



Politechnika Wrocławska

SPRAWOZDANIE Z ZAJĘĆ LABORATORYJNYCH
SZTUCZNA INTELIGENCJA I INŻYNIERIA WIEDZY

Lista 01
Problemy optymalizacyjne i algorytmy przeszukiwania

Tomasz Mroczko, 266604

8 kwietnia 2024

Spis treści

1	Dane	2
1.1	Zawartość	2
1.2	Wstępna analiza	2
1.3	Reprezentacja	3
2	Zadanie 1	4
2.1	Dijkstra - kryterium czasu	4
2.1.1	Opis	4
2.1.2	Działanie / kod	4
2.1.3	Wnioski	5
2.2	A* - kryterium czasu	5
2.2.1	Opis	5
2.2.2	Działanie / kod	6
2.2.3	Wnioski	7
2.2.4	Napotkane problemy	8
2.3	A* - kryterium przesiadek	8
2.3.1	Opis	8
2.3.2	Działanie / kod	8
2.3.3	Wnioski	8
2.4	Modyfikacja algorytmu A*	8
2.4.1	Opis	8
2.4.2	Kod	9
2.4.3	Wnioski	9
2.5	Przykładowe wyniki	9
2.5.1	Kryterium czasu	9
2.5.2	Kryterium przesiadek	10
3	Zadanie 2	11
3.1	Algorytm Tabu Search, bez ograniczenia tablicy T	11
3.1.1	Opis	11
3.1.2	Kod	12
3.1.3	Przykładowe działanie	13
3.2	Dobór długości listy T	14
3.2.1	Opis	14
3.2.2	Kod	14

1 Dane

1.1 Zawartość

Dane zawarte w pliku *connection_graph.csv* zawierały informacje o połączeniach komunikacyjnych pomiędzy przystankami komunikacji miejskiej we Wrocławiu. Każdy wiersz zawiera informacje o połączeniu pomiędzy dwoma przystankami w postaci:

- id - identyfikator połączenia
- company - przewoźnik
- line - numer linii
- departure_time - czas odjazdu w formacie HH:MM:SS
- arrival_time - czas przyjazdu w formacie HH:MM:SS
- start_stop - nazwa przystanku początkowego
- end_stop - nazwa przystanku końcowego
- start_stop_lat - szerokość geograficzna przystanku początkowego
- end_stop_lat - szerokość geograficzna przystanku końcowego
- start_stop_lon - długość geograficzna przystanku początkowego
- end_stop_lon - długość geograficzna przystanku końcowego

Rozmiar pliku wynosił 113MB, i zawierał on 996 521 wierszy.

1.2 Wstępna analiza

Pierwotna analiza danych, wykazała, że w pliku znajduje się bardzo dużo powtarzających się wierszy, które różnią się jedynie id. Sugeruje to, że dane zostały sztucznie "wydłużone", prawdopodobnie w celu jaśniejszego zaznaczenia różnicy pomiędzy szybkością działania algorytmów poprzez wydłużenie czasu przeszukiwania. Jednak jest to tylko domysł, a nie potwierdzona informacja.

```
1 cat connection_graph.csv | cut -d',' -f2- | sort | uniq | wc -l
2
```

Za pomocą tej komendy, sprawdzono unikalność wierszy w pliku. Znaczna część wierszy jest duplikatami. Ustalono, że jedynie 455 232 wierszy jest unikalnych (mniej niż połowa). Pomimo tego faktu, do dalszej analizy użyto pełnego pliku, bez usuwania duplikatów.

Warto zaznaczyć, że godziny odjazdu po 23:59:59 były zapisywane jako 24:00:00, co jest niepoprawne w kontekście zapisu czasu w formacie 24-godzinnym. Ponieważ przyjęto reprezentację odjazdu jako liczba minut od północy, nie było to problemem w dalszej analizie.

1.3 Reprezentacja

Dane reprezentowane są za pomocą kilku struktur danych w celu ułatwienia organizacji i przetwarzania. Dla przystanku oraz połączenia użyto `NamedTuple` zamiast klasy, ze względu na prostotę, niemutowalność i szybkość działania. Takie podejście pozwoliło uzyskać namiastkę typowania poprzez adnotacje typów w Pythonie, co ułatwiło pisanie kodu, a także pozwoliło **LSP** na podpowiadanie nazw pól oraz analizowanie kodu.

- **Stop** - reprezentuje pojedynczy przystanek. Przystanek jest reprezentowany przez nazwę, szerokość i długość geograficzną, jednak do porównywania przystanków używane jest jedynie pole z nazwą. Identyfikowanie ich także poprzez długość i szerokość geograficzną powodowało duże problemy związane z przeszukiwaniem połączeń (kierunki odjazdów, przystanki autobusowe a tramwajowe, wprowadzanie nazw przystanków, itp.).

```
1 class Stop(NamedTuple):
2     name: str
3     lat: float
4     lon: float
5
6     def __eq__(self, other: object) -> bool:
7         if not isinstance(other, Stop):
8             return False
9         return self.name == other.name
10
11     def __hash__(self) -> int:
12         return hash(self.name)
13
14     def __repr__(self) -> str:
15         return f"Stop: {self.name}"
16
```

- **Route** - reprezentuje połączenie między przystankami. Ze względu na wydajność oraz prostotę obliczeń, czas odjazdu oraz przyjazdu został przekształcony na minutę od północy.

```
1 class Route(NamedTuple):
2     line: str
3     start_stop: Stop
4     end_stop: Stop
5     departure_min: int
6     arrival_min: int
7     start_stop_lat: float
8     start_stop_lon: float
9     end_stop_lat: float
10    end_stop_lon: float
11
12    def __eq__(self, other) -> bool:
13        if not isinstance(other, Route):
14            return False
15        return self.line == other.line and self.end_stop == other.end_stop
16
17    def __hash__(self) -> int:
18        return hash((self.line, self.end_stop.name))
19
20    def __repr__(self) -> str:
21        return f"{self.start_stop} -> {self.end_stop}: {self.departure_min}"

```

- **Graph** - klasa która agreguje przystanki oraz połączenia z nich odjeżdżające. Posiada także zbiór unikatowych nazw linii, które przyjeżdżają na dany przystanek, używany w celu ulepszenia przeszukiwania pod kątem ilości przesiadek.

```
1 class Graph:
2     def __init__(self):
3         self.departures: dict[Stop, list[Route]] = {}
4         self.arriving_line_names: dict[Stop, set[str]] = {}

```

```

5
6     def add_route(self, route: Route):
7         if route.start_stop not in self.departures:
8             self.departures[route.start_stop] = []
9             self.arriving_line_names[route.start_stop] = set()
10
11         if route.end_stop not in self.departures:
12             self.departures[route.end_stop] = []
13             self.arriving_line_names[route.end_stop] = set()
14
15         self.departures[route.start_stop].append(route)
16         self.arriving_line_names[route.end_stop].add(route.line)
17
18     def get_stop(self, name: str) -> Optional[Stop]:
19         for stop in self.departures:
20             if stop.name == name:
21                 return stop
22         return None
23
24     def is_direct_connection(self, route: Route, destination: Stop) -> bool:
25         return route.line in self.arriving_line_names[destination]

```

2 Zadanie 1

2.1 Dijkstra - kryterium czasu

2.1.1 Opis

Celem było zaimplementowanie algorytmu dijkstry, który znajduje najkrótszą ścieżkę pomiędzy dwoma przystankami, biorąc pod uwagę *jedynie* czas podróży.

2.1.2 Działanie / kod

Algorytm działa na zasadzie zachłannego przeszukiwania grafu w głąb. W każdym kroku wybierany jest wierzchołek, który ma najkrótszą odległość od źródła. Relizowane to jest za pomocą kolejki priorytetowej, która pozwala na wybieranie wierzchołka o najmniejszej wartości funkcji kosztu.

Dla każdej krawędzi liczona jest liczba minut potrzebna na przejazd do danego przystanku. Jeśli odległość ta jest mniejsza niż dotychczasowa, to aktualizowana jest odległość oraz poprzednik, który pozwoli na odtworzenie ścieżki, a następnie wierzchołek dodawany jest do kolejki priorytetowej. Algorytm kończy się gdy dotrzemy do celu lub gdy kolejka priorytetowa jest pusta.

- Dijkstra

```

1 @route_info_decorator
2 def dijkstra(graph: Graph, start: Stop, end: Stop, departure_min: int) -> SearchResult:
3     costs: dict[Stop, float] = {start: 0}
4     came_from: dict[Stop, Optional[Route]] = {start: None}
5
6     pq: PriorityQueue[tuple[float, Stop]] = PriorityQueue()
7     pq.put((0, start))
8
9     visited_stops_counter: int = 0
10    while not pq.empty():
11        visited_stops_counter += 1
12        curr_cost, curr_stop = pq.get()
13        prev_route: Optional[Route] = came_from[curr_stop]
14
15        if curr_stop == end:
16            break
17

```

```

18     for route in graph.departures[curr_stop]:
19         end_stop: Stop = route.end_stop
20         route_cost = curr_cost + minutes_cost(
21             prev_route, route, int(departure_min + curr_cost)
22         )
23
24         if end_stop not in costs or route_cost < costs[end_stop]:
25             costs[end_stop] = route_cost
26             came_from[end_stop] = route
27             pq.put((route_cost, end_stop))
28
29     return SearchResult(costs, came_from, end, visited_stops_counter)

```

Dodatkowo, algorytm jest ozdobiony dekoratorem, który zajmuje się przeanalizowaniem oraz wypisaniem informacji o jego przebiegu. Algorytm zwraca strukturę `SearchResult`, która zawiera informacje o kosztach, poprzednikach oraz liczbę odwiedzonych przystanków.

- `SearchResult`

```

1 class SearchResult(NamedTuple):
2     costs: dict[Stop, float]
3     came_from: dict[Stop, Optional[Route]]
4     end_stop: Stop
5     visited_stops: int

```

- `minutes_cost`

```

1 def minutes_cost(
2     prev_route: Optional[Route], curr_route: Route, curr_minutes: int
3 ) -> int:
4     MINUTES_IN_DAY: int = 24 * 60
5     delay = 0
6
7     if prev_route is None or prev_route.line == curr_route.line:
8         if curr_minutes > curr_route.departure_min:
9             delay = MINUTES_IN_DAY
10    elif prev_route.line != curr_route.line: # there is a line change
11        # modify this to take more time for change
12        if curr_minutes > curr_route.departure_min:
13            delay = MINUTES_IN_DAY
14    return curr_route.arrival_min + delay - curr_minutes

```

Funkcja oblicza czas przejazdu pomiędzy dwoma przystankami. Jeśli jest to przesiadka, to wymagany może być dodatkowy czas na zmianę linii.

2.1.3 Wnioski

Algorytm Dijkstry biorący pod uwagę czas podróży działa poprawnie. W tym wypadku załączalność algorytmu nie powinna stanowić problemu, ponieważ optymalizacja czasu dojazdu lokalnie jest jednoznaczna, i nie powinna uniemożliwić dotarcia do celu optymalną trasą (zawsze możemy przesiąść się do optymalnej linii).

2.2 A* - kryterium czasu

2.2.1 Opis

Algorytm ten jest rozszerzeniem algorytmu Dijkstry, który w celu szybszego znalezienia ścieżki. Różni się on jedynie uwzględnieniem w priorytecie kolejki dodatkowej heurystyki, która jest oszacowaniem odległości od danego wierzchołka do celu. Dzięki temu, najpierw analizowane są wierzchołki, które mają najmniejszą wartość heurystyki oraz funkcji celu. Heurystyka to zazwyczaj odległość mierzona pomiędzy dwoma punktami. W tym przypadku, heurystyka jest

wstrzyknięta do algorytmu, co pozwala na łatwe zmienianie jej wartości. Używane przeze mnie heurystyki to odległość manhattan(pionowa + pozioma) oraz odległość haversine (odległość geograficzna w km).

2.2.2 Działanie / kod

Algorytm ten tak naprawdę różni się od Dijkstry jedynie w sposobie obliczania priorytetu w kolejce. W tym wypadku, priorytetem jest suma kosztu oraz heurystyki.

- A*

```
1 def astar_time(  
2     graph: Graph,  
3     start: Stop,  
4     end: Stop,  
5     departure_min: int,  
6     heuristic: Callable[[Stop, Stop], float],  
7     heuristic_weight: float,  
8 ) -> SearchResult:  
9     costs: dict[Stop, float] = {start: 0}  
10    came_from: dict[Stop, Optional[Route]] = {start: None}  
11  
12    pq: PriorityQueue[tuple[float, Stop]] = PriorityQueue()  
13    pq.put((0, start))  
14  
15    visited_stops_counter: int = 0  
16    while not pq.empty():  
17        visited_stops_counter += 1  
18        _, curr_stop = pq.get()  
19        curr_cost = costs[curr_stop]  
20        prev_route: Optional[Route] = came_from[curr_stop]  
21  
22        if curr_stop == end:  
23            break  
24  
25        for route in graph.departures[curr_stop]:  
26            end_stop: Stop = route.end_stop  
27            route_cost = curr_cost + minutes_cost(  
28                prev_route, route, int(departure_min + curr_cost)  
29            )  
30  
31            if end_stop not in costs or route_cost < costs[end_stop]:  
32                costs[end_stop] = route_cost  
33                came_from[end_stop] = route  
34  
35                priority = route_cost + heuristic(end_stop, end) * heuristic_weight  
36                pq.put((priority, end_stop))  
37  
38    return SearchResult(costs, came_from, end, visited_stops_counter)
```

- Heurystyki

```
1 def manhattan_distance(start: Stop, end: Stop) -> float:  
2     # usually something around 0.1 so scale it to about 1  
3     DISTANCE_SCALE = 10  
4  
5     distance = abs(end.lat - start.lat) + abs(end.lon - start.lon)  
6  
7     return distance * DISTANCE_SCALE  
8  
9  
10 def haversine_distance(stop1: Stop, stop2: Stop) -> float:  
11     # usually around 5, so scale it down to about 1  
12     DISTANCE_SCALE = 1 / 5  
13  
14     lat1_rad = math.radians(stop1.lat)  
15     lon1_rad = math.radians(stop1.lon)  
16     lat2_rad = math.radians(stop2.lat)  
17     lon2_rad = math.radians(stop2.lon)
```

```

18
19     dlat = lat2_rad - lat1_rad
20     dlon = lon2_rad - lon1_rad
21
22     a = (
23         math.sin(dlat / 2) ** 2
24         + math.cos(lat1_rad) * math.cos(lat2_rad) * math.sin(dlon / 2) ** 2
25     )
26
27     c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
28     radius_of_earth_km = 6371.0
29     distance_km = radius_of_earth_km * c
30
31     return distance_km * DISTANCE_SCALE

```

Heurestyki zostały skalowane w celu łatwiejszego wyważania ich wpływu na algorytm. Celem było uzyskanie wartości w okolicach 1, co pozwala na łatwe zmienianie wpływu heurystyki w dalszym działaniu.

- Funkcja fabrykująca instancje algorytmu A* - ponieważ sygnatura funkcji jest inna niż algorytmu dijkstry, stworzono funkcję, która zwraca częściowo zaaplikowaną funkcję, w celu ujednolicenia ich sygnatur, co pozwoli na łatwe przekazywanie ich do dekoratora i wywołanie.

```

1 def create_astar(
2     heuristic: Callable[[Stop, Stop], float], mode: str, print_result: bool = True
3 ) -> Callable[[Graph, Stop, Stop, int], SearchResult]:
4
5     if mode == "t":
6         heuristic_weight: float = 30
7     elif mode == "p":
8         heuristic_weight: float = 0.5
9
10    else:
11        raise ValueError(f"Invalid mode for creating astar: {mode}")
12
13    def partially_applied_astar_time(
14        graph: Graph, start: Stop, end: Stop, departure_min: int
15    ) -> SearchResult:
16        return astar_time(graph, start, end, departure_min, heuristic, heuristic_weight)
17
18    def partially_applied_astar_change(
19        graph: Graph, start: Stop, end: Stop, departure_min: int
20    ) -> SearchResult:
21        return astar_change(
22            graph, start, end, departure_min, heuristic, heuristic_weight
23        )
24
25    astar = (
26        partially_applied_astar_time if mode == "t" else partially_applied_astar_change
27    )
28    return route_info_decorator(astar) if print_result else astar

```

2.2.3 Wnioski

Algorytm A*, po odpowiednim dobraniu heurystyki, działa poprawnie. W moim przypadku, heurystyka haversine sprawdzała się lepiej niż manhattan, co pozwalało na szybsze znalezienie optymalnej trasy. Zredukowana liczba odwiedzanych przystanków poprzez zmianę kolejności pozwalała na szybsze znalezienie trasy, która była, najprawdopodobniej optymalna. Przykładowe wyniki zostaną przedstawione na końcu zadania.

2.2.4 Napotkane problemy

Jeśli chodzi o napotkane problemy, to głównym problemem, było dobranie oraz skalowanie heurystyki takiej, która pozwoli na przyspieszenie działania algorytmu, jednak nie będzie ona przesadowywała kosztu, co może prowadzić do nieoptymalnego rozwiązania. W moim przypadku, heurystyka haversine sprawdzała się lepiej niż manhattan, jednak obie potrafiły przyspieszyć przeszukiwanie grafu względem algorytmu Dijkstry.

2.3 A* - kryterium przesiadek

2.3.1 Opis

Algorytm A* z kryterium przesiadek, różni się od algorytmu A* z kryterium czasu jedynie w sposobie obliczania funkcji kosztu. W tym wypadku, koszt to liczba przesiadek, które musimy wykonać aby dotrzeć do celu. Ponieważ algorytm ten ewaluuje tylko aktualną sytuację w bardzo zachłanny sposób (albo jest przesiadka, albo jej nie ma), to nie potrafi przewidzieć przyszłych przesiadek, co prowadzi do niezadowolających wyników.

2.3.2 Działanie / kod

Algorytm ten, zamiast brać pod uwagę czas, jako funkcję kosztu, ewaluuje czy jest dla danej trasy wymagana przesiadka czy jej nie ma.

- Różnica względem algorytmu A* z kryterium czasu

```
1 route_cost = curr_cost + (  
2     prev_route is not None and prev_route.line != route.line  
3 )
```

Jest to jedyne miejsce, w którym algorytmy się różnią. W tym wypadku, jeśli linia poprzedniej trasy jest różna od linii obecnej trasy, to dodawane jest 1 do kosztu. Pozostała część kodu jest identyczna. Również instancje tej funkcji dostarczane są przez funkcję fabrykującą, która pozwala na łatwe przekazywanie jej do dekoratora oraz wywołanie.

2.3.3 Wnioski

Algorytm ten jest bardzo zachłanny, i nie potrafi przewidzieć przyszłych przesiadek. Znajduje on trasy, które są *bardzo* nieoptymalne pod względem czasu podróży oraz liczby przesiadek. Brak znajomości kontekstu globalnego oraz potencjalnych przesiadek w przyszłości prowadzi do nieoptymalnych wyników.

2.4 Modyfikacja algorytmu A*

2.4.1 Opis

Modyfikacja została dodana do algorytmu A* z kryterium przesiadek, ponieważ to jego działanie było w ewidentny sposób nieoptymalne. W celu poprawienia wyników, kolejność przeglądania tras została zmieniona poprzez sortowanie odjazdów z danego przystanku na podstawie dwóch kryteriów. Pierwsze z nich to możliwość dojechania do przystanku docelowego. Za pomocą funkcji Graph.is_direct_connection, sprawdzane jest czy z danego przystanku można dojechać bezpośrednio do celu. Następnie dodawany jest także koszt minutowy dojazdu, jednak jego waga jest *znacznie* mniejsza niż możliwość bezpośredniego dojazdu.

2.4.2 Kod

```
1 # sort all departing routes in order to check the ones directly connected with end stop first
2 # also take into account the arrival time
3 def sort_key(route: Route) -> int:
4     # Priorityize direct connection over minutes
5     DIRECT_CONNECTION_WEIGHT = 10000
6     minutes_to_arrive: int = minutes_cost(
7         prev_route,
8         route,
9         (prev_route.arrival_min if prev_route is not None else departure_min),
10    )
11    no_direct_connection: bool = not graph.is_direct_connection(route, end)
12
13    return no_direct_connection * DIRECT_CONNECTION_WEIGHT + minutes_to_arrive
14
15 for route in sorted(
16     graph.departures[curr_stop], key=lambda route: sort_key(route)
17 ):
18     ## rest of A* code with the same logic
```

Przedstawiona jest tylko różnica względem naiwnego algorytmu A* z kryterium przesiadek.

2.4.3 Wnioski

Optymalizacja znacznie poprawiła wyniki algorytmu. Jeśli znalezione zostanie bezpośrednie połączenie, to zostanie ono wybrane na każdym kroku działania. Dodatkowo, sortowanie także wg czasu odjazdu pozwala także na wzięcie pod uwagę czasu podróży.

2.5 Przykładowe wyniki

2.5.1 Kryterium czasu

- Trasa BISKUPIN -> DWORZEC GŁÓWNY o 11:00

```
[ INFO ] results.py: Running dijkstra, from BISKUPIN to DWORZEC GŁÓWNY
2      BISKUPIN                11:01:00 → Chełmońskiego                11:04:00
143    Chełmońskiego          11:05:00 → Międzyrzeczka                11:08:00
120    Międzyrzeczka          11:10:00 → Na Niskich Łąkach            11:13:00
114    Na Niskich Łąkach       11:13:00 → Wzgórze Partyzantów          11:17:00
5      Wzgórze Partyzantów    11:17:00 → DWORZEC GŁÓWNY          11:19:00

Cost function value: 19
Visited 85 stops before finishing
Search time: 0.1486s

[ INFO ] results.py: Running Astar - prioritize time, from BISKUPIN to DWORZEC GŁÓWNY
2      BISKUPIN                11:01:00 → Chełmońskiego                11:04:00
143    Chełmońskiego          11:05:00 → Międzyrzeczka                11:08:00
120    Międzyrzeczka          11:10:00 → Na Niskich Łąkach            11:13:00
114    Na Niskich Łąkach       11:13:00 → Wzgórze Partyzantów          11:17:00
5      Wzgórze Partyzantów    11:17:00 → DWORZEC GŁÓWNY          11:19:00

Cost function value: 19
Visited 25 stops before finishing
Search time: 0.0433s
```

Jak widać, Dijkstra oraz A* znalazły tą samą trasę, która wymaga wielu przesiadek, jednak pozwala dotrzeć na miejsce w krótkim czasie. Czas wykonania A* był prawie 4 razy mniejszy niż Dijkstra.

- Trasa KRZYKI -> Bielany Wrocławskie - Kościół o 13:15

```
[ INFO ] results.py: Running dijkstra, from KRZYKI to Bielany Wrocławskie - Kościół
17      KRZYKI                13:15:00 → Przyjaźni                13:18:00
113     Przyjaźni             13:30:00 → Partynice (tor wyścigów konnych)13:33:00
612     Partynice (tor wyścigów konnych)14:10:00 → Bielany Wrocławskie - Kościół 14:15:00

Cost function value: 60
Visited 1454 stops before finishing
Search time: 1.7242s

[ INFO ] results.py: Running Astar - prioritize time, from KRZYKI to Bielany Wrocławskie - Kościół
17      KRZYKI                13:15:00 → Przyjaźni                13:18:00
113     Przyjaźni             13:30:00 → Partynice (tor wyścigów konnych)13:33:00
612     Partynice (tor wyścigów konnych)14:10:00 → Bielany Wrocławskie - Kościół 14:15:00

Cost function value: 60
Visited 526 stops before finishing
Search time: 0.6747s
```

Ponownie, oba algorytmy znalazły tą samą trasę. Znalezienie trasy zajęło A* około 3 razy mniej czasu.

- Trasa KMINKOWE -> DWORZEC AUTOBUSOWY o 00:00

```
[ INFO ] results.py: Running dijkstra, from KMINKOWA to DWORZEC AUTOBUSOWY
244     KMINKOWA              00:01:00 → DWORZEC AUTOBUSOWY          00:36:00

Cost function value: 36
Visited 85 stops before finishing
Search time: 0.1510s

[ INFO ] results.py: Running Astar - prioritize time, from KMINKOWA to DWORZEC AUTOBUSOWY
244     KMINKOWA              00:01:00 → DWORZEC AUTOBUSOWY          00:36:00

Cost function value: 36
Visited 44 stops before finishing
Search time: 0.0874s
```

Ponownie, oba algorytmy znalazły tą samą trasę. Znalezienie trasy zajęło A* około 3 razy mniej czasu.

2.5.2 Kryterium przesiadek

- Trasa BISKUPIN -> DWORZEC GŁÓWNY o 11:00

```
[ INFO ] results.py: Running Astar - prioritize changes improved, from BISKUPIN to DWORZEC GŁÓWNY
2      BISKUPIN                11:01:00 → DWORZEC GŁÓWNY                11:25:00

Cost function value: 0
Visited 15 stops before finishing
Search time: 0.0627s

[ INFO ] results.py: Running Astar - prioritize changes naive, from BISKUPIN to DWORZEC GŁÓWNY
1      BISKUPIN                16:57:00 → PL. GRUNWALDZKI                17:08:00
D      PL. GRUNWALDZKI         23:01:00 → GALERIA DOMINIKAŃSKA         23:06:00
K      GALERIA DOMINIKAŃSKA    06:28:00 → DWORZEC GŁÓWNY                06:32:00

Cost function value: 2
Visited 169 stops before finishing
Search time: 0.2005s

Press enter to continue
```

Różnica pomiędzy naiwną wersją a zoptymalizowaną jest ogromna. Zoptymalizowana wersja znalazła trasę bez przesiadek. Naiwna implementacja wymagała aż 2 przesiadek, a dodatkowo nie bierze pod uwagę czasu podróży.

- Trasa KRZYKI -> Bielany Wrocławskie - Kościół o 13:15

```
[ INFO ] results.py: Running Astar - prioritize changes improved, from KRZYKI to Bielany Wrocławskie - Kościół
602 KRZYKI 13:49:00 → Partynice (tor wyścigów konnych)13:53:00
612 Partynice (tor wyścigów konnych)14:10:00 → Bielany Wrocławskie - Kościół 14:15:00

Cost function value: 1
Visited 47 stops before finishing
Search time: 0.1121s

[ INFO ] results.py: Running Astar - prioritize changes naive, from KRZYKI to Bielany Wrocławskie - Kościół
602 KRZYKI 22:02:00 → Partynice (tor wyścigów konnych)22:06:00
612 Partynice (tor wyścigów konnych)06:17:00 → Bielany Wrocławskie - Kościół 06:22:00

Cost function value: 1
Visited 83 stops before finishing
Search time: 0.0806s
```

W tym wypadku oba algorytmy wymagają 1 przesiadki, jednak zoptymalizowana wersja pozwala na szybsze dotarcie do celu.

- Trasa Katedra -> Zaolziańska o 22:00

```
[ INFO ] results.py: Running Astar - prioritize changes improved, from Katedra to Zaolziańska
2 Katedra 22:06:00 → Zaolziańska 22:19:00

Cost function value: 0
Visited 7 stops before finishing
Search time: 0.0435s

[ INFO ] results.py: Running Astar - prioritize changes naive, from Katedra to Zaolziańska
A Katedra 21:23:00 → Arkady (Capitol) 21:32:00
2 Arkady (Capitol) 10:37:00 → Zaolziańska 10:39:00

Cost function value: 1
Visited 64 stops before finishing
Search time: 0.0714s
```

Można także zaobserwować, że pomimo bardziej skomplikowanych obliczeń (sortowanie tras), ulepszona wersja często działa szybciej niż naiwna (odwiedza mniej przystanków przed znalezieniem trasy).

3 Zadanie 2

3.1 Algorytm Tabu Search, bez ograniczenia tablicy T

3.1.1 Opis

Algorytm działa poprzez przeszukiwanie przestrzeni rozwiązań w celu znalezienia najlepszego rozwiązania. Rozwiązaniem jest tutaj permutacja kolejności przystanków, która pozwoli zminimalizować czas podróży lub liczbę przesiadek. Generowane jest rozwiązanie początkowe (jest to ewaluacja kolejności przystanków otrzymanej w parametrze), a następnie przeszukiwana jest przestrzeń rozwiązań sąsiednich, poprzez generowanie sąsiedztwa za pomocą zamiany dwóch przystanków. Rozwiązania sąsiednie są oceniane, a następnie wybierane jest najlepsze z nich, które nie należy do listy tabu, a następnie dodawane do listy tabu. Cały cykl powtarza się aż do spełnienia warunku stopu, który w tym wypadku jest liczbą iteracji. W celu wyznaczenia trasy pomiędzy poszczególnymi przystankami używany jest algorytm A*, używający jako heurystyki odległość haversine, ponieważ spisywał się on najlepiej.

3.1.2 Kod

- Główny algorytm

```
1 def tabu_search(  
2     graph: Graph,  
3     start: Stop,  
4     to_visit: list[Stop],  
5     departure_min: int,  
6     search_function: Callable,  
7     max_iterations: int,  
8 ) -> TabuSearchResult:  
9     best_solution: TabuSearchResult = create_path_between_stops(  
10         graph, [start] + to_visit + [start], departure_min, search_function  
11     )  
12     current_solution = best_solution  
13  
14     tabu_list: list[TabuSearchResult] = []  
15  
16     for _ in range(max_iterations):  
17         neighbors = get_neighbors(current_solution.to_visit)  
18         best_neighbor_solution = None  
19         best_cost = float("inf")  
20  
21         for neighbor in neighbors:  
22             if neighbor not in tabu_list:  
23                 neighbor_solution = create_path_between_stops(  
24                     graph, neighbor, departure_min, search_function  
25                 )  
26  
27                 if neighbor_solution.total_cost < best_cost:  
28                     best_neighbor_solution = neighbor_solution  
29                     best_cost = neighbor_solution.total_cost  
30  
31             if best_neighbor_solution is None:  
32                 break  
33  
34         current_solution = best_neighbor_solution  
35         tabu_list.append(best_neighbor_solution)  
36  
37         if best_neighbor_solution.total_cost < best_solution.total_cost:  
38             best_solution = best_neighbor_solution  
39  
40     return best_solution
```

- Funkcja generująca sąsiedztwo

```
1 def get_neighbors(current_solution: list[Stop]) -> list[list[Stop]]:  
2     n = len(current_solution)  
3     neighbors = []  
4  
5     for i in range(1, n - 1):  
6         for j in range(i + 1, min(i + 3, n - 1)):  
7             neighbor = current_solution[:]  
8             neighbor[i], neighbor[j] = neighbor[j], neighbor[i]  
9             neighbors.append(neighbor)  
10  
11     return neighbors
```

Funkcja generująca sąsiedztwo poprzez zamianę dwóch przystanków. Funkcja omija pierwszy i ostatni przystanek, ponieważ jest to przystanek początkowy. W celu ograniczenia liczby generowanych sąsiadów, wewnętrzna pętla jest ograniczona do 3 przystanków od aktualnego przystanku.

- Funkcja tworząca ścieżkę pomiędzy przystankami

```
1 def create_path_between_stops(  
2     graph, to_visit: list[Stop], departure_min: int, search_function: Callable  
3 ) -> TabuSearchResult:  
4     came_from: list[SearchResult] = []
```

```

5     total_cost: float = 0
6     current_min: int = departure_min
7
8     for i in range(len(to_visit) - 1):
9         start_stop: Stop = to_visit[i]
10        end_stop: Stop = to_visit[i + 1]
11        result: SearchResult = search_function(graph, start_stop, end_stop, current_min)
12        arrival_min: int = result.came_from[end_stop].arrival_min
13        total_cost += result.costs[end_stop]
14        came_from.append(result)
15        current_min = arrival_min
16
17    return TabuSearchResult(came_from, to_visit, total_cost)

```

Funkcja ta tworzy ścieżkę pomiędzy przystankami, używając podanego algorytmu. Zwraca ona strukturę TabuSearchResult, która zawiera informacje o kosztach, poprzednikach, przyjętej kolejności przystanków oraz całkowitym koszcie podróży.

3.1.3 Przykładowe działanie

- Trasa BISKUPIN -> ['DWORZEC GŁÓWNY', 'Ogród Botaniczny', 'Dubois', 'PL. GRUNWALDZKI'] -> BISKUPIN 0 8:00

```

[ INFO ] create_graph.py: Found serialized graph: /home/sejsmo/repos/ai-course/list1/data/serialized_graph.pickle
[ INFO ] create_graph.py: Loading serialized graph

[ INFO ] results.py: Running Tabu without constraints - prioritize time, from BISKUPIN between ['DWORZEC GŁÓWNY', 'Ogród Botaniczny', 'Dubois', 'PL. GRUNWALDZKI']

Found solution is to visit the stops in order:
BISKUPIN -> PL. GRUNWALDZKI -> Ogród Botaniczny -> Dubois -> DWORZEC GŁÓWNY -> BISKUPIN
Path to PL. GRUNWALDZKI
2 BISKUPIN 08:02:00 -> PL. GRUNWALDZKI 08:13:00

Path to Ogród Botaniczny
19 PL. GRUNWALDZKI 08:14:00 -> Ogród Botaniczny 08:19:00

Path to Dubois
19 Ogród Botaniczny 08:19:00 -> Dubois 08:23:00

Path to DWORZEC GŁÓWNY
128 Dubois 08:25:00 -> pl. Bema 08:28:00
23 pl. Bema 08:28:00 -> GALERIA DOMINIKAŃSKA 08:33:00
110 GALERIA DOMINIKAŃSKA 08:33:00 -> DWORZEC GŁÓWNY 08:36:00

Path to BISKUPIN
2 DWORZEC GŁÓWNY 08:36:00 -> GALERIA DOMINIKAŃSKA 08:41:00
0 GALERIA DOMINIKAŃSKA 08:43:00 -> PL. GRUNWALDZKI 08:49:00
19 PL. GRUNWALDZKI 08:49:00 -> ZOO 08:54:00
2 ZOO 08:55:00 -> BISKUPIN 09:01:00

Cost function value: 61
Search time: 8.7247s

```

Jak widać dla 4 przysstanków czas przeszukiwania wyniosło około 8s. Otrzymany wynik wydaje się prawidłowy - przystanki są odwiedzane w kolejności, która pozwala na zminimalizowanie czasu podróży.

- Trasa GALERIA DOMINIKAŃSKA -> ['Paprotna', 'Wyszyńskiego', 'KMINKOWA'] -> GALERIA DOMINIKAŃSKA

```

Found solution is to visit the stops in order:
GALERIA DOMINIKAŃSKA -> Paprotna -> KMINKOWA -> Wyszyńskiego -> GALERIA DOMINIKAŃSKA
Path to Paprotna
10 GALERIA DOMINIKAŃSKA 16:00:00 -> Rynek 16:05:00
142 Rynek 16:07:00 -> Paprotna 16:24:00

Path to KMINKOWA
111 Paprotna 16:28:00 -> KMINKOWA 16:40:00

Path to Wyszyńskiego
111 KMINKOWA 16:43:00 -> Nowowiejska 17:11:00
1 Nowowiejska 17:14:00 -> Wyszyńskiego 17:16:00

Path to GALERIA DOMINIKAŃSKA
A Wyszyńskiego 17:17:00 -> Urząd Wojewódzki (Muzeum Narodowe) 17:22:00
914 Urząd Wojewódzki (Muzeum Narodowe) 17:23:00 -> GALERIA DOMINIKAŃSKA 17:26:00

Cost function value: 86
Search time: 9.2779s

```

3.2 Dobór długości listy T

3.2.1 Opis

W tym kroku, dodano do algorytmu parametr, który pozwala na ograniczenie długości listy tabu. Rozwiązanie takie, pomaga zredukować użycie pamięci oraz uniknąć zatrzymania w pewnym obszarze poszukiwań. Usunięcie starszych elementów z listy daje możliwość ponownego przeszukania obszaru, który był wcześniej na liście tabu.

3.2.2 Kod

```
1 # def tabu_search(  
2 #     graph: Graph,  
3 #     start: Stop,  
4 #     to_visit: list[Stop],  
5 #     departure_min: int,  
6 #     search_function: Callable,  
7 #     max_iterations: int,  
8 #     tabu_list_size: Optional[int] = None,  
9 # ) -> TabuSearchResult:  
10 #     best_solution: TabuSearchResult = create_path_between_stops(  
11 #         graph, [start] + to_visit + [start], departure_min, search_function  
12 #     )  
13 #     current_solution = best_solution  
14 #  
15 #     tabu_list: list[TabuSearchResult] = []  
16 #  
17 #     for _ in range(max_iterations):  
18 #         neighbors = get_neighbors(current_solution.to_visit)  
19 #         best_neighbor_solution = None  
20 #         best_cost = float("inf")  
21 #  
22 #         for neighbor in neighbors:  
23 #             if neighbor not in tabu_list:  
24 #                 neighbor_solution = create_path_between_stops(  
25 #                     graph, neighbor, departure_min, search_function  
26 #                 )  
27 #  
28 #                 if neighbor_solution.total_cost < best_cost:  
29 #                     best_neighbor_solution = neighbor_solution  
30 #                     best_cost = neighbor_solution.total_cost  
31 #  
32 #             if best_neighbor_solution is None:  
33 #                 break  
34 #  
35 #             current_solution = best_neighbor_solution  
36 #             tabu_list.append(best_neighbor_solution)  
37 #             if tabu_list_size and len(tabu_list) > tabu_list_size:  
38 #                 tabu_list.pop(0)  
39 #  
40 #             if best_neighbor_solution.total_cost < best_solution.total_cost:  
41 #                 best_solution = best_neighbor_solution  
42 #  
43 #     return best_solution
```

Dodano parametr `tabu_list_size`, który pozwala na ograniczenie długości listy tabu. W przypadku przekroczenia tej długości, usuwany jest najstarszy element z listy (czyli element o indeksie 0, ponieważ dodawanie następuje na koniec listy).