Sprawozdanie - SI

Lab 1

Wiktor Sadowy

Cel zadania

W ramach pierwszego zadania mieliśmy zaimplementować:

a) algorytm Dijkstry w oparciu o kryterium czasu,
b) algorytm A* w oparciu o kryterium czasu,
c) algorytm A* w oparciu o kryterium przesiadek
d) modyfikację A* celem optymalizacji czasu wykonania lub końcowego kosztu

W ramach drugiego zadania mieliśmy zaimplementować:

a) algorytm rozwiązujący problem odwiedzania wierzchołków w oparciu o przeszukiwanie Tabu bez ograniczenia na rozmiar tablicy T
b) modyfikację przeszukiwania Tabu o dobór długości tablicy T w zależności od długości listy L w celu minimalizacji funkcji kosztu
c) modyfikację przeszukiwania Tabu o aspirację w celu minimalizacji funkcji kosztu
d) rozszerzenie przeszukiwania Tabu poprzez dobór strategii próbkowania sąsiedztwa bieżącego rozwiązania, które pozwoli na minimalizację funkcji kosztu i skrócenie czasu działania algorytmu

Opis teoretyczny

Algorytm Dijkstry to algorytm znajdowania najkrótszych ścieżek w grafie ważonym z jednym źródłem. Algorytm działa poprzez utrzymanie zbioru wierzchołków o najkrótszej odległości od źródła oraz aktualizację tych odległości wraz z dodawaniem kolejnych wierzchołków do zbioru. Efektem końcowym będzie zbiór wierzchołków, które pozwolą na ustalenie optymalnej trasy z dowolnie wybranego punktu do innego dowolnie wybranego punktu.

Algorytm A* jest heurystycznym algorytmem służącym do znajdowania najkrótszej ścieżki. Jest to rozbudowany algorytm Dijkstry o dodatkowy moduł estymacji kosztu ścieżki do celu. Wówczas optymalizowaną funkcją jest funkcja f = g + h, gdzie g to funkcja kosztu, a h to wybrana heurystyka. Warto pamiętać, że działanie A* jest uzależnione od wybranej heurystyki. W przypadku wyboru nieoptymalnej heurystyki algorytm może zwracać nieoptymalne wyniki. Rezultatem końcowym A* będzie zwrócenie optymalnej ścieżki od punktu początkowego do końcowego.

Algorytmy przeszukiwania Tabu to metaheurystyka, które polega na iteracyjnym przeszukiwaniu sąsiedztwa aktualnego rozwiązania z uwzględnieniem zestawu ruchów, które są niedozwolone (tabu). Główną zaletą przeszukiwania Tabu jest to, że mimo użycia dodatkowej pamięci na przechowywanie niedozwolonych ruchów to nie testujemy powtórnie tych samych lub podobnych rozwiązań dzięki czemu jesteśmy się w stanie wydostać z lokalnego optimum. Rezultatem końcowym przeszukiwania Tabu będzie lista wierzchołków dla których sumaryczna funkcja kosztu jest najmniejsza

Przykładowe zastosowanie

Algorytm Dijkstry lub A* można użyć do znalezienia najkrótszej metody dojścia z jednego punktu do drugiego. Praktycznym zastosowaniem tego algorytmu może być np. ustalenie optymalnej trasy pomiędzy dwoma wybranymi punktami (jakie połączenia komunikacyjne musimy wybrać, by dojechać do wybranego punktu jak najszybciej).

Przeszukiwanie tabu może być użyte do ustalenia w jakiej kolejności należy zwiedzać listę punktów, żeby móc jak najszybciej przejść przez wszystkie z nich. Praktycznym przykładem może być sytuacja gdzie zwiedzamy atrakcje turystyczne w mieście. Wówczas zależy nam na tym, żeby jak najmniej czasu poświęcić na podróż pomiędzy atrakcjami. Wtedy możemy użyć przeszukiwanie tabu, aby ustalić w jakiej kolejności powinniśmy zwiedzać atrakcje turystyczne.

Wprowadzone modyfikacje + implementacja

W ramach preprocessingu wczytaliśmy wszystkie dane z użyciem biblioteki pandas. Odrzuciliśmy wszystkie niepotrzebne kolumny i dokonaliśmy modyfikacji danych, żeby potem łatwiej nam było pracować z nimi. Zauważyliśmy, że nie ma wszystkich połączeń nocnych (brakuje połączeń między północą a 3), więc, żeby przyspieszyć dalsze obliczenia założyliśmy, że możemy odrzucić wszystkie połączenia, które są przed czasem naszego odjazdu. Dodatkowo odrzuciliśmy wszystkie połączenia prowadzące donikąd (np. linia jadąca z przystanku pl. Grunwaldzki na przystanek pl. Grunwaldzki), gdyż założyliśmy, że każdy przystanek ma tylko jedną lokalizację. Wszystkie połączenia zostały posortowane po czasie odjazdu i przyjazdu, żeby móc potem szybciej wybierać najbardziej optymalne połączenia

```python
def setup_data(arr_time, filename="connection_graph.csv", remove_night_routes=False):
    df = pd.read_csv(filename, dtype=object, encoding='utf8')
    df.drop(['Unnamed: 0.1', 'Unnamed: 0', 'company'], axis=1, inplace=True)
    df.loc[df['line'] == 'C ', 'line'] = 'C'
    df['departure_time'] = pd.to_datetime(df['departure_time'], format="%H:%M:%S").dt.time
    df['arrival_time'] = pd.to_datetime(df['arrival_time'], format="%H:%M:%S").dt.time
    df['start_stop'] = df['start_stop'].map(lambda x: x.lower())
    df['end_stop'] = df['end_stop'].map(lambda x: x.lower())
    df['start_stop_lat'] = df['start_stop_lat'].map(lambda x: float(x))
    df['start_stop_lon'] = df['start_stop_lon'].map(lambda x: float(x))
    df['end_stop_lat'] = df['end_stop_lat'].map(lambda x: float(x))
    df['end_stop_lon'] = df['end_stop_lon'].map(lambda x: float(x))

    # Sorting values by time to quickly get the quickest connection
    all_possible_connections = df[df['departure_time'] >= arr_time]
    if remove_night_routes:
        all_possible_connections = all_possible_connections[all_possible_connections['departure_time'] <= datetime.strptime("22:0
    # Removing routes from the same stops
    all_possible_connections = all_possible_connections[all_possible_connections['start_stop'] != all_possible_connections['end_s
    # Sorting values by time to quickly get the quickest connection
    all_possible_connections = all_possible_connections.sort_values(['departure_time', 'arrival_time'])

    Node.ALL_CONNECTIONS_DF = all_possible_connections

    return df
```

Dane przechowujemy z użyciem dwóch klas:

a) Klasa Node (węzeł grafu) zawierająca:
   a. Nazwę przystanku
   b. Czas dojazdu na przystanek
   c. Linia, która była użyta do przyjazdu na przystanek
   d. Połączenia wychodzące z danego przystanku (lista zawierająca węzły grafu). Bierzemy tylko najbardziej optymalne połączenia z danego przystanku (np. jak dana linia odjeżdża z przystanku co 15 minut to zapisujemy tylko pierwsze połączenie)
   e. Lokalizację przystanku (długość i szerokość geograficzna)
b) Klasa Edge (krawędź grafu) zawierająca:
   a. Nazwę linii
   b. Czas odjazdu z przystanku początkowego
   c. Czas dojazdu na przystanek końcowy
   d. Nazwę przystanku początkowego
   e. Nazwę przystanku końcowego

Cały proces wykonania zadania 1 został przedstawiony na poniższym zrzucie ekranu

```python
def task1a(start_stop, end_stop, arrival_time):
    all_connections = setup_data(arrival_time, remove_night_routes=True)
    start_node = Node(start_stop, arrival_time)

    time_start = time.time()
    cost, path = get_shortest_path_dijkstra(start_node, end_stop)
    print(f'Execution time: {time.time() - time_start:.2f} seconds')
    print_results(start_stop, arrival_time, path, end_stop)
    print(f'Cost: {cost}')
    draw_results(all_connections, path)
```

Wpierw dokonujemy preprocessingu danych. Potem deklarujemy nasz początkowy węzeł grafu. Potem wykonujemy podany algorytm zdobywając ścieżkę (listę zawierającą węzły prowadzącego do celu) i koszt. Wykorzystując podane dane wyświetlamy połączenia w terminalu oraz wyświetlamy prostą mapę na której wyświetlamy nasze połączenie i wszystkie przystanki.

Implementacja algorytmu Dijkstry znajduje się poniżej

```python
def dijkstra(graph, start_node):
    distances = defaultdict(lambda: float('inf'))
    distances[start_node.stop] = 0
    prev_nodes = {start_node: None}

    pq = [(0, start_node)]
    while pq:
        curr_dist, curr_node = heapq.heappop(pq)

        # Add dynamically new nodes to the graph
        curr_node.generate_edges()
        graph.add_neighbour_nodes(curr_node)

        if curr_dist > distances[curr_node.stop]:
            continue

        # Avoid crash when arriving to Zorawina
        if curr_node.stop in graph.graph_dict:
            for neighbour, weight in graph.graph_dict[curr_node.stop]:
                new_dist = curr_dist + weight
                if new_dist < distances[neighbour.stop]:
                    distances[neighbour.stop] = new_dist
                    prev_nodes[neighbour] = curr_node
                    heapq.heappush(pq, (new_dist, neighbour))

    return distances, prev_nodes
```

Nie wprowadziliśmy wiele modyfikacji do algorytmu Dijkstry. Jedyna zmiana warta uwagi to fakt, że nie mamy całego grafu na początku zadania tylko generujemy dynamicznie dodatkowe węzły. Warto zauważyć, że jest to konieczne gdyż w naszej implementacji znaczenie ma nie tylko czas przejazdu między przystankami, ale też czas dojazdu na przystanek.

Implementacja algorytmu A* dla kryterium przesiadkowego znajduje się poniżej

```python
def astar_stops(start_node, end_node, neighbors_fn):
    cost_so_far = defaultdict(lambda: float('inf'))
    cost_so_far[start_node.stop] = 0

    came_from = {start_node: None}

    front = [(0, start_node)]
    while front:
        _, curr_node = heapq.heappop(front)

        # Add dynamically new nodes to the graph
        # Line prioritised should be based on the stops
        curr_node.generate_edges(curr_node.line_arr)
        neighbors_fn.add_neighbour_nodes(curr_node)

        # We can assume that there is always the end stop
        if curr_node.stop == end_node.stop:
            return came_from, cost_so_far[end_node.stop]

        # Avoid crash when arriving to Zorawina
        if curr_node.stop in neighbors_fn.graph_dict:
            for neighbor in neighbors_fn.graph_dict[curr_node.stop]:
                new_cost = cost_so_far[curr_node.stop] + time_cost(curr_node, neighbor)
                if new_cost < cost_so_far[neighbor.stop]:
                    cost_so_far[neighbor.stop] = new_cost
                    priority = new_cost + stop_heuristic(curr_node, neighbor)
                    heapq.heappush(front, (priority, neighbor))
                    came_from[neighbor] = curr_node
```

Też nie ma wielu różnic w implementacji.

Dla A* użyliśmy następujących heurystyk:

a) Manhattan Distance dla A* optymalizującego czas
b) Stop heuristic dla A* optymalizującego liczbę przesiadek, który w pierwszej kolejności priorytetyzuje połączenia tą samą linią, potem priorytetyzuje odjazdy z przystanku z którego można dojechać do końca. Potem priorytetyzuje przystanki od tego z którego odjeżdża najwięcej połączeń do tego z którego odjeżdża najmniej połączeń.

Celem optymalizacji A* dla kryterium przesiadkowego wprowadziliśmy następującą modyfikację – jeżeli jest możliwość dojechania na przystanek końcowy z analizowanego przystanku to wówczas tworzymy resztę trasy wykorzystując wiedzę na temat tego jaka linia dojeżdża na przystanek końcowy

Cały proces wykonania zadania 2 został przedstawiony na poniższym zrzucie ekranu

```python
def task2c(start_stop, optimalization_criteria, arrival_time,  *list_stops):
    all_connections = setup_data(arrival_time, remove_night_routes=True)

    stops = [Node(stop, None) for stop in list_stops]
    stops_dict = {0: start_stop, len(stops)+1: start_stop}
    best_solution, best_solution_cost = tabu_search_aspiration(stops)

    time_start = time.time()
    print("\nStops from the first to last")
    print(f'{start_stop}, ', end='')
    for i, idx in enumerate(best_solution, 1):
        stops_dict[i] = stops[idx].stop
        print(f'{stops[idx].stop}, ', end='')
    print(start_stop)
    print("Best solution cost: {}".format(best_solution_cost))
    print(f'Execution time: {time.time() - time_start:.8f} seconds\n')

    start_node = Node(stops_dict[0], arrival_time)
    start_node.generate_edges()
    path = [start_node]
    cost = 0
    cur_arr_time = arrival_time
    for i in range(len(stops_dict)-1):
        if optimalization_criteria == 't':
            part_cost, part_path = get_shortest_path_astar_time(Node(stops_dict[i], cur_arr_time), stops_dict[i+1])
        elif optimalization_criteria == 's':
            part_cost, part_path = get_shortest_path_astar_stops_mod(Node(stops_dict[i], cur_arr_time, '-1'), stops_dict[i+1])
        else:
            raise Exception("Incorrect optimalization criteria")

        print(f'\nConnection from {stops_dict[i]} to {stops_dict[i+1]}')
        print_results(stops_dict[i], cur_arr_time, part_path, stops_dict[i+1])

        # Don't remember the first stop to avoid stops duplicates
        path.extend(part_path[1:])
        cost += part_cost
        cur_arr_time = path[-1].arr_time

    print()
    print(f'Cost: {cost}')
    draw_results(all_connections, path)
```

Wpierw dokonujemy preprocessingu danych. Na podstawie podanej listy przystanków i z użyciem przeszukiwania tabu decydujemy w jakiej kolejności należy zwiedzać wszystkie przystanki.

Żeby uzyskać optymalne połączenia pomiędzy przystankami korzystamy z już zaimplementowanego algorytmu A*.

Na bieżąco wyświetlamy wszystkie połączenia komunikacyjne w terminalu a na sam koniec wyświetlamy mapę zawierającą całą trasę.

Implementacja przeszukiwania Tabu znajduje się poniżej

```python
def tabu_search(stops):
    random.seed(121)

    n_stops = len(stops)

    max_iterations = math.ceil(1.1*(n_stops**2))
    turns_improved = 0
    improve_thresh = 2 * math.floor(math.sqrt(max_iterations))

    tabu_list = []

    distances = [[distance(stops[i], stops[j]) for i in range(n_stops)] for j in range(n_stops)]

    total = 0
    for i in range(n_stops):
        for j in range(n_stops):
            total += distances[i][j]

    current_solution = list(range(n_stops))
    random.shuffle(current_solution)
    best_solution = current_solution[:]
    best_solution_cost = sum([distances[current_solution[i]][current_solution[(i+1) % n_stops]] for i in range(n_stops)])

    for iteration in range(max_iterations):
        if turns_improved > improve_thresh:
            break
        best_neighbor = None
        best_neighbor_cost = float('inf')
        tabu_candidate = (0, 0)

        for i in range(n_stops):
            for j in range(i+1, n_stops):
                neighbor = current_solution[:]
                neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
                neighbor_cost = sum([distances[neighbor[i]][neighbor[(i+1)%n_stops]] for i in range(n_stops)])
                if (i, j) not in tabu_list:
                    if neighbor_cost < best_neighbor_cost:
                        best_neighbor = neighbor[:]
                        best_neighbor_cost = neighbor_cost
                        tabu_candidate = (i, j)

        if best_neighbor is not None:
            current_solution = best_neighbor[:]
            tabu_list.append(tabu_candidate)
            if best_neighbor_cost < best_solution_cost:
                best_solution = best_neighbor[:]
                best_solution_cost = best_neighbor_cost
                turns_improved = 0
            else:
                turns_improved = turns_improved + 1

        print("Iteration {}: Best solution cost = {}".format(iteration, best_solution_cost))

    return best_solution, best_solution_cost
```

Nie ma wiele różnic w implementacji przeszukiwania Tabu.

W ramach realizacji pozostałych podpunktów dodaliśmy:

a) Ograniczenie na rozmiar długość tablicy w oparciu o listę przystanków
b) Aspirację
c) Strategię próbkowania sąsiedztwa w oparciu o losowy wybór (losowo decydujemy jakich sąsiadów analizujemy)

Materiały dodatkowe

W ramach zadania został wykorzystany kod udostępniony przez prowadzącą, który został potem dostosowany pod dane.

Dodatkowo wykorzystaliśmy instrukcję do zadania celem opisu teoretycznego implementowanych algorytmów.

Opis wykorzystanych bibliotek

Do zadania zostały wykorzystane następujące wbudowane biblioteki:

a) heaqp – biblioteka implementująca kopiec
b) collections – biblioteka zawierająca różne kolekcje. My użyliśmy defaultdict, który sprawia, że w momencie w którym nie mamy klucza w słowniku zwracana jest domyślna ustawiona przez nas wartość
c) datetime – biblioteka pozwalająca na przetwarzanie czasu
d) time – biblioteka pozwalająca liczyć czas
e) random – biblioteka pozwalająca na generowanie losowych liczb, losowe sortowanie i ustawienie tzw. seeda
f) math – biblioteka pozwalająca wykonywać operacje matematyczne np. liczenie pierwiastka

Oraz następujące dodatkowe biblioteki:

a) pandas – biblioteka pozwalająca wczytywać dane z pliku .csv i przechowywać je jako dataframe
b) numpy – biblioteka pozwalająca obsługiwać wiersze w dataframe
c) matplotlib – biblioteka pozwalająca na rysowanie elementów w oknie

Napotkane problemy implementacyjne

Mieliśmy następujące problemy implementacyjne:

a) Kwestia wygenerowania wszystkich połączeń. Warto zauważyć, że nie przyjęliśmy żadnych uproszczeń względem danych (np. nie obliczyliśmy średniego czasu połączenia między parą przystanków)
b) W Żórawinie dla linii 903 i 913 przystanek końcowy nie jest przystankiem początkowym kursu powrotnego. Generuje to błędy związane z tym, że nie możemy sensownie ustalić tam trasy
c) W pliku nie ma zawartych wszystkich linii nocnych. To sprawia, że np. jeżeli A* ustali, że linia 253 pozwala na bezpośrednie połączenie z przystanku leśnica do przystanku bartoszowice to wybór tego połączenia może skutkować błędami gdyż nie mamy wystarczająco dużo informacji, by ustalić trasę od początku do końca

d) Są też kursy wariantowe (np. kurs do zajezdni) lub kursy pomijające przystanki (np. linia C pomija część przystanków na których zatrzymują się inne linie). Generuje to problemy gdyż czasami program wybiera nieoptymalne czasowo połączenia tylko dlatego, że przesiadka na linię pomijającą kilka przystanków zmniejsza najmocniej dystans do przystanku końcowego. Jest to szczególnie zauważalne gdy dana linia pomija kilka przystanków obok przystanku końcowego (np. linia C pomija przystanek Reja i od razu z przystanku katedra dojeżdża na pl. Grunwaldzki).

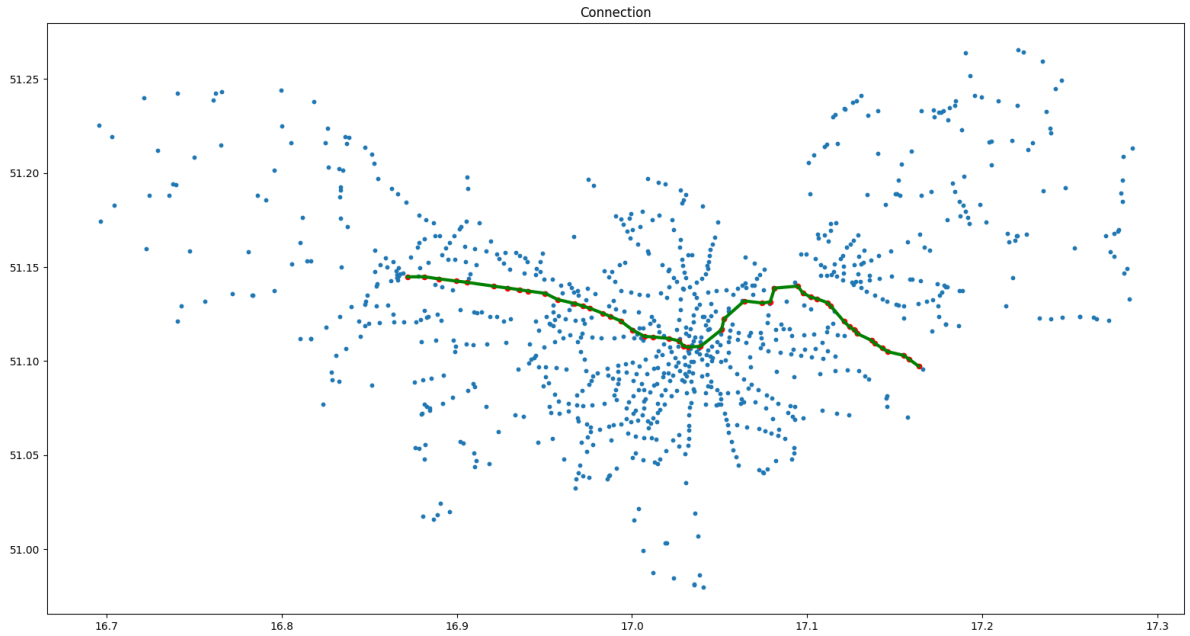
Tak rozwiązaliśmy powyższe problemy implementacyjne:

a) Dynamicznie generujemy nowe połączenia (krawędzie) od podanego przystanku (węzła). Dbamy o to, żeby analizować tylko krawędzie przypisane do danego węzła
b) Nie analizujemy połączeń do Żórawiny
c) Eliminujemy wszystkie połączenia po 22:00. Wówczas mamy gwarancję, że w naszych danych nie będzie kursów nocnych
d) Założyliśmy, że może dochodzić do takich sytuacji i nie będziemy próbować tego naprawiać, gdyż naprawa wymagałaby dokładnego przefiltrowania danych. Niestety takie przefiltrowanie danych doprowadziłoby do utraty wielu informacji i mogłoby bardziej zaszkodzić niż pomóc

Przykładowe działanie programu

Poniżej znajdują się rezultaty połączenia Leśnica-Wojnów dla A* gdzie kryterium optymalizacyjnym jest czas

```
(venv) C:\Users\Wiktor\Desktop\Studia - Biezace\SI\Lab1>python main.py
Execution time: 32.85 seconds
Start at leśnica at 10:00:00
Travel with line 148 from leśnica to złotnicka from 10:00:00 to 10:03:00
Travel with line 10 from złotnicka to galeria dominikańska from 10:03:00 to 10:41:00
Wait at galeria dominikańska from 10:41:00 to 11:05:00
Travel with line 904 from galeria dominikańska to kromera from 11:05:00 to 11:16:00
Travel with line 11 from kromera to kwidzyńska from 11:16:00 to 11:18:00
Wait at kwidzyńska from 11:18:00 to 11:19:00
Travel with line 118 from kwidzyńska to wojnów from 11:19:00 to 11:39:00
Arrived at wojnów at 11:39:00
Cost: 4320
```



Analiza wyników dla różnych funkcji

Postanowiliśmy przeanalizować działanie Dijkstry i A* dla różnych kryteriów dla relacji Kwiska – pl. Grunwaldzki o godz. 10:00

Przetestowaliśmy jak będzie wyglądało połączenie dla:

a) Algorytmu Dijkstry

```
(venv) C:\Users\Wiktor\Desktop\Studia - Biezace\SI\Lab1>python main.py
Execution time: 43.61 seconds
Start at kwiska at 10:00:00
Wait at kwiska from 10:00:00 to 10:02:00
Travel with line 143 from kwiska to wejherowska from 10:02:00 to 10:04:00
Travel with line C from wejherowska to pl. grunwaldzki from 10:04:00 to 10:25:00
Arrived at pl. grunwaldzki at 10:25:00
Cost: 1500
```

b) A* gdzie czas jest głównym kryterium

```
(venv) C:\Users\Wiktor\Desktop\Studia - Biezace\SI\Lab1>python main.py
Execution time: 10.38 seconds
Start at kwiska at 10:00:00
Wait at kwiska from 10:00:00 to 10:02:00
Travel with line 143 from kwiska to wejherowska from 10:02:00 to 10:04:00
Travel with line C from wejherowska to pl. grunwaldzki from 10:04:00 to 10:25:00
Arrived at pl. grunwaldzki at 10:25:00
Cost: 1500
```

c) A* gdzie liczba przesiadek jest głównym kryterium

```
(venv) C:\Users\Wiktor\Desktop\Studia - Biezace\SI\Lab1>python main.py
Execution time: 3.47 seconds
Start at kwiska at 10:00:00
Wait at kwiska from 10:00:00 to 10:06:00
Travel with line 10 from kwiska to katedra from 10:06:00 to 10:29:00
Wait at katedra from 10:29:00 to 10:52:00
Travel with line C from katedra to pl. grunwaldzki from 10:52:00 to 10:55:00
Arrived at pl. grunwaldzki at 10:55:00
Cost: 12040
```

d) A* zmodyfikowanego, aby zoptymalizować czas wykonania

```
(venv) C:\Users\Wiktor\Desktop\Studia - Biezace\SI\Lab1>python main.py
Execution time: 1.39 seconds
Start at kwiska at 10:00:00
Wait at kwiska from 10:00:00 to 10:06:00
Travel with line 10 from kwiska to pl. grunwaldzki from 10:06:00 to 10:34:00
Arrived at pl. grunwaldzki at 10:34:00
Cost: 2040
```

Dane dotyczące rezultatów algorytmów zebraliśmy w poniższej tabelce

Algorytm	Czas wykonania (w s)	Koszt	Uwagi
Dijkstra	43.61	1500	
A* czas	10.38	1500	Ta sama trasa i koszt jak dla algorytmu Dijkstry
A* przesiadki	3.47	12040	
A* modyfikacja	0.72	2040	Optymalizacja A* przesiadki

Widzimy, że najwięcej czasu wykonywał się algorytm Dijkstry. Wynika to z tego, że algorytm bada każde połączenie do danego przystanku, żeby uniknąć wejścia w lokalne minimum. Przez to badane są też połączenia, które nas nie interesują (np. badamy połączenia na Wojnowie gdzie nigdy nie będziemy przejeżdżać przez tą dzielnicę).

A* w momencie dotarcia do przystanku końcowego przerywa dalszą analizę połączeń dzięki czemu jest szybszy. W powyższym przypadku widzimy, że trasa jest taka sama, ale nie zawsze tak musi być gdyż A* może wejść w lokalne minimum.

Ciekawym przypadkiem do analizy jest A* przesiadki. Widzimy, że czas wykonania w porównaniu do A* czas jest o wiele szybszy. Można to uzasadnić tym, że nie analizujemy pomniejszych przystanków tylko idziemy głównie po trasie linii, którą aktualnie jedziemy. Jako, że Kwiska i Pl. Grunwaldzki to duże węzły przesiadkowe to bezproblemowo jesteśmy w stanie znaleźć połączenie pomiędzy tymi dwoma przystankami.

Niestety linia pokazana przez A* przesiadki nie jest najbardziej optymalna gdyż nie wykorzystujemy informacji, że z przystanku Kwiska linia 10 dojeżdża bezpośrednio do celu.

Jak wykorzystamy ten fakt do poszukiwania trasy to zobaczymy, że czas wykonania A* się zmniejszy, gdyż już na przystanku Kwiska wiemy, że sens ma tylko analiza połączeń linią numer 10.

Nawet jeżeli bylibyśmy na przystanku z którego nie ma bezpośredniego dojazdu do celu to i tak czas wykonania A* byłby szybszy, bo po dojechaniu do większości węzłów przesiadkowych bylibyśmy w stanie od razu wybrać odpowiednie połączenie do celu.

Przetestowaliśmy też działanie Tabu Search gdzie przystankiem początkowym była Leśnica o godz. 10:00, a przystankami pośrednimi były: pl. Grunwaldzki, klecina, racławicka, oporów

a) Algorytm bez modyfikacji

```
(venv) C:\Users\Wiktor\Desktop\Studia - Biezace\SI\Lab1>python main.py
Iteration 0: Best solution cost = 0.20627089698053516
Iteration 1: Best solution cost = 0.20627089698053516
Iteration 2: Best solution cost = 0.20627089698053516
Iteration 3: Best solution cost = 0.20627089698053516
Iteration 4: Best solution cost = 0.20627089698053516
Iteration 5: Best solution cost = 0.20627089698053516
Iteration 6: Best solution cost = 0.20627089698053516
Iteration 7: Best solution cost = 0.20627089698053516
Iteration 8: Best solution cost = 0.20627089698053516
Iteration 9: Best solution cost = 0.20627089698053516
Iteration 10: Best solution cost = 0.20627089698053516
Iteration 11: Best solution cost = 0.20627089698053516
Iteration 12: Best solution cost = 0.20627089698053516
Iteration 13: Best solution cost = 0.20627089698053516
Iteration 14: Best solution cost = 0.20627089698053516
Iteration 15: Best solution cost = 0.20627089698053516
Iteration 16: Best solution cost = 0.20627089698053516
Iteration 17: Best solution cost = 0.20627089698053516

Stops from the first to last
leśnica, klecina, racławicka, oporów, pl. grunwaldzki, leśnica
Best solution cost: 0.20627089698053516
Execution time: 0.00608802 seconds
```

b) Algorytm z ograniczeniem na długość tablicy

```
(venv) C:\Users\Wiktor\Desktop\Studia - Biezace\SI\Lab1>python main.py
Iteration 0: Best solution cost = 0.20627089698053516
Iteration 1: Best solution cost = 0.20627089698053516
Iteration 2: Best solution cost = 0.20627089698053516
Iteration 3: Best solution cost = 0.20627089698053516
Iteration 4: Best solution cost = 0.20627089698053516
Iteration 5: Best solution cost = 0.20627089698053516
Iteration 6: Best solution cost = 0.20627089698053516
Iteration 7: Best solution cost = 0.20627089698053516
Iteration 8: Best solution cost = 0.20627089698053516
Iteration 9: Best solution cost = 0.20627089698053516

Stops from the first to last
leśnica, klecina, racławicka, oporów, pl. grunwaldzki, leśnica
Best solution cost: 0.20627089698053516
Execution time: 0.00099969 seconds
```

c) Algorytm z kryterium aspiracji

```
(venv) C:\Users\Wiktor\Desktop\Studia - Biezace\SI\Lab1>python main.py
Iteration 0: Best solution cost = 0.20627089698053516
Iteration 1: Best solution cost = 0.20627089698053516
Iteration 2: Best solution cost = 0.20627089698053516
Iteration 3: Best solution cost = 0.20627089698053516
Iteration 4: Best solution cost = 0.20627089698053516
Iteration 5: Best solution cost = 0.20627089698053516
Iteration 6: Best solution cost = 0.20627089698053516
Iteration 7: Best solution cost = 0.20627089698053516
Iteration 8: Best solution cost = 0.20627089698053516
Iteration 9: Best solution cost = 0.20627089698053516

Stops from the first to last
leśnica, klecina, racławicka, oporów, pl. grunwaldzki, leśnica
Best solution cost: 0.20627089698053516
Execution time: 0.00404930 seconds
```

d) Algorytm z losowym próbkowaniem

```
(venv) C:\Users\Wiktor\Desktop\Studia - Biezace\SI\Lab1>python main.py
Iteration 0: Best solution cost = 0.20627089698053516
Iteration 1: Best solution cost = 0.20627089698053516
Iteration 2: Best solution cost = 0.20627089698053516
Iteration 3: Best solution cost = 0.20627089698053516
Iteration 4: Best solution cost = 0.20627089698053516
Iteration 5: Best solution cost = 0.20627089698053516
Iteration 6: Best solution cost = 0.20627089698053516
Iteration 7: Best solution cost = 0.20627089698053516
Iteration 8: Best solution cost = 0.20627089698053516
Iteration 9: Best solution cost = 0.20627089698053516
Iteration 10: Best solution cost = 0.20627089698053516
Iteration 11: Best solution cost = 0.20627089698053516
Iteration 12: Best solution cost = 0.20627089698053516
Iteration 13: Best solution cost = 0.20627089698053516
Iteration 14: Best solution cost = 0.20627089698053516
Iteration 15: Best solution cost = 0.20627089698053516
Iteration 16: Best solution cost = 0.20627089698053516
Iteration 17: Best solution cost = 0.20627089698053516

Stops from the first to last
leśnica, racławicka, klecina, pl. grunwaldzki, oporów, leśnica
Best solution cost: 0.20627089698053516
Execution time: 0.00404382 seconds
```

Dane dotyczące rezultatów algorytmów zebraliśmy w poniższej tabelce (w tabeli nie uwzględniliśmy kosztu gdyż był taki sam w każdym wykonaniu funkcji)

Modyfikacja	Czas wykonania (w s)	Liczba iteracji
Brak	0,006088	17
Długość tablicy	0,000999	9
Aspiracja	0,004049	9
Losowe próbkowanie	0,004043	17

Możemy zauważyć, że najdłużej wykonywał się Tabu Search bez wprowadzonej modyfikacji. Wynika to z tego, że analizujemy wszystkie możliwości oraz dodatkowo przechowujemy wszystkie zakazane rozwiązania (warto zauważyć, że wszystkie zakazane rozwiązanie przechowujemy w liście, a operacja dodawania elementu do listy jest czasochłonna). Dodatkowo nie mamy żadnych mechanizmów, które by sprawiły, że unikniemy wejścia w lokalne minimum.

Jedną z możliwych metod optymalizacji jest ograniczenie długości tablicy. Widzimy, że rezultat końcowy jest ten sam, ale nie poświęcamy już tyle czasu na dodawanie elementów do coraz to dłuższej listy.

Kolejną metodą optymalizacji jest użycie kryterium aspiracji. Dzięki użyciu kryterium aspiracji analizujemy nie tylko najbardziej optymalne w danym momencie rozwiązanie, ale też te, które są „wystarczająco dobre". Dzięki temu unikamy sytuacji w której wchodzimy w lokalne minimum co pozwoli szybciej dojść do najlepszego rozwiązania

Ostatnią metodą optymalizacji, którą możemy użyć jest próbkowanie. Widzimy, że mimo, że użyliśmy losowego próbkowania to byliśmy szybciej w stanie dojść do optymalnego rozwiązania gdyż analizowaliśmy mniej możliwych rozwiązań (mimo że mieliśmy tą samą liczbę iteracji co w przypadku Tabu Search bez modyfikacji).

Podsumowanie

W ramach zadania zaimplementowaliśmy algorytmy: Dijkstra, A* i przeszukiwanie Tabu.

Dijkstra w porównaniu do A* zawsze będzie zwracał najbardziej optymalne połączenia, jednak długi czas wykonania sprawia, że trudno znaleźć praktyczne zastosowanie tego algorytmu w aplikacjach wyszukujących trasę pomiędzy punktami. A* mimo, momentami, mniej optymalnych połączeń wykonuje się o wiele szybciej niż Dijkstra, więc może być używany w aplikacjach do wyszukiwania tras pomiędzy punktami.

Warto też zwrócić uwagę na to jak modyfikacje algorytmów wpłynęły znacząco na polepszenie czasu wykonania. W warunkach symulacyjnych nie ma większej różnicy między 1 a 10 sekundami w czasie wykonania, gdyż głównie liczy się dla nas czy wynik końcowy jest dobry. Jednak w aplikacjach dla użytkowników (np. Google Maps) taka różnica w czasie będzie mocno zauważalna i może wpłynąć na przyjemność z korzystania z aplikacji.

Warto więc pamiętać o tym, żeby nie poprzestać na tym, że zaimplementowaliśmy algorytm tylko próbować zoptymalizować jego działanie (np. wykorzystując informację na temat danych lub modyfikując działania algorytmu), żeby czas wykonania algorytmu nie był barierą w jego używaniu.