Problem komiwojażera z utrudnieniami na trasie

Podstawy teoretyczne

https://pl.wikipedia.org/wiki/Problem_komiwoja%C5%BCera
https://pages.mini.pw.edu.pl/~kaczmarskik/MiNIwyklady/grafy/prob-komiw.html
https://www.guru99.com/pl/travelling-salesman-problem.html
https://pl.wikipedia.org/wiki/Algorytm_mr%C3%B3wkowy
http://algorytmy.ency.pl/artykul/algorytm_najblizszego_sasiada
https://www.guru99.com/pl/travelling-salesman-problem.html

Cel projektu

Celem projektu jest zoptymalizowanie trasy pomiędzy pewnymi miastami, z losowo występującymi problemami na trasie, które uniemożliwiają pokonanie niektórych dróg. Rozważymy wersję algorytmu, w której na wejściu otrzymuje się rozwiązanie dla grafu pełnego. Zgodnie z założeniem algorytm będzie wykorzystywał rozwiązanie dla grafu pełnego, aby wykorzystać te fragmenty trasy, które nie muszą się zmieniać ze względu na brak utrudnień po drodze. Symulacja sytuacji, w której występuje utrudnienie na trasie (np. zawalony most) i trzeba je ominąć za pomocą objazdu. Program ma za zadanie znaleźć trasę o najmniejszym koszcie, która przechodzi przez wszystkie miasta — wierzchołki grafu. Będzie to zatem cykl Hamiltona o najmniejszej sumie wag krawędzi.

Generator danych wejściowych

GenerowanieGrafu.py

Program generuje graf nieskierowany o n wierzchołkach z losowymi wagami z przedziału. Wartość n jest podana przez użytkownika. Graf jest przedstawiony za pomocą macierzy przejść. Pola [x][x] macierzy zostały wypełnione wartością inf (nieskończoność), aby upewnić się, że kolejne algorytmy nie skorzystają z tych elementów w rozwiązaniu.

```
import random

# Generuje graf nieskierowany o n wierzchołkach z losowymi wagami z przedziału
def generate_graph(n, weight_range=(1, 100)):
    graph = []
    for i in range(n):
        row = []
        for j in range(n):
            row.append(float('inf'))
        graph.append(row)

    for i in range(n):
        for j in range(i + 1, n):
            weight = random.randint(*weight_range)
            graph[i][j] = weight
            graph[j][i] = weight
```

```
    return graph
```

UsuwanieDrog.py

Algorytm usuwa losowe D dróg. Losuje dwa miasta, upewniając się, że nie trafił dwa razy na to samo miasto oraz że między miastami istnieje jakakolwiek droga. Następnie zamienia wartość tej drogi na nieskończoność (inf), co w następnych algorytmach pozwala na rozpoznanie drogi jako nieistniejącej. Program przy każdym usunięciu drogi upewnia się, że przy wierzchołku pozostanie wystarczająco dużo połączeń, aby możliwe było znalezienie rozwiązania.

```
import random

#Usuwa D dróg z grafu o V wierzchołkach podanego za pomocą macierzy przejść
def deleting_roads(graph, D, V):
    for i in range(D):
        x = random.randint(0, V-1)
        while (graph[x].count(float('inf'))>=V-2):
            x = random.randint(0, V-1)
        y=x
        while (x==y or graph[y].count(float('inf'))>=V-2):
            y = random.randint(0, V-1)
        graph[x][y] = float('inf')
        graph[y][x] = float('inf')
    return graph
```

Algorytmy przeszukiwania

Algorytm Brute Force

Program startuje z wcześniej określonego wierzchołka i tworzy wszystkie możliwe kombinacje dróg. Dla każdej z tych kombinacji sprawdza ich koszt i poszukuje najmniejszego. Program pozwala na wyznaczenie najoptymalniejszego wyniku, jednak ze względu na złożoność n! jest wydajny jedynie dla grafów zawierających kilkanaście miast.

```
from itertools import permutations
from sys import maxsize

def tsp(graph, s, V):

    vertex = [i for i in range(V) if i != s]
    min_cost = maxsize
    next_permutation = permutations(vertex)

    for i in next_permutation:
        current_cost = 0
        k = s

        for j in i:
            current_cost += graph[k][j]
            k = j
```

```
        current_cost += graph[k][s]
        if current_cost < min_cost:
            min_cost = current_cost
            wynik = i
    # return wynik
    return min_cost, wynik
```

Program ten można wykorzystać do obliczenia rozwiązania dla grafu z nieprzejezdnymi drogami. Dzięki temu, że drogi nieprzejezdne mają wartość nieskończoną, nigdy nie trafią do rozwiązania optymalnego. Podczas próby obliczenia wyniku dla grafu niepełnego z wykorzystaniem rozwiązania przed usunięciem dróg, w pewnym momencie trzeba wymusić, aby algorytm skorzystał z dostępnej ścieżki. W ten sposób przestanie to być metoda Brute Force, ponieważ algorytm nie sprawdzi wszystkich możliwych opcji. Z powodu specyfiki zadania nie da się napisać algorytmu, który będzie w pełni opierał się na metodzie Brute Force w takim przypadku.

Algorytm Zachłanny

Algorytm poszukuje najbardziej optymalnego rozwiązania w danej chwili. Zaczyna od wierzchołka początkowego i szuka drogi o najmniejszej wartości. W ten sposób tworzy trasę przechodzącą przez wszystkie wierzchołki. Taki algorytm niestety nie daje najbardziej optymalnego rozwiązania, jednak jest zdecydowanie bardziej wydajny czasowo dzięki swojej złożoności O(n^2).

```
def greedy_tsp(graph, s):
    n = len(graph)
    visited = [False] * n
    solution = []
    total_cost = 0
    current_city = s
    visited[current_city] = True
    for _ in range(n - 1):
        next_city = None
        min_cost = float('inf')
        for city in range(n):
            if not visited[city] and graph[current_city][city] < min_cost:
                min_cost = graph[current_city][city]
                next_city = city
        if next_city is not None:
            solution.append(next_city)
            visited[next_city] = True
            total_cost += min_cost
            current_city = next_city
    total_cost += graph[current_city][s]
    return total_cost, solution
```

Ze względu na specyfikę danych wejściowych, po usunięciu części dróg program nadal będzie działał. Istnieje jednak szansa, że nie znajdzie rozwiązania w sytuacji, gdy ostatni wierzchołek nie będzie miał krawędzi łączącej go z początkiem trasy.

W poniższym algorytmie, aby ułatwić znalezienie ścieżki w grafie niepełnym, wykorzystuje się rozwiązanie optymalne dla grafu pełnego. Program podąża wcześniej wyznaczoną ścieżką do momentu, aż natrafi na przeszkodę w postaci nieistniejącej krawędzi. Wtedy, korzystając z metody zachłannej, wyznacza kolejny najmniej kosztowy wierzchołek. W kolejnym wywołaniu pętli

algorytm wraca do wcześniej wyznaczonej optymalnej ścieżki, odnajduje wierzchołek, na którym aktualnie się znajduje, i dalej próbuje podążać tą ścieżką od tego miejsca. Tym sposobem otrzymujemy rozwiązanie, które nie jest najbardziej optymalne, ale zdecydowanie jest wydajne czasowo.

```
def greedy_tsp_d(graph, vector, V):
    solution = []
    prev = 0
    i = vector[0]
    cost = 0
    for _ in range(V-1):
        if graph[prev][i] != float('inf') and i not in solution and i != 0:
            solution.append(i)
            cost += graph[prev][i]
            prev = i

        else:
            prev = i
            min_value = float('inf')
            for j in range(V):
                if min_value>graph[prev][j] and j not in solution and j != 0:
                    min_value = graph[prev][j]
                    i=j

            solution.append(i)
            cost += graph[prev][i]

        if vector.index(i)<V-2:
            i = vector[vector.index(i)+1]
    cost += graph[vector[V-2]][0]
    return solution, cost
```

Algorytm metaheurystyczny

Poniżej został przedstawiony algorytm mrówkowy dla problemu komiwojażera. Na początku programu tworzona jest macierz feromonów, która będzie modyfikowana na podstawie danych podanych jako wejście funkcji. Każda mrówka buduje trasę, zaczynając od losowego wierzchołka, przechodzi na kolejne wierzchołki, kierując się prawdopodobieństwem, które zależy od feromonów oraz heurystyki. Po ukończeniu trasy liczony jest jej całkowity koszt. W ten sposób zliczane są koszty wszystkich kolejnych tras i szukany jest ten najmniejszy. Każda kolejna iteracja, ze względu na “odparowywanie” feromonów, staje się coraz dokładniejsza.

```
import random
import numpy as np

def ant_colonyTSP(graph, num_ants, num_iterations, alpha, beta, rho, q):
    """
    graph: Macierz sąsiedztwa (kosztów przejścia między miastami)
    num_ants: Liczba mrówek
    num_iterations: Liczba iteracji algorytmu
```

```
alpha: Współczynnik znaczenia feromonów
beta: Współczynnik znaczenia heurystyki
rho: Współczynnik odparowywania feromonów
q: Ilość feromonów dodawanych przez mrówki
"""
num_cities = len(graph)
pheromones = np.ones((num_cities, num_cities))  # Inicjalizacja feromonów
best_path = None
best_cost = float('inf')

for _ in range(num_iterations):
    all_paths = []
    all_costs = []

    for ant in range(num_ants):
        # Budowanie trasy dla jednej mrówki
        visited = [False] * num_cities
        current_city = random.randint(0, num_cities - 1)
        path = [current_city]
        visited[current_city] = True
        total_cost = 0

        for _ in range(num_cities - 1):
            probabilities = []
            for next_city in range(num_cities):
                if not visited[next_city]:
                    pheromone = pheromones[current_city][next_city] ** alpha
                    if graph[current_city][next_city] != float('inf'):
                        heuristic = (1 / graph[current_city][next_city]) ** beta
                    else:
                        heuristic=0
                    probabilities.append(pheromone * heuristic)
                else:
                    probabilities.append(0)
            probabilities = np.array(probabilities) / sum(probabilities)
            next_city = np.random.choice(range(num_cities), p=probabilities)
            path.append(next_city)
            visited[next_city] = True
            total_cost += graph[current_city][next_city]
            current_city = next_city

        # Powrót do miasta początkowego
        total_cost += graph[current_city][path[0]]
        path.append(path[0])
        all_paths.append((path, total_cost))
        all_costs.append(total_cost)

        # Aktualizacja najlepszego rozwiązania
        if total_cost < best_cost:
```

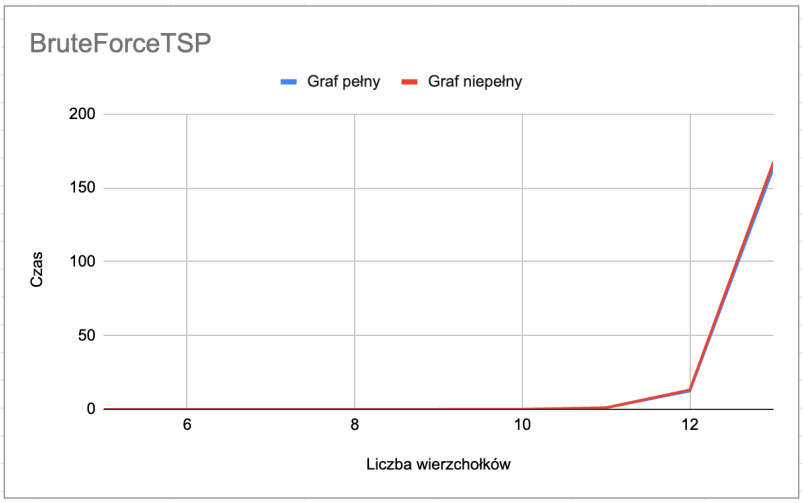
```
            best_cost = total_cost
            best_path = path

        # Aktualizacja feromonów
        pheromones *= (1 - rho)  # Odparowanie feromonów
        for path, cost in all_paths:
            for i in range(len(path) - 1):
                pheromones[path[i]][path[i+1]] += q / cost
    path = [int(x) for x in best_path]
    return path, best_cost
```

Poniższy algorytm został przekształcony, aby wyznaczał rozwiązanie dla grafu z usuniętymi krawędziami. Aby uniknąć nieistniejących dróg, wartości tych dróg w tablicy pheromones zostały ustawione na 0. Dzięki temu nie zostają one wybierane przez mrówki. Wartości dla krawędzi wykorzystanych w rozwiązaniu dla grafu pełnego otrzymały większą wartość w tablicy pheromones, co zapewnia, że mrówki będą je chętniej wybierały. Ostatnia modyfikacja to dodanie funkcji break w pętli, aby nie uwzględniać kombinacji, w których nie da się przejść dalej z powodu braku połączenia z wierzchołkiem początkowym.

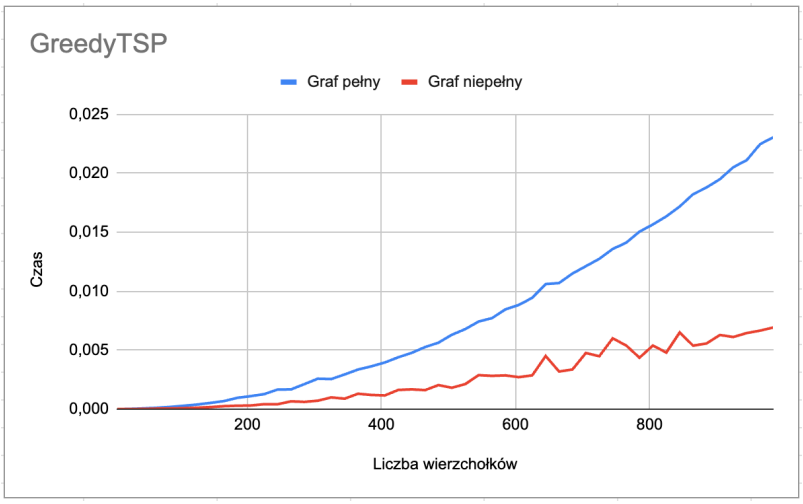
```
import random
import numpy as np
def ant_colony_deletedTSP(graph, num_ants, num_iterations, alpha, beta, rho, q, vector):
    """
    graph: Macierz sąsiedztwa (kosztów przejścia między miastami)
    num_ants: Liczba mrówek
    num_iterations: Liczba iteracji algorytmu
    alpha: Współczynnik znaczenia feromonów
    beta: Współczynnik znaczenia heurystyki
    rho: Współczynnik odparowywania feromonów
    q: Ilość feromonów dodawanych przez mrówki
    vector: rozwiazanie dla grafu pełnego
    """
    num_cities = len(graph)
    pheromones = np.ones((num_cities, num_cities))  # Inicjalizacja feromonów
    prev = vector[0]
    for i in range(1,len(vector)):
        pheromones[prev][vector[i]] = 3
        prev = vector[i]
    for i in range (num_cities):
        for j in range (num_cities):
            if graph[i][j] == float('inf'):
                pheromones[i][j]=0
    best_path = None
    best_cost = float('inf')

    for _ in range(num_iterations):
        all_paths = []
        all_costs = []

        for ant in range(num_ants):
```

```
        # Budowanie trasy dla jednej mrówki
        visited = [False] * num_cities
        current_city = random.randint(0, num_cities - 1)
        path = [current_city]
        visited[current_city] = True
        total_cost = 0

        for _ in range(num_cities - 1):

            probabilities = []
            for next_city in range(num_cities):
                if not visited[next_city] and pheromones[current_city][next_city] != 0:
                    pheromone = pheromones[current_city][next_city] ** alpha
                    if graph[current_city][next_city] != float('inf'):
                        heuristic = (1 / graph[current_city][next_city]) ** beta
                    else:
                        heuristic=0
                    probabilities.append(pheromone * heuristic)
                else:
                    probabilities.append(0)
            if sum(probabilities) == 0:
                break
            probabilities = np.array(probabilities) / sum(probabilities)
            next_city = np.random.choice(range(num_cities), p=probabilities)
            path.append(next_city)
            visited[next_city] = True
            total_cost += graph[current_city][next_city]
            current_city = next_city

        # Powrót do miasta początkowego
        total_cost += graph[current_city][path[0]]
        path.append(path[0])
        all_paths.append((path, total_cost))
        all_costs.append(total_cost)

        # Aktualizacja najlepszego rozwiązania
        if total_cost < best_cost and len(path) == num_cities+1:
            best_cost = total_cost
            best_path = path

    # Aktualizacja feromonów
    pheromones *= (1 - rho)  # Odparowanie feromonów
    for path, cost in all_paths:
        for i in range(len(path) - 1):
            pheromones[path[i]][path[i+1]] += q / cost
path = [int(x) for x in best_path]
return path, best_cost
```

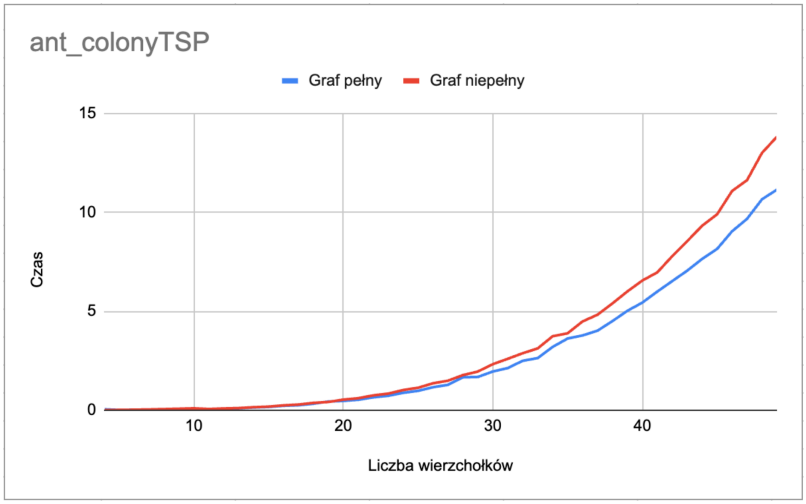
Analiza wyników



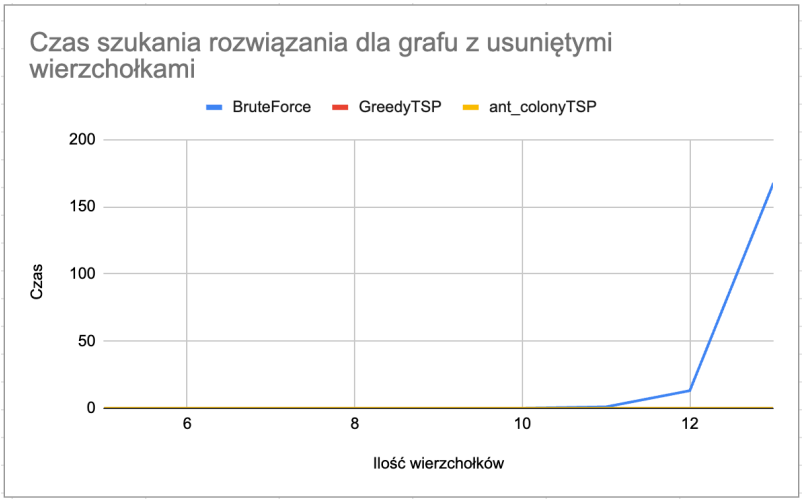
Czas wykonywania algorytmów Brute Force:



Czas wykonywania algorytmów zachłannych:



Czas wykonywania algorytmów mrówkowych:



Czas rozwiązywania problemu komiwojażera z losowo usuniętymi wierzchołkami:

Podsumowanie

W ramach analizy problemu komiwojażera z utrudnieniami na trasie przeprowadzono testy trzech różnych algorytmów, które różnią się zarówno pod względem wydajności czasowej, jak i jakości otrzymanych rozwiązań. Poniżej przedstawiono kluczowe wnioski z przeprowadzonych badań:

1. Algorytm Brute Force

Algorytm brute force, choć gwarantuje znalezienie optymalnego rozwiązania, jest najbardziej czasochłonny spośród wszystkich analizowanych metod. Jego złożoność obliczeniowa powoduje, że praktyczne zastosowanie jest ograniczone do grafów o maksymalnie 13 wierzchołkach. Dla większych instancji problemu czas obliczeń rośnie wykładniczo, co uniemożliwia efektywne wykorzystanie tej metody. Niestety, optymalizacja algorytmu pod kątem wykorzystania rozwiązań dla grafu pełnego w przypadku grafów z wykluczonymi krawędziami nie przyniosła oczekiwanych rezultatów. Niemniej jednak brute force pozostaje nieoceniony jako narzędzie referencyjne do oceny jakości innych algorytmów.

2. Algorytm Zachłanny

Algorytm zachłanny wyróżnia się niezwykle krótkim czasem obliczeń, co czyni go idealnym rozwiązaniem w przypadku dużych grafów. W porównaniu do brute force, jest w stanie obsłużyć znacznie większe instancje problemu, dostarczając wyniki w ułamku sekundy. Jakość rozwiązań, choć nie zawsze optymalna, jest wystarczająco dobra dla wielu praktycznych zastosowań. Co istotne, w przypadku grafów z usuniętymi krawędziami, algorytm zachłanny z powodzeniem wykorzystuje wyniki uzyskane dla grafu pełnego, co znacząco poprawia jego wydajność.

3. Algorytm Mrówkowy

Algorytm mrówkowy, jako zaawansowana metoda metaheurystyczna, stanowi kompromis między czasem wykonywania a jakością wyników. Jego czas obliczeń jest dłuższy niż w przypadku algorytmu zachłannego, ale znacznie krótszy niż dla brute force. Dla grafów do 60 wierzchołków algorytm jest w stanie dostarczyć rozwiązania w akceptowalnym czasie, nieprzekraczającym 15 sekund. Co więcej, wyniki są zbliżone do optymalnych, co czyni go atrakcyjnym narzędziem do rozwiązywania problemów o większej skali. W przypadku grafów z wykluczonymi krawędziami, algorytm skutecznie wykorzystuje rozwiązanie początkowe dla grafu pełnego, choć wiąże się to ze wzrostem czasu obliczeń.

Wnioski Końcowe

Każdy z analizowanych algorytmów ma swoje unikalne zalety i wady, które determinują jego zastosowanie w różnych kontekstach:

• Brute force jest idealny jako metoda referencyjna dla małych instancji problemu, gdzie kluczowa jest dokładność wyniku.

• Algorytm zachłanny to doskonały wybór w sytuacjach, gdzie priorytetem jest szybkość działania, a niewielkie odstępstwa od optymalnego rozwiązania są akceptowalne.

• Algorytm mrówkowy stanowi uniwersalne rozwiązanie, które pozwala znaleźć zbalansowane podejście pomiędzy jakością wyników a czasem obliczeń, szczególnie w przypadku większych grafów.

Projekt pokazuje, że dobór algorytmu powinien być uzależniony od specyfiki problemu oraz dostępnych zasobów czasowych i obliczeniowych. Opracowane modyfikacje i analizy umożliwiły lepsze zrozumienie zachowania algorytmów w warunkach rzeczywistych, co czyni ten projekt wartościowym wkładem w rozwiązanie problemu komiwojażera.