Algorytmy wyznaczania najkrótszej ścieżki

Na podstawie rozkładu jazdy transportu publicznego miasta Wrocław

Wiktor Kubera

Spis treści

1. Przygotowanie danych

1.1. Eksploracja danych

Po wstępnym zapoznaniu się z danymi zawartymi w pliku connection_graph.csv, widzę, że dotyczą one **połączeń między przystankami we Wrocławiu**.
Po obserwacji liczby danych, przyszła mi pierwsza myśl do głowy, że prawie **milion rekordów dotyczących połączeń między przystankami, to trochę za dużo jak na Wrocław**, prawdopodobnie dotyczą one **różnych dni (w tygodniu?, w roku?)** niemniej jednak te dane nie są zawarte w pliku.
Tak więc stwierdziłem, że sprawdzę programowo czy (i jeśli tak, to ile) wiersze mają swoje duplikaty.
Dane pobieram do programu używając do tego **biblioteki Pandas**.
Biblioteka pandas służy do ogólnie rozumianej analizy danych, między innymi możemy ładować różne formaty plików, które udostępniane są nam w postaci tablic, na których możemy wykonać sporo użytecznych operacji. Do badania duplikatów używam skryptu, który sam napisałem:

```
def find_duplicates_in_data(df):
    uniques = set()
    uniques_index_dict = {}
    duplicates_count = {}

    for i, row in df.iterrows():
        raw = get_raw_row(row)
        if raw not in uniques:
            uniques.add(raw)
            uniques_index_dict[raw] = i
        else:
            index = uniques_index_dict[raw]
            if index not in duplicates_count:
                duplicates_count[index] = 1
            else:
                duplicates_count[index] += 1
    return duplicates_count
```

Po zakończeniu działania skryptu wyliczyłem interesujące mnie wartości i sporządziłem podsumowanie:

```
Liczba wszystkich wierszy: 996520
Liczba duplikatów: 541289
Liczba unikalnych wierszy: 455231
Maksymalna liczba powtórzeń: 7
Minimalna liczba powtórzeń: 1
```

Tak więc możemy pozbyć się więcej niż połowy pliku w kontekście problemów rozpatrywanych na tej liście.
Tak więc korzystając ze skryptu, który napisałem sam, tworzę nowy plik, który nie zawiera żadnych duplikatów:

```
Wiktor *
def create_unique_connections(file_name):
    unique_raw_csv_rows = set()
    df_main = retrieve_data("connection_graph.csv")
    for i, row in df_main.iterrows():
        csv_row = get_raw_row(row)
        unique_raw_csv_rows.add(csv_row)

    with codecs.open(get_data_source_path(file_name), "w", "utf-8") as f:
        csv_header = "nr,company,line,departure_time,arrival_time,start_st
        f.write(csv_header)
        i = 0
        for row in unique_raw_csv_rows:
            f.write(f"{i},"+ row + "\n")
            i += 1
```

1.2. Modelowanie danych

1.2.1. Wierzchołek - przystanek

Wierzchołek w grafie, czyli przystanek, modeluję jako trójkę $(nazwa, latitude, longitude)$. Na początku chciałem, aby wierzchołek był unikalny ze względu na te trzy wartości, niemniej jednak po namyśle i testowaniu, stwierdziłem, że takie podejście by bardzo utrudniało testowanie, gdyż zamiast samej nazwy, trzeba by było podać dokładne współrzędne geograficzne, aby odnieść się do konkretnego przystanku, ponadto algorytmy, które zaimplementowałem, rozważają tylko możliwość poruszania się za pomocą linii, czytaj, nie uwzględnia się możliwość przejścia pieszo.
Tak więc stwierdziłem, że uogólnie przystanki, tak więc:

- Zebrałem wszystkie przystanki ze względu na nazwę i je pogrupowałem
- Każdemu przystankowi wyliczyłem średnią arytmetyczną współrzędnych geograficznych
- Każdemu przystankowi przypisałem wszystkie połączenia ze względu na nazwę

Z 2284 unikalnych przystanków, zrobiło się 939 po generalizacji.
Ze względu na potrzeby istnienia informacji o sąsiedztwie wierzchołków, koniec końców pojedynczy wierzchołek w grafie, to mapa wierzchołka do listy innych wierzchołków.

1.2.2. Krawędź - połączenie do listy realizacji

Pojedyncza krawędź grafu łączy dwa przystanki ze sobą (w sposób skierowany).
Tutaj przychodzi pewien problem, przecież np z Placu Grunwaldzkiego na Galerię Dominikańską możemy się przedstać o różnych godzinach!
Tak więc mój model krawędzi wygląda tak, że jest to mapa danego połączenia do listy możliwych realizacji.
Realizacje modeluje za pomocą czwórki $(firma, linia, czas\_odjazdu, czas\_dojazdu)$.
Natomiast połączenie modeluję za pomocą dwójki $(start, stop)$.

2. Wyszukiwanie najkrótszych połączeń

2.1. Algorytm Dijkstry

Najważniejsze rzeczy, które wykorzystuję do implementacji algorytmu, to:

- Kolejka priorytetowa (minheap)
- Mapa kosztu
- Mapa poprzedników
- Algorytm wybierania najlepszej realizacji połączenia
- Algorytm obliczania kosztu
- Algorytm rekonstrukcji ścieżki

W następnej części omówię każdy element.

2.1.1. Kolejka priorytetowa

Jest to kluczowy element, dzięki któremy algorytm w łatwy sposób wybiera węzeł, który aktualnie ma odwiedzić, ta struktura danych jest kluczowa, aby odnaleźć najkrótszą ścieżkę (o najmniejszym koszcie).
Wykorzystuję gotową kolejkę priorytetową typu minheap, którą **udostępnia biblioteka heapq**
Minheap to taki kopiec, który zawiera porównywalne elementy, oraz zawsze zawiera na wierzchu element, który jest najmniejszy.
Ekstrakcja najmniejszego elementu jest mało kosztowna, niemniej jednak po ekstrakcji następuje rearanżacja elementów kopcu.
Tak więc w przypadku tego algorytmu, kopiec jako najmniejszy element, trzyma przystanek, do którego koszt jest najmniejszy.

2.1.2. Mapa kosztu

Wykorzystuję słownik aby przechowywać koszt do poszczególnych wierzchołków, aby móc w łatwy sposób śledzić i aktualizować aktualny koszt ścieżki, czytaj aby dla każdego wierzchołka mieć koszt ścieżki aż do poprzednika.

2.1.3. Mapa poprzedników

Dla każdego odwiedzonego wierzchołka (oprócz pierwszego) przechowuję poprzednika oraz realizację, która sprawiła że do niego dotarłem.
Jest to konieczne aby dokonać rekonstrukcji ścieżki.

2.1.4. Algorytm wybierania najlepszej realizacji połączenia

W przypadku algorytmu dijkstry, między dwoma przystankami wybieram realizację połączenia w taki sposób, że jest to

- **Realizacja która odbędzie się w przyszłości**
- **Realizacja, której czas przejazdu plus czas oczekiwania na nią jest najmniejsza**

```
def find_best_future_execution_time(current_time, executions):
    lowest_cost = infinity
    best = None
    for execution in executions:
        if execution.departure_time >= current_time:
            cost = execution.departure_time - current_time +
            (execution.arrival_time - execution.departure_time)
            cost_int = int(cost.seconds)
            if cost_int < lowest_cost:
                lowest_cost = cost_int
                best = execution
    return best
```

2.1.5. Algorytm obliczania kosztu

To w zasadzie koszt realizacji plus czas czekania na realizację, taki koszt jest dodawany wraz z wierzchołkiem do kolejki priorytetowej.

2.1.6. Algorytm rekonstrukcji ścieżki

To przejścia po słowniku poprzedników aż do węzła początkowego:

```
current = destination
path = [previous[current]]

while True:
    current = previous[current][0]
    if current == source:
        break
    path.append(previous[current])

path.reverse()
return path
```

Na końcu ścieżkę oczywiście odwracam, gdyż chcemy ścieżkę od początku do końca a nie na odwrót.

2.1.7. Problemy przy implementacji

Na samym początku sąsiedztwo wierzchołka realizowałem za pomocą zbioru.
Wówczas dla tego samego początku A i celu B i czasu t dostawałem czasem różne ścieżki, jest to związane z tym, że w grafie może być więcej niż jedna najkrótsza ścieżka, a algorytm kończy swoje działanie, kiedy dotrze do celu, tak więc tutaj ważna była kolejność wierzchołków w sąsiedztwie, **zbiór nie gwarantuje żadnej kolejności**, tak więc aby algorytm był deterministyczny, to stwiedziłem, że sąsiedztwem będzie lista.

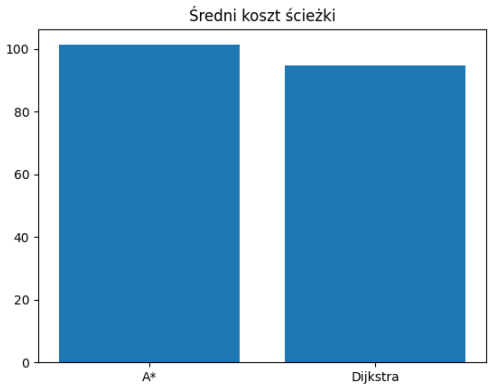
2.2. Algorytm A*

Algorytm A* to w zasadzie algorytm Dijkstry z dodatkowym elementem, **heurystyką**. Jest to pewna funkcja, która estymuje koszt ścieżki od następnego wierzchołka aż do celu.
Wówczas wypadkowa kosztu, to lokalne kryterium plus wartość heurystyki.
W moim przypadku używam **heurystyki odległości euklidesowej**, gdyż w naszej przestrzeni możemy wyróżnić nieskończenie wiele kierunków (wsp. geograficzne to liczby rzeczywiste)

2.2.1. A* z kryterium czasu

Ten algorytm działa prawie identycznie, co Dijkstra (z kryterium czasu), z tą różnicą, że do kosztu dodawana jest heurystyka (w tym przypadku odległość euklidesowa pomiędzy węzłem następnym, a docelowym).
Wybór realizacja połączenia w przypadku A* z kryterium czasu jest identyczny co w Dijkstrze.
Można sobie pomyśleć, że skoro tak mało się różni od algorytmu Dijkstry, to powinien mieć podobne wyniki, przekonajmy się.
Tak jak wcześniej, **uruchomiłem dijkstrę oraz A* z kryterium czasu dla 750 testowych przypadków i zapisałem takie dane, jak długość ścieżki, koszt ścieżki i czas obliczeń**. Zobaczmy na wykresach jakie wyniki osiągają te algorytmy. Powyższe wy-

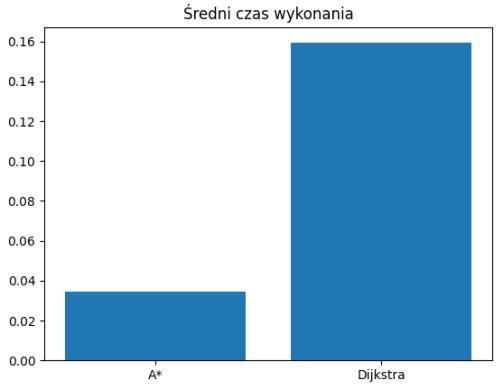
Rysunek 1: Średnia długość ścieżki



Rysunek 2: Średni koszt ścieżki

kresy prezentują wartości średnie dla 750 uruchomień.
Jak widać, **A* ma nieco gorsze metryki jakościowe odnośnie samej ścieżki**, czego byśmy się spodziewali.

Niemniej jednak zostało jeszcze porównanie czasu obliczeń. Powyższy wykres zestawia



Rysunek 3: Średni czas obliczeń

porównanie średniego czasu obliczeń ścieżki, jak widzimy, **Algorytm A* jest znacznie szybszy niż Dijkstra, przy małych odstępach od optymalnych metryk jakościowych**!

2.2.2. A* z kryterium przesiadek

Ta wersja algorytmu ma minimalizować liczbę przesiadek podczas trasy.
W porównaniu z algorytmem A* z kryterium czasu, należy wprowadzić następujące modyfikacje:

- Zmodyfikować algorytm wybierania najlepszej realizacji połączenia
- Zmodyfikować funkcję liczenia kosztu połączenia jako funkcję przesiadki

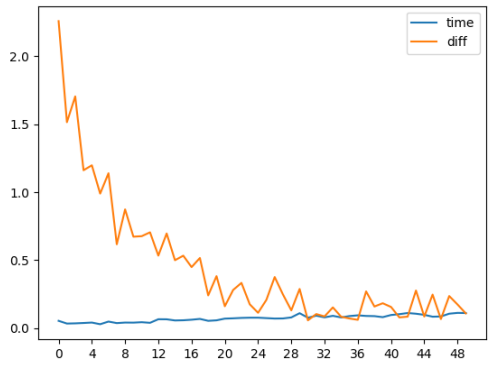
Tak więc w porównaniu do poprzedniej wersji algorytmu, algorytm wyboru najlepszej realizacji połączenia najpierw wyszukuje najlepsze czasowo realizace, które nie zmieniają linii, niemniej jednak, jeśli taka nie istnieje, to deleguje wybór do poprzedniej wersji funkcji, czytaj, jeśli nie istnieje taka realizacja, w której mogę wybrać tę samą linię, to po prostu wybierz taką, która jest najlepsza pod względem czasu.
Jeśli chodzi o modyfikację funkcji liczenia kosztu, to działa ona w taki sposób, że tak jak poprzednia, liczy koszt ze względu na czas i **jeżeli jest to połączenie z przesiadką, to dodaje pewien dodatkowy koszt**.
Wartość tego kosztu w pewien sposób przesłania heurystykę. Gdy koszt jest mały, to istnieje większe prawdopodobieństwo wybrania przesiadki, jeśli jest duży, to prawdopodobieństwo jest znikome, niemniej jednak wraz ze wzrastającym kosztem rośnie liczba odwiedzonych wierzchołków, gdyż heurystyka ma mniejszą moc.

2.2.3. Modyfikacja algorytmu A* z punktu (c), który pozwoli na zmniejszenie wartości funkcji kosztu uzyskanego rozwiązania lub czasu obliczeń

Tak więc najprostszą modyfikacją tutaj jest pokazanie, że jeśli zmniejszymy koszt przesiadki, to maleje czas obliczeń.
Niemniej jednak taka odpowiedź mnie nie satysfakcjonuje i chciałbym przeprowadzić badania statystyczne, które pokażą zależność wartości kosztu przesiadki i metryk jakościowych.
Tak więc w ramach badań, przeprowadzę dwa testy:

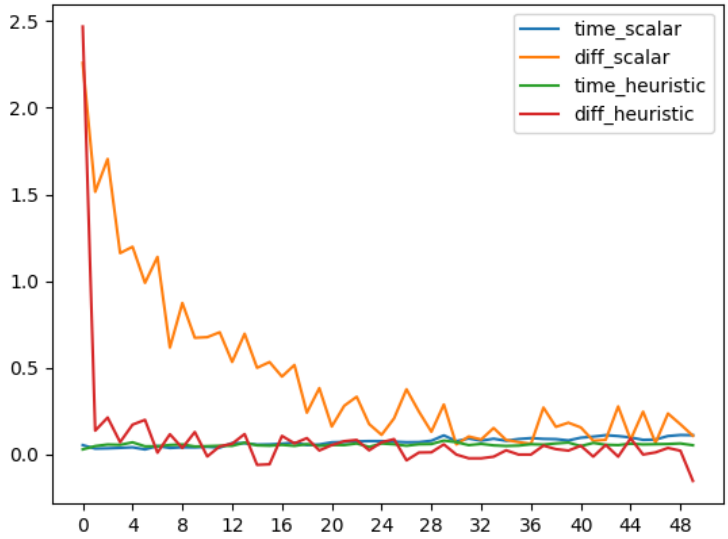
- Przegląd jakości dla kosztu w przedziale $< 0; 49 >$
- Przegląd jakości dla kosztu jako funkcji heurystyki - wartość kosztu będzie mnożona przez aktualną wartość heurystyki $c = n * h$

Dla każdego przypadku będę wyliczał liczbę przesiadek, dla kosztu $c = 500$, gdyż dla tak dużej wartości, nie przesiadanie się będzie miało bardzo duży priorytet i będę porównywał tę liczbę z aktualnym wynikiem algorytmu, im różnica mniejsza, tym lepiej, kolejną wartością, którą będę wyliczał, to czas działania algorytmu, dla każdej wartości kosztu testowego, rozpatruję ponad 90 losowych przypadków, tak, aby móc wyliczyć średnią.



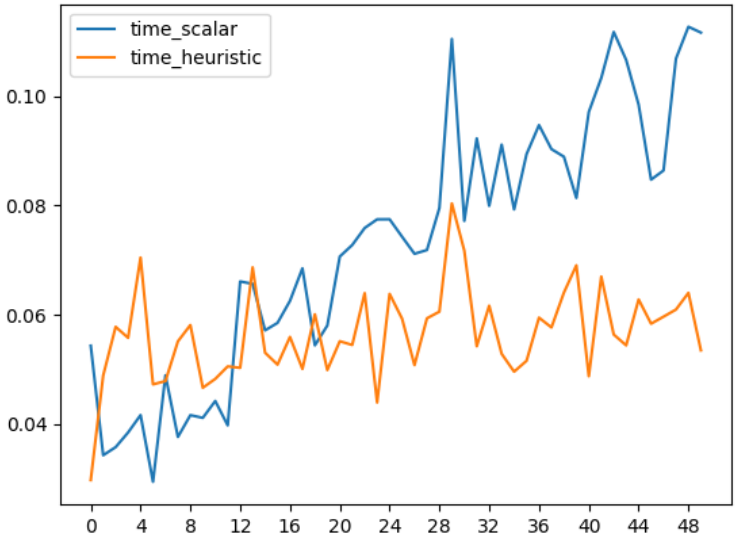
Rysunek 4: Średnie odchylenie od 'dobrej' liczby przesiadek i czas w zależności od wartości stałej kosztu

Jak widzimy z poprzedniego wykresu, jeśli ta stała rośnie, to rośnie czas i maleje średnie odchylenie.
Miejscem w którym odchylenie przestaje aż tak maleć, to około $c = 20$.
Nie ma sensu wybierać większej wartości, gdyż małą poprawą jakości odchylenia będzie rosnął czas.
Porównajmy teraz jak to się ma jeśli koszt jest funkcją heurystyki w postaci $c = n * h$



Rysunek 5: Porównanie wykresów

Jak widzimy, jeśli koszt jest funkcją heurystyki, to osiąga o wiele lepsze wyniki, **nawet osiąga wartości ujemne !** - to oznacza, że ma lepsze osiągi, niż dla stałej $c = 500$! (dla której swoją drogą czas obliczeń jest podobny do dijkstry!)
Tak więc na pewno lepiej wyliczać koszt, jako funkcję heurystyki, tylko jaką wartość c dobrać tak, aby ulepszyć wystarczająco jakość wyników przy wystarczająco małym czasie?
Z wykresu potencjalnymi punktami krytycznymi są $c = 15$, $c = 26$ i $c = 49$.
Warto spojrzeć na wykres zeskalowany pod czas i wtedy zdecydować.



Rysunek 6: Porównanie wykresów czasu

Z wymienionych wcześniej punktów charakterystycznych, najlepszy pod względem czasu jest $c = 26$, skoro A* powinien być szybki, to wybieram właśnie ten punkt, warto zauważyć, że odchylenie dla tego punktu jest ujemne!
Tak więc aby zoptymalizować wyniki algorytmu A* z kryterium przesiadek, **należy liczyć koszt przesiadki, jako funkcję heurystyki**.
Co prawda czas pozwolił mi zbadać tylko wyniki funkcji kosztu jako liniowej funkcji heurystyki, można by było próbować innych klas funkcji, np logarytmicznej oraz zwiększyć dokładność przeglądu zupełnego, niemniej jednak jest to temat na badania naukowe, na których można by było spędzić bardzo dużo czasu!

3. Podsumowanie

Zadanie wymagało dobrego zamodelowania grafu, aby wykorzystać abstrakcję, jaką jest ogólny graf, do tego konkretnego zadania.
Według mnie ważnym aspektem tego zadania również była eksploracja danych, mi udało odchudzić się plik bez utraty danych o ponad 50%!
To spowodowało znaczne przyspieszenie ładowania danych do grafu.
Jeśli chodzi o implementacje algorytmów, to najcięższe dla mnie było debugowanie, w tak wielkim grafie ciężko było się połapać co gdzie następuje.

3.1. Komentarz odnośnie algorytmów

Oba algorytmy, czytaj, Dijkstra i A* mają swoje wady i zalety i wybór algorytmu bardzo zależy od okoliczności.
Jeśli mamy mało rozbudowany graf lub potrzebujemy optymalnych (w kontekście kosztu ścieżki) wyników, to powinniśmy użyć Algorytmu Dijkstry.
Natomiast jeżeli mamy duży graf i możemy sobie pozwolić na pewne dopuszczalne odchylenia jakościowe, to powinniśmy użyc Algorytmu A*.
Warto zaznaczyć, że A* bardzo zależy od heurystyki, tutaj też, w zależności od konkretnego problemu heurystyki mogą być różne, nie ma raczej uniwersalnej.
Należy wobec takiej heurystyki przeprowadzić różne badania, gdyż źle dobrana heurystyka może spowodować, że A* będzie działał gorzej czasowo i jakościowo niż dijkstra!
W przypadku A*, na podstawie sytuacji, którą napotkałem, warto rozważyć podejście hybrydowe, czytaj rozważyć, czy nie można w jakiś sposób powiązać lokalnego kryterium wraz z heurystyką w celu optymalizacji kryteriów jakości.
Ważne jest umiejętne korzystanie z danych, zestawianie znaczących informacji oraz dokonywanie eksperymentów w odpowiednich przestrzeniach i ich interpretacja.