lat	end_stop_lon
0	0	0	MPK Autobusy	A	20:00:00	20:01:00	KRZYKI	Sowia	51.074884	17.006569	51.073793	17.001845
1	1	1	MPK Autobusy	A	20:01:00	20:02:00	Sowia	Chłodna	51.073793	17.001845	51.075122	16.996671
2	2	2	MPK Autobusy	A	20:02:00	20:03:00	Chłodna	Wawrzyniaka	51.075122	16.996671	51.078074	16.998202
3	3	3	MPK Autobusy	A	20:03:00	20:05:00	Wawrzyniaka	Rymarska	51.078074	16.998202	51.079323	16.991258
4	4	4	MPK Autobusy	A	20:05:00	20:06:00	Rymarska	RACŁAWICKA	51.079323	16.991258	51.077395	16.983938

Rysunek 1: Ramka danych

W naszych programach nie wykorzystujemy pierwszych dwóch kolumn, ani informacji na temat firmy przewoźnika. Kluczowe są natomiast następujące informacje:

- Liczba rekordów: 273471
- Liczba przystanków: 934

2.1 Implementacja grafu

Dane reprezentowane są za pomocą grafu. Wykorzystujemy tu podstawowe własności programowania zaorientowanego obiektowo. Wierzchołki grafu to poszczególne przystanki z zachowaną informacją na temat wszystkich połączeń wychodzących z danego przystanku. W implementacji korzystamy z **zagnieżdżonego słownika**, w którym każdy klucz oznacza nazwę przystanku, a wartość jest kolejnym słownikiem z kluczami będącymi jego sąsiadującymi przystankami. W końcu wartość zagnieżdżonego słownika jest listą wszystkich połączeń z danego przystanku do jego sąsiada. Połączenie samo w sobie reprezentowane jest jako lista, trzymając się konwencji:

$$Polaczenie = [linia, czasodjazdu, czasprzyjazdu]$$

Poza tym graf posiada informację na temat liczby wierzchołków, która równoważna jest liczbie przystanków w naszym zbiorze danych czyli 934. Posiada również słownik z zapisanymi współrzędnymi geograficznymi dla każdego przystanku.

Czas przekształcony został z napisu w formie H:M:S na liczbę minut jaka minęła tego dnia od północy do podanego czasu.

Rysunek 2: Przykłady reprezentacji danych

3 Algorytm Dijkstry

Za pomocą Algorytmu Dijkstry możemy wyszukiwać najkrótszą trasę z przystanku A do B w oparciu o kryterium czasu. Implementację rozpoczynamy od zainicjalizowania trzech słowników:

- *dist dict* reprezentujący odległość (w minutach) każdego przystanku (wierzchołka) od przystanku początkowego. Wartości inicjalizujemy jako 0 dla przystanku początkowego i nieskończoność dla wszystkich pozostałych.
  UWAGA: W naszych algorytmach nieskończoność reprezentowana jest jako 1e7, ponieważ wszystkie pojawiające się wartości w obliczeniach są znacznie mniejsze. Przypomnijmy, że liczba minut w dobie wynosi 1440.

- *prev* przechowujący informacje o sposobie w jaki dostaliśmy się na dany przystanek na drodze poszukiwania najkrótszej drogi do tego przystanku. Wartością dla każdego klucza (przystanku) w tym słowniku jest lista trzymająca się konwencji:

$$prev[Przystanek] = [Poprzednik, Linia, CzasOdjazdu, CzasPrzyjazdu]$$

- *calc finished* przechowuje informacje, czy najkrótsza ścieżka dla danego wierzchołka została już wyliczona, czyli pozostanie niezmienna. Inicjalizujemy słownik wartościami False dla każdego wierzchołka.

Dodatkowo inicjalizujemy zbiór *visited not finished* przechowujący nazwy przystanków, które już odwiedziliśmy ale wyliczanie dla nich najkrótszej ścieżki jeszcze się nie skończyło. Może się bowiem zdarzyć że droga przez innego z jego sąsiadów będzie krótsza. Korzystanie ze zbioru jako struktury danych do przechowywania tych informacji jest bardzo wygodne ze względu na fakt, iż duplikaty nie są zapisywane, więc nie musimy za każdym razem sprawdzać czy wierzchołek już znajduje się w secie.

Główna część algorytmu polega na przejściu pętli przeszukiwania, aż nie dotrzemy po drodze do przystanku końcowego. "Po drodze" jest tu dobrym sformułowaniem, ponieważ algorytm w żaden sposób nie koncentruje się na dotarciu do przystanku, na którym nam zależy, tylko przeszukuje wierzchołki w każdą stronę. Tworzy w ten sposób krok po kroku drzewo z najkrótszymi ścieżkami od źródła do każdego z wierzchołków. Gdybyśmy nie zatrzymali algorytmu w momencie dotarcia do celu, policzyłby nam to drzewo do końca. Łatwo się domyślić, że w najgorszym wypadku przejdziemy pętlę tyle razy ile jest wierzchołków.

W samej pętli wybieramy najpierw przystanek z najmniejszą odległością od przystanku początkowego, spełniający warunek bycia w zbiorze *visited not finished*. Do wyznaczenia tego wierzchołka posłuży nam funkcja *minDistanceNode(distdict, visitednotfinished)*. Jeśli wyznaczony został przystanek końcowy, jest to czas na zakończenie działania algorytmu i zwrócenie słownika z poprzednikami *prev* oraz czasu przejazdu (wartości funkcji kosztu), jaką algorytm wyliczył z przystanku początkowego do końcowego.

Jeśli jednak wyznaczyliśmy właśnie najkrótszą ścieżkę dla innego wierzchołka, dokonujemy następujących kroków:

- Zachowujemy informacje o zakończeniu obliczeń dla tego wierzchołka: *calc_finished[node] = True* oraz *visited_not_finished.remove(node)*
- Pobieramy czas, kiedy znaleźliśmy się na tym przystanku *arr_time_at_node = prev[node][3]*
- Przechodzimy w pętli po wszystkich sąsiadach przystanku, czyli przystankach do których ma jakiekolwiek połączenia. Jeśli najkrótsza ścieżka do danego sąsiada nie została jeszcze obliczona, podejmujemy następujące kroki:
  - Wyszukujemy najszybsze połączenie z przystanku do jego sąsiada. Korzystamy tutaj z funkcji *earliestConnection(graph,node,neighbour,arr_time_at_node)*, która zwróci nam linię, czas odjazdu i przyjazdu, czyli najszybsze połączenie między przystankami o danej godzinie. Jeśli jednak takiego połączenia już nie ma (wszystkie były o wcześniejszej godzinie), funkcja zwróci wartość -1 i przejdzie do kolejnego sąsiada.
  - Jeśli jednak znaleźliśmy połączenie, to dodajemy sąsiada do zbioru odwiedzonych przystanków (jeśli już tam jest to nic się nie dzieje). Wyliczamy jaka byłaby długość przejazdu dla sąsiada jeśli zmieniłby trasę na trasę naszego wierzchołka plus czas przejazdu z naszego przystanku do siebie. Jeśli trasa ta jest szybsza bądź wierzchołek nie miał jeszcze wyznaczonej trasy to aktualizujemy dane na temat jego poprzednika i połączenia w jaki sposób się do niego dostaliśmy.

Kontynujemy działanie algorytmu aż do uzyskania najkrótszej trasy do przystanku końcowego. Wy wydobyć trasę i ją wydrukować, przechodzimy rekurencyjnie od przystanku końcowego do początkowego. Pomoże nam w tym celu funkcja *printRoute(prev,target,src,dep_time,short=False)*. Możemy wybrać, czy zależy nam na drukowaniu pełnej wersji z każdym przejechanym przystankiem, czy tylko wyświetlaniu przystanków na których zmieniamy linię, lub zaczynamy/kończymy trasę.

```
Final destination:  most Grunwaldzki  reached by  12 : 37

Urząd Wojewódzki (Impart) - 4 - 12 : 36
GALERIA DOMINIKAŃSKA - 4 - 12 : 33
Świdnicka - 4 - 12 : 30
Zamkowa - 4 - 12 : 28
Narodowe Forum Muzyki - 4 - 12 : 27
pl. Legionów - 4 - 12 : 25
Kolejowa - 4 - 12 : 23
Grabiszyńska - 4 - 12 : 22
Pereca - 4 - 12 : 21
Stalowa - 4 - 12 : 19
pl. Srebrny - 4 - 12 : 18
Bzowa (Centrum Zajezdnia) - 4 - 12 : 17
Hutmen - 4 - 12 : 16
FAT - 4 - 12 : 15

FAT reached at 12:13 You have to take next bus/tram now!

Ostrowskiego - 107 - 12 : 12
Końcowa - 107 - 12 : 10
Krzemieniecka - 107 - 12 : 9
Trawowa - 107 - 12 : 8
Stanisławowska (W.K. Formaty) - 107 - 12 : 7
Muchobór Wielki - 107 - 12 : 6
Zagony - 107 - 12 : 5
Tyrmanda - 107 - 12 : 4
You are starting from:  Tyrmanda at:  12 : 0

Time in minutes of shortest path from source to target stop:  37

Time of running algorithm:  0.03508329391479492
```

Rysunek 3: Najkrótsza trasa z Tyrmanda na most Grunwaldzki

```
1  RunDijkstraAlgorithm(wroclaw_graph, 'Tyrmanda', 'most Grunwaldzki', '12:00:00')
✓ 0.1s

Final destination:  most Grunwaldzki  reached at  12 : 37  by line  4

GO IN:  FAT - 4 - 12 : 15
GO OUT:  FAT - 107 - 12 : 13
GO IN:  Tyrmanda - 107 - 12 : 4

You are starting from:  Tyrmanda at:  12 : 0

Time in minutes of shortest path from source to target stop:  37

Time of running algorithm:  0.027889728546142578
```

Rysunek 4: Najkrótsza trasa z Tyrmanda na most Grunwaldzki - wyświetlanie skrócone

```
1  RunDijkstraAlgorithm(wroclaw_graph, 'Nowodworska', 'FAT', '12:00:00')
✓ 0.0s

Final destination:  FAT  reached by  12 : 11

ROD Oświata - 126 - 12 : 7
Wrocławski Park Technologiczny - 126 - 12 : 6
Szkocka - 126 - 12 : 5
Nowodworska - 126 - 12 : 3
You are starting from:  Nowodworska at:  12 : 0

Time in minutes of shortest path from source to target stop:  11

Time of running algorithm:  0.001981973648071289
```

Rysunek 5: Trasa Nowodworska - FAT

4 Algorytm A*

Przy szukaniu strategii na znajdowanie połączeń między przystankami, rodzi się pytanie czy nie dałoby się nakierowywać algorytmu, by faworyzował drogi zbliżające się do celu. Z pomocą przychodzi nam Algorytm A*, heurystyczna modyfikacja Algorytmu Dijkstry, rozbudowana o estymację kosztu ścieżki do celu. Tym razem porównywać będziemy wartość funkcji kosztu, zdefiniowaną jako:

$$f(curr, next) = g(curr, next) + h(next, end).$$

Funkcja g reprezentuje fukcję kosztu przejścia do danego wierzchołka (jak poprzednio), a h odległość euklidesową między przystankiem a naszym celem. Chcemy jednak trzymać się jednolitej metryki, więc podzielimy odległość przez średnią prędkość poruszania się samochodu w mieście, która według różnych źródeł wynosi około 23 km/h. Otrzymamy w ten sposób średni czas w minutach, jaki pojazd potrzebowałby by pokonać odległość między danymi przystankami w lini prostej.

Do obliczeń wykorzystamy współrzędne geograficzne i formułę wykorzystywaną w profesjonalnych kalkulatorach. Poniższy wzór zaczerpnięty został ze strony National Hurricane Center[1].

```
1  from math import cos, sin, asin, sqrt, radians
2
3  #function calculating distance between stops in kilometers
4  def euclideanDist(node1_lat_lon, node2_lat_lon):
5      x1,y1=node1_lat_lon
6      x2,y2=node2_lat_lon
7      EARTH_RAD = 6371
8
9      d=2*EARTH_RAD*asin(sqrt((sin((radians(x1-x2))/2))**2 + cos(radians(x1))*cos(radians(x2))*(sin((radians(y1-y2))/2))**2))
10
11     return d
```

```
1  euclideanDist(wroclaw_graph.m_lat_lon['Tyrmanda'],wroclaw_graph.m_lat_lon['most Grunwaldzki'])
```

```
7.714420821712388
```

Rysunek 6: Funkcja wyliczająca odległość euklidesową między przystankami

Odległość między powyższymi przystankami w kilometrach wynosi dokładnie tyle ile wskazuje wyliczona wartość.

4.1 Implementacja algorytmu

Będziemy operować na dwóch zbiorach: otwartym z węzłami odwiedzonymi, ale których nie wszyscy sąsiedzi zostali odwiedzeni oraz zamkniętym z węzłami, których wszyscy sąsiedzi zostali odwiedzeni. Sam algorytm podąża krok po kroku za pseudokodem z listy zadań, więc skupimy się głównie na sposobie w jaki pobierać i interpretować dane z grafu, by algorytm sobie z nimi poradził. Bardziej skomplikowanym problemem będzie przekształcenie algorytmu, by wyszukiwał połączenia z jak najmniejszą ilością przesiadek. Będzie to moment by wprowadzić modyfikację, która pozwoli na uzykanie satysfakcjonujących rezultatów tj. zmniejszenie wartości funkcji kosztu. Z tego względu podzielimy omawianie implementacji algorytmu na dwie podsekcje w zależności od wybranego kryterium.

4.1.1 Algorytm A* dla kryterium czasu

Implemenację rozpoczynamy od zainicjalizowania trzech słowników odpowiadających za wartości funkcji g, h, f dla każdego z przystanków. Przypomnijmy ich znaczenie:

- g - koszt czasowy między danym przystankiem, a przystankiem początkowym
- h - estymowany czas potrzebny na pokonanie dystansu między danym przystankiem a przystankiem końcowym
- f - suma dwóch powyższych wartości

Ponownie inicjalizujemy wszystkie wartości jako nieskończoność, poza przystankiem początkowym dla którego wprowadzamy wartość zero. Poza tym generujemy słownik poprzedników analogicznie jak w Algorytmie Dijskry oraz dwa puste zbiory: otwarty oraz zamknięty. Na wstępie dodajemy do zbioru otwartego przystanek początkowy.

Algorytm działa w pętli while póki w zbiorze otwartym znajdują się przystanki. Kolejno przechodzimy przez następujące kroki:

- Wybieramy wierzchołek-przystanek z najniższą wartością funkcji f znajdujący się obecnie z zbiorze otwartym.
- Jeśli wybraliśmy przystanek końcowy, kończymy działanie algorytmu i zwracamy słownik poprzedników oraz wartość funkcji g, czyli wyliczony czas przejazdu z przystanku początkowego. Nie trudno się domyślić, że wartość funkcji h jest tutaj równa 0.
- Jeśli wybraliśmy inny wierzchołek, to dodajemy go zamkniętego zbioru i usuwamy z otwartego. Pobieramy czas dodarcia do danego wierzchołka.
- Iterujemy po sąsiadach wierzchołka w celu potencjalnego zaktualizowania najkrótszej ścieżki oraz
  - jeśli sąsiad nie był nigdy odwiedzony przez algorytm, to wyliczamy dla niego wartości funkcji h, g, f z uwzględnieniem najszybszego połączenia z aktualnego wierzchołka do sąsiada,
  - jeśli sąsiad był odwiedzony przez algorytm i droga przez aktualny przystanek okaże się szybsza to aktualizujemy informacje dla tego sąsiada, dodatkowo jeśli znajdował się on w zamkniętym zbiorze, to go stamtąd zabieramy i dodajemy z powrotem do otwartego zbioru.

4.1.2 Algorytm A* dla kryterium przesiadek

Kroki algorytmu przy zmianie kryterium na liczbę przesiadek pozostają takie same, jednak musimy przedefiniować nasze funkcje kosztu. Funkcja h pozostanie ta sama tj. będzie oznaczała estymowany czas przejazdu z danego przystanku do przystanku końcowego. Nadamy jej jednak inną wagę. Funkcja g natomiast będzie łączyła w sobie zarówno koszt przesiadki, jak i czasu. W celu wyłapania potencjalnej przesiadki dodajemy do algorytmu krok sprawdzający, czy doszło do zmiany linii. Jeśli chodzi o koszt czasowy nie jesteśmy w stanie zupełnie się go pozbyć z algorytmu, ponieważ liczyłby on bardzo długo, czekając godzinami na bezpośrednie połączenie. Zależy nam na balansie pomiędzy czasem obliczeń, a satysfakcjonującymi wynikami. Rozwiązaniem jest manipulacja wagami poszczególnych kosztów (czasu, przesiadek i odległości).

Optymalne wyniki osiągnęłam dla kosztów:

- Estymowany czas do celu/wartość funkcji h - liczba minut * 10
- Koszt czasu przejazdu oraz oczekiwania na połączenie - liczba minut *1
- Koszt przesiadki - liczba przesiadek*120, można to interpretować jako cena przesiadki = koszt 2 godzin.

4.1.3 Modyfikacja Algorytmu A* dla kryterium przesiadek

Sama manipulacja funkcjami kosztów nie przyniosła jednak wystarczająco satysfakcjonujących rezultatów w przypadku Algorytmu A* dla kryterium przesiadek. To co przyniosło ogromny postęp, było przedefiniowaniem przeszukiwania wierzchołków w grafie. Wyzwaniem było znalezienie sposobu, by zmusić algorytm do poczekania na przystanku na kolejne połączenia (być może bezpośrednie!) lub co gorsza zdecydowanie się po drodze opuścić autobus/tramwaj i poczekać na inne połączenie, by finalnie wyznaczyć trasę z mniejszą ilością przesiadek. Udało mi się to osiągnąć poprzez traktowanie jako osobnych wierzchołków przystanków wraz z czasem pojawienia się na nich. Tworzymy w ten sposób wielką przestrzeń w postaci krotek *nazwa przystanku, czas pojawienia się na nim*. Krawędziami w takiej interpretacji grafu są znów możliwe połączenia jednak tylko takie, które przechowują godzinę późniejszą niż jest zapisana w naszym wierzchołku. Przykładowo z (Tyrmanda,'12:00:00') mamy krawędź do (Zagony,'14:00:00'), (Zagony,'21:30:00'), ale nie do (Zagony,'09:20:00'). W ten sposób tworzymy o wiele większą ale efektywniejszą przestrzeń poszukiwań.

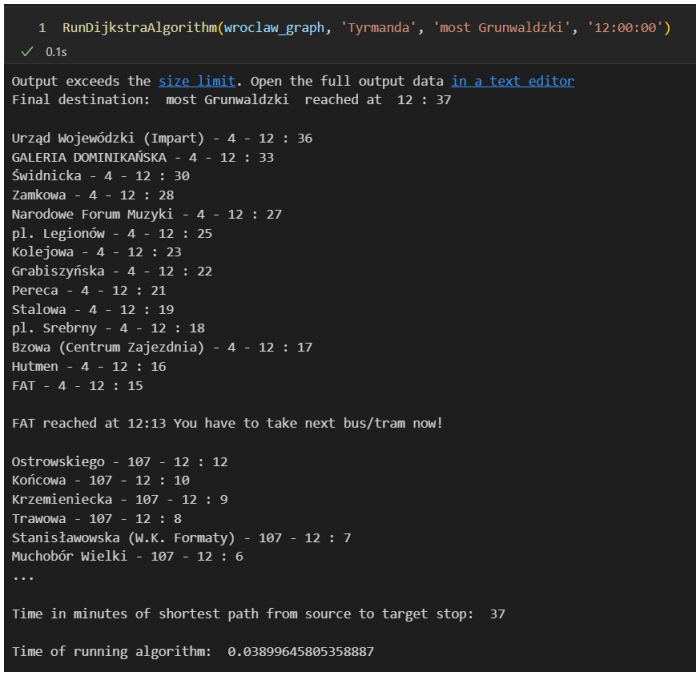
Ze względu na przedefiniowanie grafu (chodź sam obiekt wroclaw_graph pozostał taki sam), musimy zmodyfikować struktury danych w Algorytmie A*. Mianowice:

- Słowniki odpowiadające za przechowywanie wartości funkcji g, h, f są teraz słownikami zagnieżdzonymi, gdzie pierwszym kluczem jest nazwa przystanku, a drugim godzina pojawienia się na nim. Analogicznie słownik przechowujący informacje o poprzednikach.
- Węzeł sam w sobie reprezentowany jest krotką : *nazwa przystanku, czas pojawienia się na nim.*
- Węzeł początkowy redefiniujemy jako nazwę przystanku i czas pierwszego odjazdu.
- Iterujemy nie tylko po sąsiadujących przystankach, ale również po wszystkich połączeniach z jednego przystanku do drugiego (akceptując tylko późniejsze godziny!) dodając utworzone węzły do zbioru otwartego.

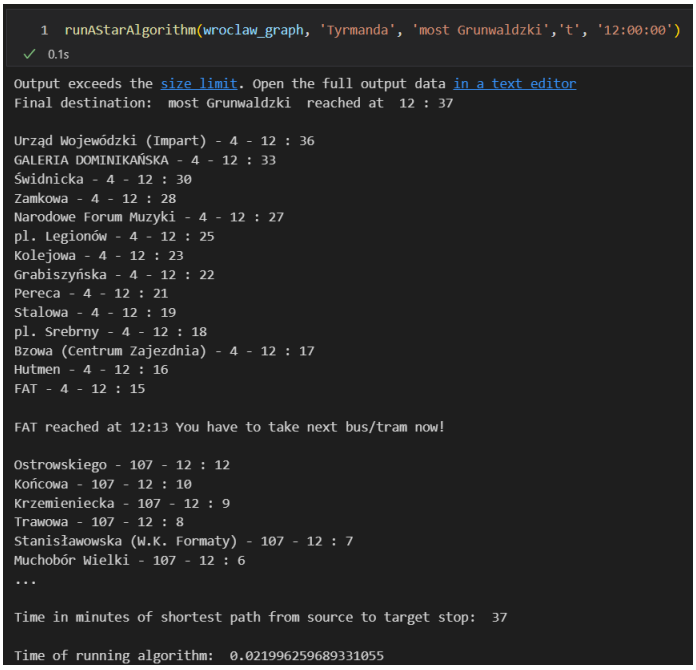
Idea, jak i kroki algorytmu pozostają te same.

4.2 Porównanie czasu działania Algorytmów Dijkstry i A* dla kryterium czasu

Testując oba algorytmy dla tych samych problemów, szczególnie poszukiwania połączeń dla przystanków bardziej oddalonych od siebie, łatwo zauważyć, że Algorytm A* jest znacznie szybszy. Poniżej przykład.



Rysunek 7: Porównanie czasu działania: Algorytm Dijkstry



Rysunek 8: Porównanie czasu działania: Algorytm A*

4.3 Porównanie działania Algorytmu A* dla kryterium czasu i przesiadek

```
1  runAStarAlgorithm(wroclaw_graph, 'Tyrmanda', 'PL. JANA PAWŁA II','t', '12:10:00')
2  #change cost = 120
✓ 0.0s
```

```
Final destination:  PL. JANA PAWŁA II  reached at  12 : 38

pl. Orląt Lwowskich - 74 - 12 : 36

pl. Orląt Lwowskich reached at 12:36 You have to take next bus/tram now!

Dworzec Świebodzki - 148 - 12 : 34
Smolecka - 148 - 12 : 32
Śrubowa - 148 - 12 : 31
Wrocławski Park Przemysłowy - 148 - 12 : 29
Dolnośląska Szkoła Wyższa - 148 - 12 : 28
Fabryczna - 148 - 12 : 27
Otyńska - 148 - 12 : 26
Strzegomska 148 - 148 - 12 : 24
Nowodworska - 148 - 12 : 22

Nowodworska reached at 12:18 You have to take next bus/tram now!

Nowy Dwór - 119 - 12 : 17
Rogowska - 119 - 12 : 15
MIŃSKA (Rondo Rotm. Pileckiego) - 119 - 12 : 13
Tyrmanda - 119 - 12 : 12
You are starting from:  Tyrmanda at:  12 : 10

Time in minutes of shortest path from source to target stop:  28

Time of running algorithm:  0.009976863861083984
```

Rysunek 9: Kryterium czasu na trasie Tyrmanda - PL. JANA PAWŁA

```
1  runAStarAlgorithm(wroclaw_graph, 'Tyrmanda', 'PL. JANA PAWŁA II','p', '12:10:00')
2  #change cost = 120
✓ 0.9s
```

```
Final destination :  PL. JANA PAWŁA II  reached at  12 : 57

Młodych Techników - 132 - 12 : 54
pl. Strzegomski (Muzeum Współczesne) - 132 - 12 : 53
Śrubowa - 132 - 12 : 50
Wrocławski Park Przemysłowy - 132 - 12 : 48
Dolnośląska Szkoła Wyższa - 132 - 12 : 47
Fabryczna - 132 - 12 : 46
Otyńska - 132 - 12 : 45
Strzegomska 148 - 132 - 12 : 43
Nowodworska - 132 - 12 : 41
Nowy Dwór - 132 - 12 : 40
Rogowska - 132 - 12 : 38
MIŃSKA (Rondo Rotm. Pileckiego) - 132 - 12 : 36
Tyrmanda - 132 - 12 : 35
You are starting from:  Tyrmanda at:  12 : 10

Number of changes needed on path from source to target stop:  0

Time of running algorithm:  0.8814737796783447
```

Rysunek 10: Kryterium przesiadek na trasie Tyrmanda - PL. JANA PAWŁA

4.4 Porównanie działania Algorytmu A* dla różnej wagi kosztu przesiadki

Źle dobrane wagi w wyliczaniu funkcji kosztu mogą przynieść wyniki dalekie od optymalnych. Spójrzmy na przykład poniżej.

```
1  runAStarAlgorithm(wroclaw_graph, 'Tyrmanda', 'PL. JANA PAWŁA II','p', '12:10:00')
2  #change cost = 120
✓ 0.9s
Final destination :  PL. JANA PAWŁA II  reached at  12 : 57

Młodych Techników - 132 - 12 : 54
pl. Strzegomski (Muzeum Współczesne) - 132 - 12 : 53
Śrubowa - 132 - 12 : 50
Wrocławski Park Przemysłowy - 132 - 12 : 48
Dolnośląska Szkoła Wyższa - 132 - 12 : 47
Fabryczna - 132 - 12 : 46
Otyńska - 132 - 12 : 45
Strzegomska 148 - 132 - 12 : 43
Nowodworska - 132 - 12 : 41
Nowy Dwór - 132 - 12 : 40
Rogowska - 132 - 12 : 38
MIŃSKA (Rondo Rotm. Pileckiego) - 132 - 12 : 36
Tyrmanda - 132 - 12 : 35
You are starting from:  Tyrmanda at:  12 : 10

Number of changes needed on path from source to target stop:  0

Time of running algorithm:  0.8814737796783447
```

Rysunek 11: Koszt przesiadki = 120

```
1  runAStarAlgorithm(wroclaw_graph, 'Tyrmanda', 'PL. JANA PAWŁA II','p', '12:10:00')
2  #change cost = 80
✓ 0.3s
Final destination :  PL. JANA PAWŁA II  reached at  12 : 51

PL. SOLIDARNOŚCI - 127 - 12 : 49

PL. SOLIDARNOŚCI reached at 12:49 You have to take next bus/tram now!

Inowrocławska - 107 - 12 : 47
Szczepin - 107 - 12 : 46
pl. Strzegomski (Muzeum Współczesne) - 107 - 12 : 45
Śrubowa - 107 - 12 : 42
Wrocławski Park Przemysłowy - 107 - 12 : 40
Dolnośląska Szkoła Wyższa - 107 - 12 : 39
Fabryczna - 107 - 12 : 38
Otyńska - 107 - 12 : 37
Strzegomska 148 - 107 - 12 : 35
Nowodworska - 107 - 12 : 33
Nowy Dwór - 107 - 12 : 32
Rogowska - 107 - 12 : 30
MIŃSKA (Rondo Rotm. Pileckiego) - 107 - 12 : 28
Tyrmanda - 107 - 12 : 27
You are starting from:  Tyrmanda at:  12 : 10

Number of changes needed on path from source to target stop:  1

Time of running algorithm:  0.28503847122192383
```

Rysunek 12: Koszt przesiadki = 80

4.5 Przykład uzyskania takiego samego czasu przejazdu przy niższej liczbie przesiadek przez Algorytm A* z kryterium przesiadek

```
1  runAStarAlgorithm(wroclaw_graph, 'Tyrmanda', 'Klimasa','p', '15:00:00')
✓ 7.1s

Output exceeds the size limit. Open the full output data in a text editor
Final destination :  Klimasa  reached at  15 : 49

Transbud - 125 - 15 : 47
Nyska - 125 - 15 : 46
Bardzka - 125 - 15 : 45
Kamienna - 125 - 15 : 42
Prudnicka - 125 - 15 : 40
Hubska (Dawida) - 125 - 15 : 38
DWORZEC AUTOBUSOWY - 125 - 15 : 35
EPI - 125 - 15 : 32
Arena - 125 - 15 : 31
Komandorska - 125 - 15 : 29
Pl. Hirszfelda - 125 - 15 : 26
Krucza - 125 - 15 : 23
Krucza (Mielecka) - 125 - 15 : 21
Kolbuszowska (Stadion) - 125 - 15 : 20
Inżynierska - 125 - 15 : 19
Aleja Pracy - 125 - 15 : 17
FAT - 125 - 15 : 15

FAT reached at 15:14 You have to take next bus/tram now!

Ostrowskiego - 319 - 15 : 12
Końcowa - 319 - 15 : 10
Krzemieniecka - 319 - 15 : 9
...

Number of changes needed on path from source to target stop:  1

Time of running algorithm:  7.058332443237305
```

Rysunek 13: Kryterium przesiadek na trasie Tyrmanda - Klimasa

```
1  runAStarAlgorithm(wroclaw_graph, 'Tyrmanda', 'Klimasa','t', '15:00:00')
✓ 0.1s

Output exceeds the size limit. Open the full output data in a text editor
Final destination:  Klimasa  reached at  15 : 49

Transbud - 125 - 15 : 47
Nyska - 125 - 15 : 46
Bardzka - 125 - 15 : 45

Bardzka reached at 15:42 You have to take next bus/tram now!

Kamienna - 31 - 15 : 41
Prudnicka - 31 - 15 : 39
Gajowa - 31 - 15 : 37
Joannitów - 31 - 15 : 36
DWORZEC AUTOBUSOWY - 31 - 15 : 34

DWORZEC AUTOBUSOWY reached at 15:31 You have to take next bus/tram now!

EPI - 122 - 15 : 30
Piłsudskiego - 122 - 15 : 26
pl. Legionów - 122 - 15 : 24

pl. Legionów reached at 15:24 You have to take next bus/tram now!

Kolejowa - 5 - 15 : 22
Grabiszyńska - 5 - 15 : 21
Pereca - 5 - 15 : 20
...

Time in minutes of shortest path from source to target stop:  49

Time of running algorithm:  0.05897235870361328
```

Rysunek 14: Kryterium czasu na trasie Tyrmanda - Klimasa

5 Tabu search

Omówione algorytmy wykorzystamy, by pójść krok dalej i spróbować zmierzyć się z problemem komiwojażera dla listy przystanków. Pozwoli nam na to przeszukiwanie Tabu.

5.1 Implementacja algorytmu

Działanie algorytmu rozpoczynamy od losowego wygenerowania początkowego rozwiązania. Jest to losowa permutacja zadanych przystanków. Pozwala nam na to funkcja *randomPermutation(stops_list)*. Inicializujemy również pusty zbiór Tabu. Ustalamy w tym miejscu liczbę kroków iteracji.

Ustawiamy rozwiązanie początkowe jako najlepsze znane rozwiązanie oraz aktualne rozwiązanie. Wyliczamy jego koszt za pomocą funkcji *calcCostforSolution*. Wyznaczanie kosztu to nic innego jak zsumowanie kosztów połączeń pomiędzy poszczególnymi przystankami w kolejności jakie wyznacza dana permutacja. Pamiętamy przy tym, że zaczynamy oraz kończymy w przystanku początkowym. Funkcja monitoruje czas jaki mija przy zdobywaniu kolejnych przystanków i przekazuje go w kolejnym kroku jako czas startowy dla kolejnego połączenia. Na koniec odejmuje czas kiedy wyruszyliśmy z przystanku startowego od czasu kiedy znów po całej podróży do niego dotarliśmy. Dla kryterium przesiadek będzie to liczba przesiadek.

Następnie w pętli będziemy szukać najlepszego sąsiada aktualnego rozwiązania. Na początku aktualnym rozwiązaniem będzie rozwiązanie początkowe. Następnie, będzie to najlepszy sąsiad poprzedniego aktualnego rozwiązania. Sąsiadami są rozwiązania, które różnią się od aktualnego podmienieniem dwóch dowolnych przystanków. Sąsiadów jest zawsze $n * (n-1)/2$. Generujemy więc w pętli każdego z sąsiadów po kolei, następnie sprawdzamy czy sąsiad znajduje się już w zbiorze Tabu. Jeśli nie to dodajemy go tam plus sprawdzamy, czy wygenerował on lepsze rozwiązanie niż poprzedni sprawdzeni sąsiedzi. Jeśli tak to zapisujemy go jako najlepsze aktualne rozwiązanie. Po całej takiej pętli sprawdzania wszystkich sąsiadów po kolei, sprawdzamy jeszcze, czy lokalne najlepsze rozwiązanie nie dało nam lepszego wyniku niż najlepsze znane dotychczas rozwiązanie. W takim wypadku aktualizujemy je. Natępnie najlepsze lokalne rozwiązanie staje się aktualnym rozwiązaniem i powtarzamy wymienione kroki.

5.2 Modyfikacja algorytmu o losowanie sąsiadów i ograniczenie na rozmiar tablicy Tabu

W celu przyspieszenia działania algorytmu wprowadziłam do implementacji dwie modyfikacje. Pierwszą jest ograniczenie na rozmiar tablicy T. Jako maksymalny rozmiar tablicy ustaliłam $5 * len(stops)^2$. Kontrolę rozmiaru tablicy uzyskuję przez sprawdzanie przed innymi krokami w pętli czy rozmiar nie został przekroczony. Jeśli tak, to ucinam początkowe nadmiarowe rozwiązania imitując tym samym kolejkę FIFO. Nie sprawdzam rozmiaru tablicy przy każdej próbie dodania sąsiada do niej, by zaoszczędzić czas. Tworzę natomiast pewien bufor, który pozwala każdemu z sąsiadów w danym rzucie dostać się do tablicy, nawet jeśli rozmiar został w międzyczasie przekroczony. Maksymalne możliwe przekroczenie rozmiaru tablicy przed kolejnym przejściem pętli jest równe liczbie sąsiadów czyli $n * (n-1)/2$.

Wprowadziłam również próbkowanie sąsiadów, polegające na losowym wygenerowaniu n par przystanków do podmiany bez powtarzania. Limit sąsiadów sprawdzanych w każdej pętli ustawiłam jako połowę wszystkich sąsiadów.

5.3 Porównanie czasu oraz efektywności działania podstawowej i zmodyfikowanej wersji algorytmu.

Liczby na górze oraz na dole oznaczają łączny czas całej podróży. Pierwsza z liczb oznacza czas losowo wygenerowanego rozwiązania, a druga wyznaczonegoprzez algorytm. Widzimy, że zmodyfikowany algorytm uzyskał ten sam wynik w znacznie krótszym czasie.

```
1  travellingSalesmanUsingTabuSearch(wroclaw_graph,"Tyrmanda",["most Grunwaldzki","FAT","Morelowskiego","Pereca","Kwiska"],740, criterion="t")
✓ 7.1s

Output exceeds the size limit. Open the full output data in a text editor
153
------------------------------------------------------------
Final destination:  Kwiska  reached at  12 : 43  by line  122

GO IN:  Na Ostatnim Groszu - 122 - 12 : 40
GO OUT:  Na Ostatnim Groszu - 136 - 12 : 39
GO IN:  Szkocka - 136 - 12 : 35
GO OUT:  Szkocka - 126 - 12 : 35
GO IN:  Nowodworska - 126 - 12 : 33
GO OUT:  Nowodworska - 107 - 12 : 33
GO IN:  Tyrmanda - 107 - 12 : 27

You are starting from:  Tyrmanda at:  12 : 20
------------------------------------------------------------
Final destination:  most Grunwaldzki  reached at  13 : 10  by line  16

GO IN:  pl. Wróblewskiego - 16 - 13 : 7
GO OUT:  pl. Wróblewskiego - 3 - 13 : 7
GO IN:  Kwiska - 3 - 12 : 45

You are starting from:  Kwiska at:  12 : 43
------------------------------------------------------------
Final destination:  Pereca  reached at  13 : 35  by line  5

GO IN:  DWORZEC GŁÓWNY - 5 - 13 : 26
...

You are starting from:  Morelowskiego at:  13 : 52
------------------------------------------------------------
111
```

Rysunek 15: Wynik i czas działania klasycznego algorytmu Tabu

```
1  travellingSalesmanUsingTabuSearchWithTConstraint(wroclaw_graph,"Tyrmanda",["most Grunwaldzki","FAT","Morelowskiego","Pereca","Kwiska"],740)
✓ 4.8s

Output exceeds the size limit. Open the full output data in a text editor
153
------------------------------------------------------------
Final destination:  Kwiska  reached at  12 : 43  by line  122

GO IN:  Na Ostatnim Groszu - 122 - 12 : 40
GO OUT:  Na Ostatnim Groszu - 136 - 12 : 39
GO IN:  Szkocka - 136 - 12 : 35
GO OUT:  Szkocka - 126 - 12 : 35
GO IN:  Nowodworska - 126 - 12 : 33
GO OUT:  Nowodworska - 107 - 12 : 33
GO IN:  Tyrmanda - 107 - 12 : 27

You are starting from:  Tyrmanda at:  12 : 20
------------------------------------------------------------
Final destination:  most Grunwaldzki  reached at  13 : 10  by line  16

GO IN:  pl. Wróblewskiego - 16 - 13 : 7
GO OUT:  pl. Wróblewskiego - 3 - 13 : 7
GO IN:  Kwiska - 3 - 12 : 45

You are starting from:  Kwiska at:  12 : 43
------------------------------------------------------------
Final destination:  Pereca  reached at  13 : 35  by line  5

GO IN:  DWORZEC GŁÓWNY - 5 - 13 : 26
...

You are starting from:  Morelowskiego at:  13 : 52
------------------------------------------------------------
111
```

Rysunek 16: Wynik i czas działania ulepszonego algorytmu Tabu

6 Biblioteki

Wykorzystane biblioteki:

- Pandas - zaimportowanie danych z pliku csv, zapisanie ich w ramce danych, skorzystanie z podstawowych funkcji pomocnych do przeanalizowania danych i dalszych operacji na kolumnach i wierszach.
- Datetime - przekonwertowanie godzin z formatu string na obiekt czas i umożliwienie dalszego przekształcenia czasu na liczbę minut.
- Time - mierzenie i porównywanie czasu działania algorytmów.
- Math - skorzystanie z funkcji cos, sin, asin, sqrt, radians przy obliczaniu odległości euklidesowej
- Random - wygenerownie losowego rozwiązania problemu komiwojażera.
- Copy - głębokie kopiowanie listy rozwiązań.

7 Problemy implementacyjne

Implementacja powyższych algorytmów okazała się dość dużym wyzwaniem ze względu na specyfikę problemu oraz rodzaj danych. Niemniej po wielu próbach i poświęconym czasie, uważam, że udało mi się osiągnąć satysfakcjonujące rezultaty. Głównymi problemami jakie napotkałam były:

1. Interpretacja danych, zdefiniowanie krawędzi i wierzchołków w grafie, w szczególności przy algorytmie A* dla najmniejszej ilości przesiadek.
2. Problemem okazały się sąsiadujące wierzchołki, które po danej godzinie nie mają żadnych połączeń - trzeba było tą sytuację obsłużyć.
3. Znalezienie złotego środka przy określeniu kosztu przesiadki, czasu i odległości w algorytmie A*

Biorąc pod uwagę wszystkie trudności w implementacji, największym wyzwaniem zdecydowanie był problem znalezienia połączenia z najmniejszą ilością przesiadek.

Literatura

[1] Ed Williams, *Great Circle Calculator*, http://edwilliams.org/avform147.htm#Dist.