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

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 połączniu 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 Wstepna 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 wieszy 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 odjazdju 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.).

```python
class Stop(NamedTuple):
    name: str
    lat: float
    lon: float

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Stop):
            return False
        return self.name == other.name

    def __hash__(self) -> int:
        return hash(self.name)

    def __repr__(self) -> str:
        return f"Stop: {self.name}"

```

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

```python
class Route(NamedTuple):
    line: str
    start_stop: Stop
    end_stop: Stop
    departure_min: int
    arrival_min: int
    start_stop_lat: float
    start_stop_lon: float
    end_stop_lat: float
    end_stop_lon: float

    def __eq__(self, other) -> bool:
        if not isinstance(other, Route):
            return False
        return self.line == other.line and self.end_stop == other.end_stop

    def __hash__(self) -> int:
        return hash((self.line, self.end_stop.name))

    def __repr__(self) -> str:
        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 przseszukiwania pod kątem ilości przesiadek.

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

```
    def add_route(self, route: Route):
        if route.start_stop not in self.departures:
            self.departures[route.start_stop] = []
            self.arriving_line_names[route.start_stop] = set()

        if route.end_stop not in self.departures:
            self.departures[route.end_stop] = []
            self.arriving_line_names[route.end_stop] = set()

        self.departures[route.start_stop].append(route)
        self.arriving_line_names[route.end_stop].add(route.line)

    def get_stop(self, name: str) -> Optional[Stop]:
        for stop in self.departures:
            if stop.name == name:
                return stop
        return None

    def is_direct_connection(self, route: Route, destination: Stop) -> bool:
        return route.line in self.arriving_line_names[destination]
```

2 Zadanie 1

2.1 a) 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

```
@route_info_decorator
def dijkstra(graph: Graph, start: Stop, end: Stop, departure_min: int) -> SearchResult:
    costs: dict[Stop, float] = {start: 0}
    came_from: dict[Stop, Optional[Route]] = {start: None}

    pq: PriorityQueue[tuple[float, Stop]] = PriorityQueue()
    pq.put((0, start))

    visited_stops_counter: int = 0
    while not pq.empty():
        visited_stops_counter += 1
        curr_cost, curr_stop = pq.get()
        prev_route: Optional[Route] = came_from[curr_stop]

        if curr_stop == end:
            break

```

```
        for route in graph.departures[curr_stop]:
            end_stop: Stop = route.end_stop
            route_cost = curr_cost + minutes_cost(
                prev_route, route, int(departure_min + curr_cost)
            )

            if end_stop not in costs or route_cost < costs[end_stop]:
                costs[end_stop] = route_cost
                came_from[end_stop] = route
                pq.put((route_cost, end_stop))

    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 informcje o kosztach, poprzednikach oraz liczbę odwiedzonych przystanków.

- SearchResult

```
class SearchResult(NamedTuple):
    costs: dict[Stop, float]
    came_from: dict[Stop, Optional[Route]]
    end_stop: Stop
    visited_stops: int
```

- minutes_cost

```
def minutes_cost(
    prev_route: Optional[Route], curr_route: Route, curr_minutes: int
) -> int:
    MINUTES_IN_DAY: int = 24 * 60
    delay = 0

    if prev_route is None or prev_route.line == curr_route.line:
        if curr_minutes > curr_route.departure_min:
            delay = MINUTES_IN_DAY
    elif prev_route.line != curr_route.line:  # there is a line change
        # modify this to take more time for change
        if curr_minutes > curr_route.departure_min:
            delay = MINUTES_IN_DAY
    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 zachłanność algorytmu nie powinna stanowić problemu, poniweaż optymalizacja czasu dojazdu lokalnie jest jednoznaczna, i nie powinna uniemożliwić dotarcia do celu optymalną trasą (zawsze możemy przesiąść się do optymalnej linii).

2.2 b) A* - kryterium czasu

2.2.1 Opis

Algorytm ten jest rozszszeniem 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 heurestyki oraz funkcji celu. Heurestyka to zazwyczaj odległość mierzona pomiędzy dwoma punktami. W tym przypadku, heurestyka jest

wstrzyknięta do algorytmu, co pozwala na łatwe zmienianie jej wartości. Używane przeze mnie heurestyki 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 heurestyki.

- A*

```python
def astar_time(
    graph: Graph,
    start: Stop,
    end: Stop,
    departure_min: int,
    heuristic: Callable[[Stop, Stop], float],
    heuristic_weight: float,
) -> SearchResult:
    costs: dict[Stop, float] = {start: 0}
    came_from: dict[Stop, Optional[Route]] = {start: None}

    pq: PriorityQueue[tuple[float, Stop]] = PriorityQueue()
    pq.put((0, start))

    visited_stops_counter: int = 0
    while not pq.empty():
        visited_stops_counter += 1
        _, curr_stop = pq.get()
        curr_cost = costs[curr_stop]
        prev_route: Optional[Route] = came_from[curr_stop]

        if curr_stop == end:
            break

        for route in graph.departures[curr_stop]:
            end_stop: Stop = route.end_stop
            route_cost = curr_cost + minutes_cost(
                prev_route, route, int(departure_min + curr_cost)
            )

            if end_stop not in costs or route_cost < costs[end_stop]:
                costs[end_stop] = route_cost
                came_from[end_stop] = route

                priority = route_cost + heuristic(end_stop, end) * heuristic_weight
                pq.put((priority, end_stop))

    return SearchResult(costs, came_from, end, visited_stops_counter)
```

- Heurystyki

```python
def manhattan_distance(start: Stop, end: Stop) -> float:
    # usually something around 0.1 so scale it to about 1
    DISTANCE_SCALE = 10

    distance = abs(end.lat - start.lat) + abs(end.lon - start.lon)

    return distance * DISTANCE_SCALE


def haversine_distance(stop1: Stop, stop2: Stop) -> float:
    # usually around 5, so scale it down to about 1
    DISTANCE_SCALE = 1 / 5

    lat1_rad = math.radians(stop1.lat)
    lon1_rad = math.radians(stop1.lon)
    lat2_rad = math.radians(stop2.lat)
    lon2_rad = math.radians(stop2.lon)
```

```
    dlat = lat2_rad - lat1_rad
    dlon = lon2_rad - lon1_rad

    a = (
        math.sin(dlat / 2) ** 2
        + math.cos(lat1_rad) * math.cos(lat2_rad) * math.sin(dlon / 2) ** 2
    )

    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    radius_of_earth_km = 6371.0
    distance_km = radius_of_earth_km * c

    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 heurestyki 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.

```
def create_astar(
    heuristic: Callable[[Stop, Stop], float], mode: str, print_result: bool = True
) -> Callable[[Graph, Stop, Stop, int], SearchResult]:

    if mode == "t":
        heuristic_weight: float = 30
    elif mode == "p":
        heuristic_weight: float = 0.5

    else:
        raise ValueError(f"Invalid mode for creating astar: {mode}")

    def partially_applied_astar_time(
        graph: Graph, start: Stop, end: Stop, departure_min: int
    ) -> SearchResult:
        return astar_time(graph, start, end, departure_min, heuristic, heuristic_weight)

    def partially_applied_astar_change(
        graph: Graph, start: Stop, end: Stop, departure_min: int
    ) -> SearchResult:
        return astar_change(
            graph, start, end, departure_min, heuristic, heuristic_weight
        )

    astar = (
        partially_applied_astar_time if mode == "t" else partially_applied_astar_change
    )
    return route_info_decorator(astar) if print_result else astar
```

2.2.3 Wnioski

Algorytm A*, po odpowiednim dobraniu heurestyki, działa poprawnie. W moim przypadku, heurestyka haversine sprawdzała się lepiej niż manhattan, co pozwalało na szybsze znalezienie optymalnej trasy. Zredukowana liczba odwiedznych przystanków poprzez zmianę kolejności pozwalała na szybsze znalezienie trasy, która była, najprawpodobniej 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 heurestyki takiej, która pozwoli na przyspieszenie działania algorytmu, jednak nie będzie ona przesacowywała kosztu, co może prowadzić do nieoptymalnego rozwiązania. W moim przypadku, heurestyka haversine sprawdzała się lepiej niż manhattan, jednak obie potrafiły przyspieszyć przeszukiwanie grafu względem algorytmu Dijkstry.

2.3 c) 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 niezadowalających 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

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

  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 d) Modyfikacja algorytmu A*

2.4.1 Opis

Modyfikacja została dodana do algorytmy 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 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

```python
# sort all departing routes in order to check the ones directly connected with end stop first
# also take into account the arrival time
def sort_key(route: Route) -> int:
    # Prioritiyize direct connetion over minutes
    DIRECT_CONNECTION_WEIGHT = 10000
    minutes_to_arrive: int = minutes_cost(
        prev_route,
        route,
        (prev_route.arrival_min if prev_route is not None else departure_min),
    )
    no_direct_connection: bool = not graph.is_direct_connection(route, end)

    return no_direct_connection * DIRECT_CONNECTION_WEIGHT + minutes_to_arrive

for route in sorted(
    graph.departures[curr_stop], key=lambda route: sort_key(route)
):
## 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 → Międzyrzecka               11:08:00
120       Międzyrzecka                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 → Międzyrzecka               11:08:00
120       Międzyrzecka                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 zaobserwoać, że pomimo bardziej skomplikowanych obliczeń (sortowanie tras), ulepszona wersja dzęsto 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 kolejnośći przystanków otrzymanej w parametrze), a następnie przeszukiwana jest przestrzeń rozwiązań sąsiednich, poprzez generowanie sąsiedztwa za pomocą zamiany dwóch przystanków. Rozwiązania sąsiedznie 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 heurestyki odległość haversine, ponieważ spisywał się on najlepiej.

3.1.2 Kod

- Główny algorytm

```python
def tabu_search(
    graph: Graph,
    start: Stop,
    to_visit: list[Stop],
    departure_min: int,
    search_function: Callable,
    max_iterations: int,
) -> TabuSearchResult:
    best_solution: TabuSearchResult = create_path_between_stops(
        graph, [start] + to_visit + [start], departure_min, search_function
    )
    current_solution = best_solution

    tabu_list: list[TabuSearchResult] = []

    for _ in range(max_iterations):
        neighbors = get_neighbors(current_solution.to_visit)
        best_neighbor_solution = None
        best_cost = float("inf")

        for neighbor in neighbors:
            if neighbor not in tabu_list:
                neighbor_solution = create_path_between_stops(
                    graph, neighbor, departure_min, search_function
                )

                if neighbor_solution.total_cost < best_cost:
                    best_neighbor_solution = neighbor_solution
                    best_cost = neighbor_solution.total_cost

        if best_neighbor_solution is None:
            break

        current_solution = best_neighbor_solution
        tabu_list.append(best_neighbor_solution)

        if best_neighbor_solution.total_cost < best_solution.total_cost:
            best_solution = best_neighbor_solution

    return best_solution
```

- Funkcja generująca sąsiedztwo

```python
def get_neighbors(current_solution: list[Stop]) -> list[list[Stop]]:
    n = len(current_solution)
    neighbors = []

    for i in range(1, n - 1):
        for j in range(i + 1, min(i + 3, n - 1)):
            neighbor = current_solution[:]
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(neighbor)

    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

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

```
    total_cost: float = 0
    current_min: int = departure_min

    for i in range(len(to_visit) - 1):
        start_stop: Stop = to_visit[i]
        end_stop: Stop = to_visit[i + 1]
        result: SearchResult = search_function(graph, start_stop, end_stop, current_min)
        arrival_min: int = result.came_from[end_stop].arrival_min
        total_cost += result.costs[end_stop]
        came_from.append(result)
        current_min = arrival_min

    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
D         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 wyniósło 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 tługości 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

```
# def tabu_search(
#     graph: Graph,
#     start: Stop,
#     to_visit: list[Stop],
#     departure_min: int,
#     search_function: Callable,
#     max_iterations: int,
    tabu_list_size: Optional[int] = None,
# ) -> TabuSearchResult:
#     best_solution: TabuSearchResult = create_path_between_stops(
#         graph, [start] + to_visit + [start], departure_min, search_function
#     )
#     current_solution = best_solution
#
#     tabu_list: list[TabuSearchResult] = []
#
#     for _ in range(max_iterations):
#         neighbors = get_neighbors(current_solution.to_visit)
#         best_neighbor_solution = None
#         best_cost = float("inf")
#
#         for neighbor in neighbors:
#             if neighbor not in tabu_list:
#                 neighbor_solution = create_path_between_stops(
#                     graph, neighbor, departure_min, search_function
#                 )
#
#                 if neighbor_solution.total_cost < best_cost:
#                     best_neighbor_solution = neighbor_solution
#                     best_cost = neighbor_solution.total_cost
#
#         if best_neighbor_solution is None:
#             break
#
#         current_solution = best_neighbor_solution
#         tabu_list.append(best_neighbor_solution)
        if tabu_list_size and len(tabu_list) > tabu_list_size:
            tabu_list.pop(0)
#
#         if best_neighbor_solution.total_cost < best_solution.total_cost:
#             best_solution = best_neighbor_solution
#
#     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).