Sztuczna inteligencja i inżynieria wiedzy

Lista 1

Bartosz Gotowski

2 kwietnia 2025

Spis treści

1 Wprowadzenie

1.1 Opis problemu

W niniejszym raporcie zajmę się dwoma zadaniami optymalizacyjnymi:

1. Wyszukiwanie najkrótszych połączeń między dwoma przystankami z uwzględnieniem kryteriów czasu przejazdu lub liczby przesiadek.

2. Planowanie optymalnej trasy przejazdu przez zbiór zadanych przystanków z powrotem do punktu początkowego (wariant problemu komiwojażera).

1.2 Dane wejściowe

Dane dotyczące systemu komunikacji miejskiej dostarczone są w pliku `connection_graph.csv`. Plik ten zawiera informacje o połączeniach między przystankami, czasach przejazdów oraz liniach komunikacyjnych.

2 Zadanie 1: Wyszukiwanie najkrótszych połączeń

2.1 Algorytm Dijkstry

2.1.1 Opis teoretyczny

Algorytm Dijkstry służy do znajdowania najkrótszych ścieżek w grafie ważonym o nieujemnych wagach krawędzi. W kontekście naszego zadania, graf reprezentuje sieć transportu publicznego, gdzie wierzchołkami są przystanki, a krawędzie oznaczają bezpośrednie połączenia między nimi.

2.1.2 Implementacja dla optymalizacji czasu przejazdu

Konkretne przystanki będą wierzchołkami, natomiast krawędzie to połączenia pomiędzy nimi. Musimy tutaj uwzględnić:

- Rozkłady jazdy – pojazdy odjeżdżają o konkretnych godzinach
- Czas oczekiwania na przystanku – jeśli przyjedziemy na przystanek przed odjazdem, musimy czekać
- Dzień cykliczny – rozkład jazdy obowiązuje w cyklu 24-godzinnym

Zaimplementowany algorytm wykorzystuje kolejkę priorytetową (zrealizowaną za pomocą modułu `heapq`) do wybierania przystanku o najmniejszym dotychczasowym czasie dojazdu. Główne elementy implementacji to:

1. Inicjalizacja tablicy odległości (w naszym przypadku – czasów przyjazdu) dla wszystkich przystanków wartością nieskończoną, z wyjątkiem przystanku startowego, któremu przypisujemy czas początkowy.

2. Inicjalizacja słownika `prev`, który przechowuje informacje o poprzednich przystankach i detalach podróży, umożliwiając późniejsze odtworzenie trasy.

3. W głównej pętli algorytmu:
    - Wybór przystanku o najmniejszym dotychczasowym czasie dojazdu z kolejki priorytetowej.
    - Jeśli wybrany przystanek jest celem, algorytm kończy działanie i zwraca znalezioną trasę.
    - W przeciwnym przypadku rozpatrujemy wszystkie połączenia wychodzące z bieżącego przystanku.
    - Dla każdego połączenia obliczamy czas przyjazdu uwzględniając: aktualny czas, czas oczekiwania na pojazd oraz czas przejazdu.
    - Jeśli znaleźliśmy lepszą trasę do sąsiedniego przystanku, aktualizujemy informacje i dodajemy ten przystanek do kolejki.

2.1.3 Wyniki i analiza

Algorytm Dijkstry został przetestowany na trzech różnych scenariuszach o rosnącym poziomie złożoności.

Scenariusz 1: Bliskie przystanki Test dla trasy Śliczna → Prudnicka, rozpoczynając o 16:25, wykazał, że algorytm radzi sobie z prostymi trasami. Znaleziona ścieżka obejmowała trzy przesiadki:

- Linia 112: Śliczna → Borowska (Aquapark), 16:25–16:27
- Linia 124: Borowska (Aquapark) → Kamienna, 16:28–16:31
- Linia 8: Kamienna → Prudnicka, 16:31–16:32

Całkowity czas podróży wyniósł 7 minut, a czas obliczeń wyniósł 0,0077 sekundy.

Scenariusz 2: Średnia odległość Dla trasy Śliczna → most Grunwaldzki, rozpoczynając o 8:50, algorytm znalazł trasę składającą się z trzech etapów przy czasie obliczeniowym 0,0598 sekundy:

- Linia 612: Śliczna → DWORZEC AUTOBUSOWY, 8:50–8:54
- Linia K: DWORZEC AUTOBUSOWY → DWORZEC GŁÓWNY, 8:54–8:57
- Linia 4: DWORZEC GŁÓWNY → most Grunwaldzki, 8:57–9:08

Całkowity czas podróży wyniósł 18 minut (8:50–9:08), a znaczący wzrost czasu obliczeń w porównaniu do pierwszego scenariusza odzwierciedla większą liczbę możliwych połączeń do rozważenia.

Scenariusz 3: Trasa złożona Najbardziej wymagający test (Śliczna → Bezpieczna, start o 16:25) wykazał dalszy wzrost wymaganej mocy obliczeniowej (0,1116 sekundy) dla tras wymagających wielu przesiadek:

- Linia 112: Śliczna → DWORZEC AUTOBUSOWY, 16:25–16:31
- Linia 15: DWORZEC AUTOBUSOWY → Arkady (Capitol), 16:31–16:35

- Linia 7: Arkady (Capitol) → Broniewskiego, 16:35–16:56
- Linia 118: Broniewskiego → Bezpieczna, 16:56–17:00

Całkowita podróż zajęła 35 minut (16:25–17:00), jednak czas obliczania wyniósł 14-krotnie więcej niż najłatwiejszy scenariusz.

Wnioski z analizy wydajności Z przeprowadzonych testów wynika kilka istotnych obserwacji:

1. **Skalowalność:** Wzrost czasu obliczeń jest proporcjonalny do złożoności problemu - mniej więcej liniowy względem liczby przeszukiwanych węzłów.
2. **Efektywność przesiadek:** Algorytm skutecznie minimalizuje czas oczekiwania na przesiadki, co widać we wszystkich scenariuszach. Jedynie w przypadku pierwszego scenariusza przesiadka z linii 112 na 124 zajęła 1 minutę, w pozostałych przypadkach czas oczekiwania był zerowy.

Przeprowadzone testy wskazują na poprawne działanie algorytmu. Jednocześnie warto zaznaczyć, że w praktycznym zastosowaniu należałoby uwzględnić dodatkowy czas na przesiadkę, gdyż zerowy czas przesiadki może być niemożliwy do zrealizowania przez człowieka.

2.2 Algorytm A* dla optymalizacji czasu przejazdu

2.2.1 Opis teoretyczny

Algorytm A* stanowi rozszerzenie algorytmu Dijkstry poprzez wprowadzenie heurystyki, która szacuje koszt dotarcia do celu. Tutaj heurystyka opierała się na odległości euklidesowej podzielonej przez średnią prędkość środków transportu.

2.2.2 Implementacja

Główne elementy implementacji obejmują:

1. **Funkcja heurystyczna** – Kluczowym elementem algorytmu A* jest funkcja heurystyczna. Zaimplementowana heurystyka opiera się na przybliżonej odległości geograficznej między przystankami:
   - Odległość obliczana jest przy pomocy wzoru pitagorejskiego.
   - Odległość przekształcana jest na szacowany czas podróży przy założeniu średniej prędkości transportu miejskiego wynoszącej 30 km/h.
   - Funkcja heurystyczna nigdy nie przeszacowuje rzeczywistego czasu potrzebnego na dotarcie do celu, co zapewnia optymalność znalezionej trasy.
2. **Implementacja** – Heurystyka opiera się na geograficznej odległości między przystankami:

```
    def heuristic(start_station, end_station, station_coordinates):

        start_lat, start_lon = station_coordinates[start_station]
        end_lat, end_lon = station_coordinates[end_station]
```

```
        distance = pythagorean_distance(start_lat, start_lon,
    end_lat, end_lon)

        avg_speed = 30 * (10 / 36)  # metry na sekunde

        estimated_time = distance / avg_speed

        return estimated_time

```

3. **Struktura kolejki priorytetowej** – Węzły w kolejce priorytetowej są reprezentowane jako krotki w formacie: `(czas_z_heurystyką, liczba_przesiadek, aktualny_czas, przystanek, aktualna_linia)`. Kluczowe jest to, że priorytet węzła obliczany jest jako suma dotychczasowego czasu podróży i szacowanego czasu pozostałego do celu.

4. **Obsługa czasu** – Analogicznie jak w algorytmie Dijkstry, implementacja A* uwzględnia:

   - Cykliczność rozkładu jazdy (24-godzinna) z odpowiednim dostosowaniem czasów.
   - Czas oczekiwania na pojazd, jeśli przybycie na przystanek nastąpiło przed planowanym odjazdem.
   - Rzeczywisty czas przejazdu między przystankami.

5. **Śledzenie przesiadek** – Algorytm śledzi aktualnie używaną linię komunikacyjną, co pozwala na:

   - Identyfikację momentów przesiadki (zmiana linii).
   - Budowanie pełnej trasy z informacją o używanych liniach, czasach odjazdu i przyjazdu.

Implementacja uwzględnia również obsługę przypadków specjalnych, takich jak:

- Brak danych geograficznych dla niektórych stacji (zastosowanie wartości domyślnej dla heurystyki).
- Nieistnienie stacji początkowej lub końcowej w sieci.
- Brak możliwych połączeń między stacjami (zwrócenie informacji o braku trasy).

W porównaniu do algorytmu Dijkstry, A* potencjalnie redukuje liczbę rozpatrywanych węzłów poprzez ukierunkowanie przeszukiwania w stronę celu, co może znacząco przyspieszyć znalezienie optymalnej trasy, szczególnie w przypadku rozległych sieci komunikacyjnych.

Znaczenie funkcji heurystycznej dla wydajności Odpowiednio dobrana funkcja heurystyczna ma kluczowe znaczenie dla wydajności algorytmu A*:

- Bez heurystyki (gdy $h(n) = 0$ dla wszystkich węzłów), A* redukuje się do algorytmu Dijkstry, który przeszukuje węzły we wszystkich kierunkach równomiernie.

- Z dokładną heurystyką (gdyby $h(n)$ dokładnie odpowiadała rzeczywistemu kosztowi), A* podążałby bezpośrednio do celu bez zbędnego przeszukiwania.
- W praktyce używamy przybliżonej heurystyki, która balansuje między tymi skrajnościami, znacząco redukując liczbę przeszukiwanych węzłów w porównaniu do Dijkstry.

2.2.3 Wyniki i analiza

Algorytm A* został przetestowany na tych samych trzech scenariuszach co algorytm Dijkstry, co pozwala na bezpośrednie porównanie efektywności obu podejść. Poniżej przedstawiamy wyniki eksperymentów i ich analizę, ze szczególnym uwzględnieniem czasu obliczeń oraz jakości znalezionych rozwiązań.

Porównanie czasów obliczeń Poniższa tabela przedstawia porównanie czasów obliczeń dla obu algorytmów:

Trasa	**Czas Dijkstra [s]**	**Czas A* [s]**
Śliczna → Prudnicka	0,0077	0,0052
Śliczna → most Grunwaldzki	0,0598	0,0382
Śliczna → Bezpieczna	0,1116	0,1099

Należy zaznaczyć, że w przypadku algorytmu A* czas tworzenia grafu nie jest wliczany w podany czas obliczeń. Pomimo tego, różnica jest znacząca i wskazuje na wysoką efektywność algorytmu A* w przeszukiwaniu przestrzeni rozwiązań.

Porównanie jakości rozwiązań W przypadku wszystkich trzech scenariuszy, oba algorytmy znalazły rozwiązania o identycznym koszcie czasowym:

Trasa	**Czas podróży**	**Dijkstra**	**A***
Śliczna → Prudnicka	7 min	16:25 - 16:32	16:25 - 16:32
Śliczna → most Grunwaldzki	18 min	8:50 - 9:08	8:50 - 9:08
Śliczna → Bezpieczna	35 min	16:25 - 17:00	16:25 - 17:00

Warto jednak zwrócić uwagę na różnice w znalezionych trasach:

- **Śliczna → Prudnicka**: Oba algorytmy znalazły tę samą trasę z identycznymi liniami i czasami przesiadek.
- **Śliczna → most Grunwaldzki**: Oba algorytmy znalazły trasę z 3 odcinkami, wykorzystując te same linie i przystanki przesiadkowe.
- **Śliczna → Bezpieczna**: Interesująca różnica pojawia się w tym przypadku. Podczas gdy algorytm Dijkstry znalazł trasę z 4 odcinkami:

  Linia 112: Śliczna → DWORZEC AUTOBUSOWY
  Linia 15: DWORZEC AUTOBUSOWY → Arkady (Capitol)
  Linia 7: Arkady (Capitol) → Broniewskiego
  Linia 118: Broniewskiego → Bezpieczna

  Algorytm A* znalazł krótszą trasę z 3 odcinkami:

Linia 112: Śliczna → DWORZEC AUTOBUSOWY
Linia 15: DWORZEC AUTOBUSOWY → Pomorska
Linia K: Pomorska → Bezpieczna

Oba rozwiązania mają ten sam koszt czasowy (35 minut), jednak A* znalazło trasę z mniejszą liczbą przesiadek, co może być korzystniejsze dla pasażera.

Wnioski z porównania Przeprowadzone eksperymenty wskazują na przewagę algorytmu A* nad algorytmem Dijkstry:

1. A* zapewnia krótsze czasy obliczeń
2. Jakość znalezionych rozwiązań jest identyczna w porównaniu do algorytmu Dijkstry.

2.3 Algorytm A* dla optymalizacji liczby przesiadek

2.3.1 Opis teoretyczny

W tym wariancie funkcja kosztu oraz heurystyka są dostosowane do minimalizacji liczby przesiadek między liniami komunikacyjnymi.

2.3.2 Implementacja

Implementacja algorytmu A* dla minimalizacji liczby przesiadek opiera się na tej samej podstawowej strukturze co algorytm optymalizujący czas przejazdu, ale z kluczowymi modyfikacjami dotyczącymi funkcji priorytetyzującej i porównywania węzłów. Główne różnice i elementy tej implementacji to:

1. **Struktura węzłów** – W przypadku optymalizacji przesiadek, struktura informacji o węzłach zostaje odwrócona:
   - Dla optymalizacji czasu, węzły są reprezentowane jako: `(czas, liczba_przesiadek)`
   - Dla optymalizacji przesiadek, węzły są reprezentowane jako: `(liczba_przesiadek, czas)`

   Takie odwrócenie porządku zapewnia, że podczas porównywania węzłów pierwszeństwo ma liczba przesiadek, a czas jest brany pod uwagę tylko jako kryterium drugorzędne.

2. **Kolejka priorytetowa** – Elementy w kolejce priorytetowej dla optymalizacji przesiadek mają format:

   ```
   (liczba_przesiadek, czas + heurystyka, aktualny_czas, przystanek,
                                aktualna_linia)
   ```

   W przeciwieństwie do optymalizacji czasowej, gdzie pierwszy element to `czas + heurystyka`, tutaj kluczowym czynnikiem decydującym o priorytecie jest liczba przesiadek.

3. **Funkcja heurystyczna** – Sama funkcja heurystyczna pozostaje ta sama (oparta na odległości geograficznej i szacowanym czasie), jednak jest ona wykorzystywana w inny sposób:

- Dla optymalizacji czasu, heurystyka jest dodawana do czasu: `priorytet = czas + heurystyka`
- Dla optymalizacji przesiadek, heurystyka jest dodawana do czasu, ale czas jest drugim elementem krotki: `priorytet = (przesiadki, czas + heurystyka)`

4. **Śledzenie przesiadek** – Algorytm zlicza przesiadki poprzez porównanie aktualnej linii z linią dla nowego połączenia:

```
    new_line = connection["line"]
    new_transfers = transfers
    if current_line is not None and new_line != current_line:
        new_transfers += 1

```

5. **Aktualizacja węzłów** – Dla optymalizacji przesiadek, warunek aktualizacji węzła jest oparty na porównaniu krotek `(przesiadki, czas)`:

```
    if (new_transfers, new_time) < nodes[neighbor]:
        nodes[neighbor] = (new_transfers, new_time)
        # Aktualizacja informacji o poprzedniku i dodanie do
    kolejki...

```

Poniższy fragment kodu ilustruje kluczowe różnice w implementacji A* dla optymalizacji liczby przesiadek:

```
# Poczatkowo wezly sa inicjalizowane inaczej dla roznych kryteriow
if criteria == "t":
    nodes[start_station] = (start_seconds, 0)  # (czas, przesiadki)
else:  # criteria == 's'
    nodes[start_station] = (0, start_seconds)  # (przesiadki, czas)

# Poczatkowy element kolejki priorytetowej zalezy od kryterium
if criteria == "t":
    # Priorytetyzacja czasu
    heapq.heappush(
        pq, (start_seconds + time_heuristic, 0, start_seconds,
    start_station, None)
    )
else:  # criteria == 's'
    # Priorytetyzacja przesiadek
    heapq.heappush(
        pq, (0, start_seconds + time_heuristic, start_seconds,
    start_station, None)
    )

# W glownej petli wyciagamy element z kolejki odpowiednio do kryterium
if criteria == "t":
    # Dla optymalizacji czasu
    _, transfers, current_time, current_node, current_line = heapq.
    heappop(pq)
else:  # criteria == 's'
    # Dla optymalizacji przesiadek
    transfers, _, current_time, current_node, current_line = heapq.
    heappop(pq
```

Taka implementacja zapewnia, że algorytm A* w pierwszej kolejności znajdzie trasę z minimalną liczbą przesiadek, a dopiero w drugiej kolejności zoptymalizuje czas podróży.

2.3.3 Wyniki i analiza

Algorytm A* zoptymalizowany pod kątem minimalizacji liczby przesiadek został przetestowany na tych samych trzech scenariuszach co poprzednie implementacje. Poniżej przedstawiamy wyniki oraz analizę ich skuteczności i efektywności.

Porównanie wyników Poniższa tabela zestawia wyniki dla trzech scenariuszy testowych:

Trasa	Czas oblicz. [s]	Liczba przesiadek	Czas podróży	Godziny
Łatwa	0,0463	1	12 min	16:25 - 16:37
Średnia	0,0665	2	18 min	08:50 - 09:08
Trudna	0,0326	0	11h 33min	16:27 - 04:00*

Analiza szczegółowa scenariuszy

- **Śliczna → Prudnicka**: Algorytm znalazł trasę z jedną przesiadką, która trwa 12 minut:

  Linia 112: Śliczna → DWORZEC AUTOBUSOWY (16:25 - 16:31)
  Linia 8: DWORZEC AUTOBUSOWY → Prudnicka (16:32 - 16:37)

  Warto zauważyć, że trasa ta różni się od znalezionej przez algorytm optymalizujący czas, który wykorzystał trzy przesiadki dla uzyskania podróży trwającej 7 minut.

- **Śliczna → most Grunwaldzki**: Trasa zawiera dwie przesiadki i trwa 18 minut:

  Linia 612: Śliczna → DWORZEC AUTOBUSOWY (08:50 - 08:54)
  Linia K: DWORZEC AUTOBUSOWY → DWORZEC GŁÓWNY (08:54 - 08:57)
  Linia 4: DWORZEC GŁÓWNY → most Grunwaldzki (08:57 - 09:08)

  W tym przypadku algorytm znalazł identyczną trasę jak w przypadku optymalizacji czasowej, co sugeruje, że jest to rozwiązanie optymalne pod względem obu kryteriów.

- **Śliczna → Bezpieczna**: Najciekawszy przypadek - algorytm znalazł bezpośrednie połączenie bez przesiadek:

  Linia 143: Śliczna → Bezpieczna (16:27 - 04:00 dnia następnego)

  Jest to drastyczna różnica w porównaniu do trasy znalezionej przez algorytm optymalizujący czas, który wykorzystał 4 przesiadki dla trasy trwającej 35 minut. Oznacza to, że istnieje bezpośrednie połączenie, które pozwala uniknąć przesiadek kosztem znacznie dłuższego czasu podróży (11 godzin i 33 minuty).

Efektywność obliczeniowa Interesujące jest, że czas obliczeń dla scenariusza 3 (Śliczna → Bezpieczna) był najniższy spośród wszystkich testów (0,0326 s), mimo że teoretycznie jest to najbardziej złożony problem. Wynika to prawdopodobnie z faktu, że algorytm szybko znalazł bezpośrednie połączenie, które jest optymalne pod względem liczby przesiadek, i mógł wcześnie zakończyć poszukiwania.

Kompromis czas vs. wygoda Przeprowadzone testy wyraźnie ilustrują kompromis między czasem podróży a wygodą pasażera:

- Dla trasy Śliczna → Prudnicka: redukcja z 3 do 1 przesiadki kosztem wydłużenia czasu podróży z 7 do 12 minut (+71%).
- Dla trasy Śliczna → most Grunwaldzki: identyczna trasa dla obu kryteriów.
- Dla trasy Śliczna → Bezpieczna: redukcja z 3 przesiadek do 0, kosztem znacznego wydłużenia czasu podróży z 35 minut do 11 godzin i 33 minut. To skrajny przykład kompromisu między wygodą a czasem.

Wnioski Algorytm A* zoptymalizowany pod kątem minimalizacji liczby przesiadek skutecznie znajduje trasy wymagające minimalnej liczby zmian środków transportu. Wyniki wskazują, że:

1. Dla niektórych tras możliwe jest znalezienie rozwiązań, które są optymalne zarówno pod względem czasu, jak i liczby przesiadek.
2. Dla innych tras istnieje wyraźny kompromis, gdzie redukcja liczby przesiadek wiąże się z wydłużeniem czasu podróży.
3. Algorytm jest wystarczająco szybki dla zastosowań praktycznych, z czasami obliczeń nieprzekraczającymi 0,07 sekundy dla testowanych scenariuszy.

2.4 Modyfikacje algorytmu A*

2.4.1 Opis wprowadzonych modyfikacji

W celu znaczącego przyspieszenia działania algorytmu A* wprowadzono kilka optymalizacji, które zmniejszają złożoność obliczeniową i poprawiają wydajność bez wpływu na jakość uzyskiwanych rozwiązań. Najważniejsze zaimplementowane usprawnienia to:

1. **Mechanizmy cachowania** – Wprowadzono dwupoziomowy system cachowania obliczeń:
   - Dekorator `lru_cache` dla funkcji obliczającej odległość geograficzną, co eliminuje powtarzające się kosztowne obliczenia trygonometryczne.
   - Dedykowany słownik `_heuristic_cache` dla przechowywania wyników funkcji heurystycznej między parami przystanków.
2. **Optymalizacja struktury grafu** – Przebudowano sposób reprezentacji i przetwarzania grafu:
   - Prekalkulacja unikalnych przystanków i wstępna alokacja struktury grafu.
   - Wstępne obliczanie czasu trwania przejazdów i przechowywanie ich jako część struktury grafu.
   - Sortowanie połączeń według czasu odjazdu dla efektywniejszego wyszukiwania.

2.4.2 Wyniki i porównanie z podstawową wersją

Przeprowadzone testy wydajnościowe wykazały znaczącą poprawę efektywności zoptymalizowanego algorytmu A* w porównaniu z wersją podstawową:

Scenariusz	Czas podstawowy [s]	Czas po optymalizacji [s]	Poprawa [%
Śliczna → Prudnicka	0,0052	0,0018	65,4%
Śliczna → most Grunwaldzki	0,0382	0,0094	75,4%
Śliczna → Bezpieczna	0,1099	0,0215	80,4%

Najważniejsze obserwacje z przeprowadzonych testów:

- Największy zysk wydajnościowy obserwujemy dla bardziej złożonych tras, gdzie optymalizacje przynoszą ponad 80% redukcji czasu obliczeń.
- Cachowanie heurystyki okazało się szczególnie efektywne w przypadkach, gdzie powtarzają się obliczenia dla tych samych par przystanków.

Co istotne, wprowadzone optymalizacje nie wpłynęły na jakość znajdowanych tras - algorytm nadal znajduje identyczne rozwiązania jak wersja podstawowa, ale robi to znacznie szybciej.

3 Zadanie 2: Problem odwiedzenia zbioru przystanków

3.1 Metoda przeszukiwania z zabronieniami (Tabu Search)

3.1.1 Opis teoretyczny

Tabu Search to metaheurystyka wykorzystywana do rozwiązywania problemów optymalizacyjnych. W kontekście problemu komiwojażera, metoda ta pozwala na efektywne przeszukiwanie przestrzeni rozwiązań z uniknięciem utknięcia w lokalnym optimum.

Problem komiwojażera (Traveling Salesman Problem, TSP) to jedno z klasycznych zagadnień optymalizacyjnych, w którym należy znaleźć najkrótszą trasę przechodzącą przez wszystkie zadane punkty dokładnie raz i powracającą do punktu początkowego. W kontekście systemu komunikacji miejskiej problem ten nabiera dodatkowej złożoności:

- Odległości między przystankami nie są stałe, ale zależą od aktualnych połączeń komunikacyjnych dostępnych w danym momencie.
- Czas przejazdu między przystankami zależy od rozkładu jazdy oraz ewentualnych przesiadek.
- Zamiast minimalizacji dystansu, optymalizujemy czas przejazdu lub liczbę przesiadek.
- Przestrzeń rozwiązań jest ogromna - dla n przystanków liczba możliwych permutacji wynosi $(n-1)!/2$.

Metoda przeszukiwania z zabronieniami (Tabu Search) została zaproponowana przez Freda Glovera w 1986 roku i stanowi zaawansowaną technikę metaheurystyczną, która rozszerza klasyczne metody lokalnego przeszukiwania o struktury pamięciowe. Kluczowe elementy algorytmu Tabu Search to:

1. **Lista tabu** - struktura pamięciowa przechowująca ostatnio wykonane ruchy, które są czasowo zabronione, by uniknąć cyklicznego przeszukiwania tych samych rozwiązań.

2. **Mechanizm aspiracji** - pozwala na zaakceptowanie ruchu z listy tabu, jeśli prowadzi on do rozwiązania lepszego niż najlepsze dotychczas znalezione.

3. **Intensyfikacja** - koncentracja przeszukiwania w obiecujących regionach przestrzeni rozwiązań.

4. **Dywersyfikacja** - zachęcanie do eksploracji niezbadanych obszarów przestrzeni rozwiązań.

W odróżnieniu od klasycznych algorytmów lokalnego przeszukiwania, Tabu Search może zaakceptować ruchy pogarszające obecne rozwiązanie, co umożliwia wydostanie się z lokalnych optimów. Jednocześnie, dzięki liście tabu, algorytm unika powrotu do rozwiązań już przeanalizowanych.

3.1.2 Podstawowa implementacja

Implementacja algorytmu przeszukiwania z zabronieniami dla problemu odwiedzenia zbioru przystanków obejmuje kilka kluczowych elementów. Poniżej przedstawiono szczegółowy opis implementacji:

Funkcja kosztu Funkcja `calculate_route_cost` ocenia jakość danego rozwiązania poprzez obliczenie całkowitego kosztu trasy. W zależności od wybranego kryterium optymalizacji, funkcja oblicza:

- Całkowity czas podróży - suma czasów przejazdów między kolejnymi przystankami oraz czasów oczekiwania na połączenia.

- Całkowitą liczbę przesiadek - suma wszystkich zmian linii komunikacyjnych wymaganych na trasie.

Generowanie sąsiedztwa W każdej iteracji algorytmu generujemy zbiór sąsiednich rozwiązań poprzez stosowanie dwóch typów ruchów:

- **Zamiana (swap)** - wymiana pozycji dwóch losowo wybranych przystanków w sekwencji.

- **Wstawienie (insert)** - wyjęcie przystanku z jednej pozycji i wstawienie go na inną pozycję w sekwencji.

Warto podkreślić, że w kontekście sieci komunikacji miejskiej obliczanie kosztu trasy jest złożoną operacją, która wymaga wyszukiwania optymalnych połączeń między kolejnymi przystankami przy uwzględnieniu czasów odjazdów, przyjazdów i ewentualnych przesiadek. Funkcja oceny rozwiązania wykorzystuje wcześniej zaimplementowany algorytm A* do znajdowania optymalnych połączeń między kolejnymi przystankami w sekwencji.

Dzięki zastosowaniu metaheurystyki Tabu Search możliwe jest efektywne przeszukiwanie ogromnej przestrzeni rozwiązań i znajdowanie wysokiej jakości tras w rozsądnym czasie, co byłoby niemożliwe przy użyciu metod dokładnych dla większych instancji problemu.

3.1.3 Wyniki i analiza

Podstawowa wersja algorytmu Tabu Search została przetestowana na trzech scenariuszach o rosnącym poziomie złożoności. Celem było zbadanie skuteczności algorytmu dla problemu odwiedzenia zbioru przystanków i powrotu do punktu początkowego przy optymalizacji całkowitego czasu przejazdu.

Scenariusze testowe i metodologia Przeprowadzono następujące testy:

1. **Scenariusz prosty**: Odwiedzenie jednego przystanku (Prudnicka) z powrotem do przystanku początkowego (Śliczna), z czasem startu 16:25.
2. **Scenariusz średni**: Odwiedzenie dwóch przystanków (most Grunwaldzki, GALERIA DOMINIKAŃSKA) i powrót do przystanku Śliczna, z czasem startu 8:50.
3. **Scenariusz złożony**: Odwiedzenie czterech przystanków (Bezpieczna, most Grunwaldzki, PL. GRUNWALDZKI, Ogród Botaniczny) i powrót do przystanku Śliczna, z czasem startu 16:25.

W każdym teście algorytm wykonywał określoną liczbę iteracji (domyślnie 100, w przypadku scenariusza złożonego zwiększono do 200 ze względu na większą złożoność problemu). Wszystkie testy przeprowadzono z optymalizacją czasu przejazdu jako głównego kryterium.

Wyniki eksperymentów Poniższa tabela podsumowuje wyniki uzyskane dla trzech scenariuszy testowych:

Scenariusz	**Koszt [min]**	**Czas obliczeń [s]**	**Liczba odwiedzonych przystanków**
Prosty	14	0,98	1 + początkowy
Średni	40	14,52	2 + początkowy
Złożony	76	258,55	4 + początkowy

Analiza szczegółowa trasy dla scenariusza prostego Dla prostego problemu odwiedzenia jednego przystanku (Prudnicka) i powrotu do przystanku początkowego (Śliczna), algorytm znalazł trasę o całkowitym czasie przejazdu 14 minut:

- Trasa do przystanku Prudnicka:
  - Linia 112: Śliczna → Borowska (Aquapark) (16:25–16:27)
  - Linia 124: Borowska (Aquapark) → Kamienna (16:28–16:31)
  - Linia 8: Kamienna → Prudnicka (16:31–16:32)
- Trasa powrotna:
  - Linia 18: Prudnicka → Bardzka (16:32–16:35)
  - Linia 146: Bardzka → Śliczna (16:36–16:39)

Czas obliczeń dla tego scenariusza wyniósł zaledwie 0,98 sekundy, co wskazuje na wysoką efektywność algorytmu dla prostych problemów.

Analiza szczegółowa trasy dla scenariusza średniego Dla średniego problemu odwiedzenia dwóch przystanków i powrotu, algorytm znalazł trasę o całkowitym czasie przejazdu 40 minut:

- Trasa do przystanku most Grunwaldzki:
  - Linia 612: Śliczna → DWORZEC AUTOBUSOWY (08:50–08:54)
  - Linia K: DWORZEC AUTOBUSOWY → DWORZEC GŁÓWNY (08:54–08:57)
  - Linia 4: DWORZEC GŁÓWNY → most Grunwaldzki (08:57–09:08)
- Trasa do GALERIA DOMINIKAŃSKA:
  - Linia 145: most Grunwaldzki → Poczta Główna (09:09–09:11)
  - Linia N: Poczta Główna → GALERIA DOMINIKAŃSKA (09:12–09:15)
- Trasa powrotna:
  - Linia N: GALERIA DOMINIKAŃSKA → DWORZEC AUTOBUSOWY (09:15–09:22)
  - Linia 113: DWORZEC AUTOBUSOWY → Śliczna (09:24–09:30)

Czas obliczeń dla tego scenariusza wyniósł 14,52 sekundy, co stanowi około 15-krotny wzrost w porównaniu ze scenariuszem prostym.

Analiza szczegółowa trasy dla scenariusza złożonego Dla złożonego problemu odwiedzenia czterech przystanków, algorytm znalazł trasę o całkowitym czasie przejazdu 76 minut. Ze względu na złożoność trasy, przedstawiono ją w skróconej formie:

- Śliczna → Bezpieczna (16:25–17:00) - z wykorzystaniem linii 112, 15 i K
- Bezpieczna → Ogród Botaniczny (17:00–17:18) - z wykorzystaniem linii K, 930 i 23
- Ogród Botaniczny → PL. GRUNWALDZKI (17:18–17:23) - z wykorzystaniem linii 111
- PL. GRUNWALDZKI → most Grunwaldzki (17:23–17:24) - z wykorzystaniem linii 13
- most Grunwaldzki → Śliczna (17:24–17:41) - z wykorzystaniem linii 13, 16, 134 i 112

Czas obliczeń dla tego scenariusza wyniósł 258,55 sekundy (ponad 4 minuty), co pokazuje znaczący wzrost złożoności obliczeniowej wraz ze zwiększeniem liczby przystanków do odwiedzenia.

Analiza skalowania algorytmu Na podstawie przeprowadzonych testów można zauważyć, że czas obliczeń algorytmu Tabu Search rośnie wykładniczo wraz ze wzrostem liczby przystanków do odwiedzenia:

- Dla 1 przystanku: 0,98 sekundy
- Dla 2 przystanków: 14,52 sekundy (około 15-krotny wzrost)
- Dla 4 przystanków: 258,55 sekundy (około 18-krotny wzrost względem scenariusza średniego)

Ten wykładniczy wzrost czasu obliczeń jest zgodny z teoretyczną złożonością problemu komiwojażera, gdzie liczba możliwych permutacji stacji rośnie jako $(n-1)!/2$ dla n przystanków.

Wnioski Podstawowa implementacja algorytmu Tabu Search skutecznie znajduje rozwiązania dla problemu odwiedzenia zbioru przystanków w systemie komunikacji miejskiej, jednak z pewnymi ograniczeniami:

1. Algorytm jest wysoce efektywny dla małych instancji problemu (1-2 przystanki), z czasami obliczeń poniżej 15 sekund.
2. Dla większych instancji (4 przystanki) czas obliczeń znacząco rośnie, osiągając ponad 4 minuty, co może być problematyczne dla aplikacji działających w czasie rzeczywistym.
3. Jakość znalezionych rozwiązań wydaje się być dobra, z logicznymi sekwencjami przystanków i efektywnym wykorzystaniem dostępnych połączeń.
4. Zauważalna jest tendencja algorytmu do preferowania przystanków położonych w podobnych rejonach miasta, co sugeruje, że algorytm skutecznie wykorzystuje strukturę geograficzną sieci komunikacyjnej.

Wyniki wskazują na potrzebę dalszych modyfikacji algorytmu w celu poprawy jego efektywności dla większych instancji problemu. W kolejnych sekcjach zbadamy wpływ modyfikacji takich jak adaptacyjna długość listy tabu, kryterium aspiracji oraz strategie próbkowania sąsiedztwa na wydajność algorytmu.

3.2 Modyfikacja długości listy tabu

Długość listy tabu ma istotny wpływ na efektywność algorytmu. Zbyt krótka lista może prowadzić do cyklicznego przeszukiwania tych samych rozwiązań, natomiast zbyt długa może nadmiernie ograniczać przestrzeń poszukiwań.

W ramach modyfikacji algorytmu Tabu Search zaimplementowano mechanizm dynamicznego dostosowywania długości listy tabu w zależności od rozmiaru problemu. Istotą tego podejścia jest dostosowanie parametru tabu_size do liczby przystanków w trasie.

Implementacja tej modyfikacji opiera się na kilku kluczowych zasadach:

1. **Dynamiczne obliczanie rozmiaru listy** – Rozmiar listy tabu jest obliczany na podstawie liczby stacji do odwiedzenia, a nie ustalany jako stała wartość.

2. **Wzór oparty na możliwych ruchach** – Optymalny rozmiar listy jest szacowany na podstawie liczby możliwych operacji swap i insert, które rosną kwadratowo wraz z liczbą przystanków.

3. **Funkcja pierwiastkowa** – Zastosowano skalowanie z użyciem pierwiastka kwadratowego liczby możliwych ruchów, co zapewnia rozsądny wzrost rozmiaru listy wraz ze wzrostem złożoności problemu.

4. **Ograniczenia brzegowe** – Wprowadzono wartości minimalne i maksymalne (5-50), aby zapewnić odpowiedni zakres długości listy tabu niezależnie od wielkości problemu.

3.3 Modyfikacja z kryterium aspiracji

Kryterium aspiracji pozwala na akceptację ruchów tabu, jeśli prowadzą do lepszego rozwiązania niż najlepsze dotychczas znalezione lub znacząco poprawiają bieżące rozwiązanie.

Zaawansowane kryterium aspiracji opiera się na trzech mechanizmach:

1. Akceptacja ruchów tabu poprawiających najlepsze rozwiązanie.

2. Akceptacja ruchów poprawiających bieżące rozwiązanie o co najmniej 5%.

3. Dywersyfikacja w późniejszych iteracjach (30% szans na akceptację tabu ruchu).

Mechanizm ten zwiększa eksplorację przestrzeni rozwiązań i unikanie lokalnych minimów.

3.4 Strategia próbkowania sąsiedztwa

Efektywne próbkowanie sąsiedztwa bieżącego rozwiązania może znacząco wpłynąć na wydajność algorytmu, szczególnie dla dużych instancji problemu.

Strategia próbkowania sąsiedztwa została zaimplementowana w celu ograniczenia liczby generowanych sąsiadów w każdej iteracji algorytmu Tabu Search. Wykorzystano trzy podejścia:

- **Vertex sampling** – generowanie sąsiadów poprzez zamianę pozycji dwóch wierzchołków w sekwencji.

- **Edge sampling** – odwracanie kolejności wierzchołków w wybranym podciągu sekwencji.

- **Edge-vertex sampling** – kombinacja zamiany wierzchołków i odwracania podciągów.

Każda strategia umożliwia dynamiczne ograniczenie liczby sąsiadów do ustalonej wartości, co pozwala na redukcję czasu obliczeń przy zachowaniu jakości rozwiązań.

3.5 Wyniki i analiza zmodyfikowanego algorytmu

Po wprowadzeniu modyfikacji algorytmu Tabu Search, przeprowadzono ponowne testy dla trzech scenariuszy: prostego, średniego i złożonego. Wyniki wskazują na znaczną poprawę efektywności obliczeniowej przy zachowaniu jakości rozwiązań.

Porównanie czasów obliczeń Poniższa tabela przedstawia porównanie czasów obliczeń przed i po modyfikacjach:

Scenariusz	Przed modyfikacjami [s]	Po modyfikacjach [s]	Zmiana (%)
Prosty	0.98	1.15	+17.35%
Średni	14.52	7.55	-48.00%
Złożony	258.55	45.23	-82.51%

Analiza wyników

- **Scenariusz prosty:** Czas obliczeń nieznacznie wzrósł (+17.35%), co wynika z dodatkowych operacji związanych z dynamiczną długością listy tabu. Jakość rozwiązania pozostała bez zmian, a trasa była identyczna.
- **Scenariusz średni:** Czas obliczeń zmniejszył się o 48%, co wskazuje na skuteczność strategii próbkowania sąsiedztwa. Znalezione rozwiązanie było identyczne pod względem kosztu i trasy.
- **Scenariusz złożony:** Największa poprawa wydajności (-82.51%) została osiągnięta dzięki dynamicznej długości listy tabu i ograniczeniu liczby generowanych sąsiadów. Trasa i koszt pozostały bez zmian, co potwierdza skuteczność modyfikacji.

Koszt trasy przy każdym scenariuszu pozostał taki sam przed i po modyfikacjach, co może sugerować znalezienie optymalnego rozwiązania.

Wnioski z analizy Modyfikacje algorytmu Tabu Search znacząco poprawiły jego efektywność obliczeniową, szczególnie dla bardziej złożonych scenariuszy. Kluczowe zmiany, takie jak dynamiczna długość listy tabu, kryterium aspiracji i strategia próbkowania sąsiedztwa, pozwoliły na redukcję czasu obliczeń bez wpływu na jakość rozwiązań. Algorytm jest teraz bardziej skalowalny i lepiej nadaje się do zastosowań w czasie rzeczywistym.

4 Podsumowanie i wnioski

4.1 Napotkane problemy implementacyjne

W trakcie implementacji algorytmów napotkano szereg problemów, między innymi:

- W miarę dodawania kolejnych modyfikacji (takich jak dynamiczne dostosowywanie długości listy tabu, kryterium aspiracji czy zaawansowane strategie próbkowania sąsiedztwa) kod stawał się coraz bardziej złożony, dlatego konieczne było stworzenie dedykowanych testów jednostkowych.
- Trudności w debugowaniu złożonych struktur danych, takich jak grafy i kolejki priorytetowe, co wymagało dodatkowego logowania i analizy.
- Problemy związane z poprawnym przeliczaniem godzin w cyklu 24-godzinnym, co wpływało na dokładność obliczeń czasu przejazdu. Dodatkowo niepoprawne godziny takie jak 28:00.

4.2 Wnioski

Podsumowując sprawozdanie, można stwierdzić, że zaawansowane algorytmy, takie jak Dijkstry i A*, demonstrują wysoką skuteczność w kontekście znajdowania optymalnych tras. Szczególnie wyróżnia się tutaj algorytm A* z zaimplementowaną funkcją heurystyczną oraz dodatkowymi optymalizacjami, które znacząco poprawiają szybkość działania w złożonych scenariuszach - nierzadko końcowy rezultat był obliczany poniżej 100 milisekund. Warto także podkreślić, że te algorytmy można łatwo modyfikować pod oczekiwane rezultaty, podmieniając funkcję kosztu.

Znalezienie optymalnego rozwiązania dla problemów kombinatorycznych, takich jak problem komiwojażera, nadal stanowi wyzwanie. W tym zakresie, zastosowanie Tabu Search wraz z kilkoma ulepszeniami, takimi jak dynamiczna lista tabu czy kryteria aspiracji, pokazuje obiecujące rezultaty w zwiększeniu efektywności rozwiązań, jednak wciąż zajmuje znaczącą ilość czasu - tutaj przydałyby się dalsze optymalizacje.

5 Wykorzystane biblioteki

- heapq - Do implementacji kolejki priorytetowej w algorytmach przeszukiwania grafu
- pandas - Do wczytywania i przetwarzania danych z plików CSV