МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КРЕМЕНЧУЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ МИХАЙЛА ОСТРОГРАДСЬКОГО

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ЕЛЕКТРИЧНОЇ ІНЖЕНЕРІЇ
ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА АВТОМАТИЗАЦІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

НАВЧАЛЬНА ДИСЦИПЛІНА
«**АЛГОРИТМИ І СТРУКТУРИ ДАНИХ**»

ЗВІТ
З ПРАКТИЧНОЇ РОБОТИ №8

Виконав:
студент групи КН-24-1
Соломка Б. О.

Перевірив:
доцент кафедри АІС
Сидоренко В. М.

Кременчук 2025

Тема: Жадібні алгоритми. Розв'язання задачі комівояжера.

Мета: Ознайомитися з принципами роботи жадібних алгоритмів, їх перевагами та недоліками. Розв'язати задачу комівояжера для заданого графа використовуючи метод повного перебору та алгоритм найближчого сусіда.

Хід роботи

Переваги жадібних алгоритмів:

Простота розуміння та реалізації

Надають хороші наближені розв'язки

Ефективні для задач з великою розмірністю

Недоліки жадібних алгоритмів:

Не завжди знаходять глобально оптимальний розв'язок

Ефективність залежить від структури задачі

Складність може бути NP-складною для деяких задач

Задача комівояжера: Задано неорієнтований граф з n вершин-міст і dij = d(vi, vj) – позитивні цілі відстані між містами. Потрібно знайти найменшу можливу довжину гамільтонового циклу – кільцевого маршруту, який проходить по одному разу через усі міста.

Точний розв'язок можна знайти методом повного перебору, але він ефективний лише для графів з невеликою кількістю вершин. Для графів великої розмірності використовують наближені методи, наприклад, жадібний алгоритм найближчого сусіда.

2. Порівняння алгоритмів

Алгоритм	**Асимптотична складність**
Груба сила (повний перебір)	$O(n!)$
Найближчий сусід	$O(n^2 \cdot \log n)$

3. Розв'язання задачі для заданого графа

Згідно з варіантом №3, заданий зважений граф: [(1,3,8), (1,2,5), (2,3,6), (2,4,4), (3,4,3)]

3.1 Візуалізація графа

```
import networkx as nx
import matplotlib.pyplot as plt
from itertools import permutations

# Створення графа
G = nx.Graph()
G.add_nodes_from(range(1, 5))
G.add_weighted_edges_from([(1, 3, 8), (1, 2, 5), (2, 3, 6), (2, 4, 4),
(3, 4, 3)])

# Візуалізація графа
plt.figure(figsize=(10, 6))
pos = nx.spring_layout(G, seed=42)  # Позиції для вузлів

# Вузли та їх мітки
nx.draw_networkx_nodes(G, pos, node_size=700)
nx.draw_networkx_labels(G, pos, font_size=20, font_family='sans-serif')

# Ребра
edges = [(u, v) for (u, v, d) in G.edges(data=True)]
nx.draw_networkx_edges(G, pos, edgelist=edges)

# Підписи ребер (ваги)
edge_labels = dict([((u, v,), d['weight']) for u, v, d in
G.edges(data=True)])
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

plt.axis('off')
plt.title('Зважений неорієнтований граф')
plt.show()
```

3.2 Алгоритм повного перебору (груба сила)

```
# Константа, яка представляє нескінченність
INFINITY = pow(10, 20)

# Функція для обчислення довжини шляху
def get_path_length(path, G):
    path_length = 0
    # Проходимо по всіх вершинах шляху
    for i, v1 in enumerate(path):
```

```
        # Знаходимо наступну вершину у шляху
        v2 = path[(i + 1) % len(path)]
        # Перевіряємо, чи існує ребро між поточною та наступною вершиною
        if not G.has_edge(v1, v2):
            # Якщо ребра не існує, повертаємо нескінченність
            return INFINITY
        # Додаємо вагу ребра до довжини шляху
        path_length += G[v1][v2]["weight"]
    return path_length

# Функція, яка генерує всі можливі перестановки вершин графа,
# починаючи із заданої вершини
def node_permutations(G, init_node_index):
    nodes = list(G.nodes())
    # Видаляємо початкову вершину з переліку для перестановок
    init_node = nodes[init_node_index]
    nodes.remove(init_node)
    # Генеруємо всі можливі перестановки залишкових вершин
    return    [[init_node]    +    list(a_tuple)    for    a_tuple    in
permutations(nodes)]

# Головна функція для розв'язання задачі комівояжера за
# допомогою перебору всіх можливих шляхів
def TSP_BruteForce(G, init_node_index):
    min_path = None
    min_path_length = None
    # Перебираємо всі можливі шляхи із заданою початковою вершиною
    for path in node_permutations(G, init_node_index):
        # Обчислюємо довжину поточного шляху
        path_length = get_path_length(path, G)
        # Порівнюємо довжину поточного шляху з мінімальною
        # довжиною, яка була знайдена до цього моменту
        if min_path is None or min_path_length > path_length:
            # Якщо поточний шлях коротший, ніж попередній найкоротший,
            # оновлюємо мінімальний шлях та його довжину
            min_path, min_path_length = path, path_length
    # Замикаємо шлях, додаючи початкову вершину в кінець
    min_path.append(min_path[0])
    # Повертаємо найкоротший шлях та його довжину
    return min_path, min_path_length

#  Знаходження  оптимального  маршруту  (початкова  вершина  -  0  індекс  в
```

```
списку, тобто вершина 1)
    optimal_path, optimal_length = TSP_BruteForce(G, 0)
    print(f"Оптимальний маршрут методом повного перебору: {optimal_path}. 
Його вартість = {optimal_length}")
```

3.3 Алгоритм найближчого сусіда

```
import numpy as np

def nearest_neighbor_algorithm(G, start_node):
    nodes = list(G.nodes())
    N = len(nodes)
    visited = [False] * N
    tour = []

    # Знаходимо індекс стартової вершини у списку вершин
    current_index = nodes.index(start_node)
    current_node = nodes[current_index]

    # Початок маршруту
    tour.append(current_node)
    visited[current_index] = True

    # Проходимо через всі інші вершини
    for _ in range(1, N):
        previous_node = current_node
        min_distance = float('inf')
        next_node = None
        next_index = None

        # Шукаємо найближчу невідвідану вершину
        for i, node in enumerate(nodes):
            if not visited[i]:
                if G.has_edge(previous_node, node):
                    distance = G[previous_node][node]['weight']
                    if distance < min_distance:
                        min_distance = distance
                        next_node = node
                        next_index = i

        # Додаємо вершину до маршруту
        if next_node is not None:
            tour.append(next_node)
            visited[next_index] = True
```

```
            current_node = next_node

        # Замикаємо цикл, повертаючись до початкової вершини
        tour.append(tour[0])

        # Обчислюємо довжину маршруту
        tour_length = 0
        for i in range(len(tour)-1):
            tour_length += G[tour[i]][tour[i+1]]['weight']

        return tour, tour_length

    # Знаходження маршруту алгоритмом найближчого сусіда (починаючи з вершини 1)
    nn_path, nn_length = nearest_neighbor_algorithm(G, 1)
    print(f"Маршрут, знайдений алгоритмом найближчого сусіда: {nn_path}. Його вартість = {nn_length}")
```

4. Обґрунтування асимптотики алгоритмів

4.1 Алгоритм повного перебору (груба сила)

Асимптотична складність: O(n!)

Обґрунтування:

Алгоритм перебирає всі можливі перестановки вершин графа.

Кількість таких перестановок для n вершин дорівнює n!.

Для кожної перестановки обчислюється довжина шляху, що вимагає O(n) операцій.

Отже, загальна складність алгоритму становить O(n * n!) = O(n!).

4.2 Алгоритм найближчого сусіда

Асимптотична складність: $O(n^2 \cdot \log n)$

Обґрунтування:

Алгоритм проходить по всіх n вершинах.

На кожному кроці потрібно знайти найближчу невідвідану вершину, що вимагає перегляду до n вершин.

Якщо для знаходження найближчої вершини використовується сортування (наприклад, пріоритетна черга), то це вимагає O(log n) часу.

Отже, загальна складність алгоритму становить $O(n \cdot n \cdot \log n) = O(n^2 \cdot$

log n).

Висновки

У ході виконання практичної роботи були розглянуті принципи роботи жадібних алгоритмів та їх застосування до розв'язання задачі комівояжера. Був заданий зважений неорієнтований граф з 4 вершинами, і для нього реалізовані два методи розв'язання: метод повного перебору та алгоритм найближчого сусіда.

Метод повного перебору забезпечує знаходження точного оптимального розв'язку, але має експоненційну складність $O(n!)$, що обмежує його застосування для задач великої розмірності. Жадібний алгоритм найближчого сусіда має значно кращу складність $O(n^2 \cdot \log n)$, але не гарантує знаходження глобально оптимального розв'язку.

Порівняння результатів розв'язання задачі обома методами для заданого графа дозволяє оцінити ефективність жадібного алгоритму та його відхилення від оптимального розв'язку. Для заданого графа, який має лише 4 вершини, обидва алгоритми працюють швидко, але з збільшенням кількості вершин різниця в ефективності стане більш відчутною.

Жадібні алгоритми є цінним інструментом для отримання наближених розв'язків NP-складних задач, таких як задача комівояжера, особливо у випадках, коли точний розв'язок не є критично важливим, а час виконання обмежений.