МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КРЕМЕНЧУЦЬКИЙ НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ ІМЕНІ МИХАЙЛА
ОСТРОГРАДСЬКОГО

Кафедра комп’ютерної інженерії та електроніки

ЗВІТ З ПРАКТИЧНОЇ РОБОТИ №8

з навчальної дисципліни

«Алгоритми та структури даних»

Тема «Жадібні алгоритми.
Наближене розв’язання екстремальних задач»

Студентка гр. КН-24-1 Бояринцова П. С.
Викладач Сидоренко В. М.

Кременчук 2025

Тема роботи: Жадібні алгоритми. Наближене розв'язання екстремальних задач

1.1 Постановка завдання

Мета роботи: набути практичних навичок застосування деяких жадібних алгоритмів для розв'язання екстремальних задач.

Завдання:

1. Розв'язати задачу комівояжера для графа, заданого варіантом, використовуючи код (рис. 1), наведений вище.
2. Візуалізувати граф (рис. 2).
3. Обґрунтувати асимптотику для алгоритмів.

1.2 Розв'язання задачі

Завдання

3. Заданий зважений граф: [(1,3,8), (1,2,5), (2,3,6), (2,4,4), (3,4,3)]

Розв'язання.

```python
from itertools import permutations


# Константа, яка подає нескінченність (дуже велике число)
INFINITY = pow(10, 20)

# Функція для обчислення довжини шляху
def get_path_length(G, path):
    path_length = 0
    # Проглядаємо усі вершини у шляху
    for i, v1 in enumerate(path):
        # Знаходимо наступну вершину у шляху (цикл замкнутий)
        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 – шлях недійсний, повертаємо нескінченність
            return INFINITY
        # Додаємо вагу ребра до загальної довжини шляху
        path_length += G[v1][v2]["weight"]
    return path_length

# Функція, яка генерує всі можливі перестановки вершин графа, починаючи з заданої вершини
def node_permutations(G, init_node_index):
    # Отримуємо список вершин графа
    nodes = list(G.nodes())
    # Отримуємо початкову вершину за індексом
    init_node = nodes[init_node_index]
    # Видаляємо початкову вершину зі списку
    nodes.remove(init_node)
    # Генеруємо всі можливі перестановки залишкових вершин
    return [[init_node] + list(p) for p in permutations(nodes)]
```

```python
            min_path_length = path_length
    # Повертаємо найкоротший шлях і його довжину
    return min_path, min_path_length
```

```python
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 8))

# бібліотека для роботи з графами
import networkx as nx

# Створюємо порожній неорієнтований граф
G = nx.Graph()
# Додаємо вузли (від 1 до 4 включно)
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)])

pos = nx.spring_layout(G) # позиції для всіх вузлів

# вузли та підписи вузлів
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
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.show()  # вивести графік
```

Рисунок 1 – Розв'язання задачі, використовуючи код (алгоритм грубої сили)

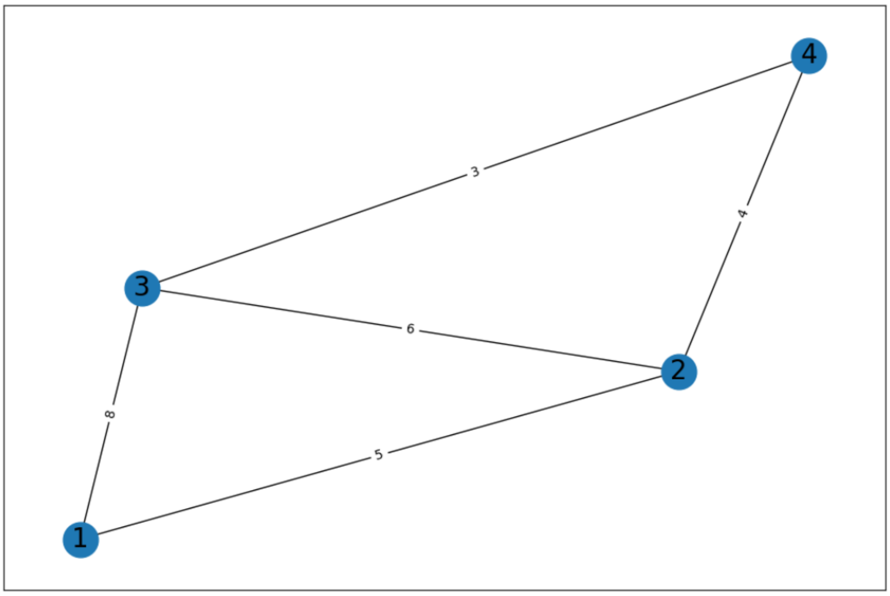

Рисунок 2 – Візуалізація

1.3 Обґрунтування асимптотичної складності двох алгоритмів для задачі комівояжера:

1. Алгоритм грубої сили (Brute Force)
2. Алгоритм найближчого сусіда (Nearest Neighbor)

1. Алгоритм Грубої сили (Brute Force)

Суть:

– Перебираються всі можливі перестановки вершин (шляхів), які починаються з фіксованої початкової вершини.
– Обчислюється довжина кожного з них.
– Вибирається найкоротший.

Кількість можливих маршрутів:

– Для n вершин (з фіксованим стартом) існує $(n-1)!$ маршрутів.

Вартість обчислення довжини одного маршруту:

– Потрібно пройтися по n вершинах – $O(n)$

Повна складність: $O(n \cdot (n-1)!) = O(n!)$

Висновок:

– Алгоритм дуже повільний, але дає точний розв'язок.
– Підходить лише для малих графів ($n \leq 10$).

2. Алгоритм Найближчого сусіда (Nearest Neighbor)

Суть:

- Починаючи з деякої вершини, на кожному кроці вибирається найближча непройдена вершина.
- Повторюється, поки не буде відвідано всі вершини.
- Потім повертається у початкову.

Кількість дій:

- У першій ітерації — перевіряється n−1 сусідів,
- У другій — n−2,
- Останній крок — 1 сусід.

Сума: (n−1)+(n−2)+⋯+1= $\frac{n(n-1)}{2}$

Повна складність: $O(n^2)$

Висновок:

- Набагато швидший, ніж Brute Force.
- Дає неточний (наближений) результат, але часто хороший на практиці.
- Підходить для великих графів, де потрібна швидкість.

Порівняльна таблиця:

Алгоритм	**Точність**	**Асимптотична складність**	**Підходить для**
Грубої сили (Brute Force)	Висока	$O(n!)$	Малих графів
Найближчого сусіда	Середня	$O(n^2)$	Великих графів

1.4 Відповіді на контрольні питання

1. Що таке жадібний алгоритм?

Жадібний алгоритм — це метод розв'язання задачі, який на кожному кроці приймає локально оптимальне рішення, сподіваючись, що в результаті отримає глобально оптимальний результат.

2. Які головні принципи роботи жадібних алгоритмів?
- На кожному кроці вибирається найкраще (найоптимальніше) локальне рішення.
- Прийняте рішення не змінюється у подальших кроках (відсутність повернень).
- Проблема має властивість «жадібної вибірки» — локальний вибір веде до глобального оптимуму.
3. Яка головна відмінність між жадібними алгоритмами та динамічним програмуванням?
- Жадібні алгоритми приймають рішення один раз і не повертаються назад.
- Динамічне програмування розглядає всі можливі підзадачі і запам'ятовує їх рішення, що дозволяє знаходити глобально оптимальний результат навіть при відсутності властивості жадібної вибірки.
4. Наведіть приклади задач, які можна розв'язати за допомогою жадібних алгоритмів.
- алгоритм Дейкстри знаходження найкоротшого шляху від однієї вершини до всіх інших у зваженому ациклічному графі
- алгоритм Краскала, який знаходить мінімальне кістякове дерево у зваженому графі
- алгоритм оптимального кодування Гафмена
- задача про рюкзак.
5. Які можуть бути обмеження у використанні жадібних алгоритмів для розв'язання екстремальних задач?
- Жадібні алгоритми не гарантують глобального оптимуму для усіх задач.
- Вони можуть працювати некоректно, якщо задача не має властивості жадібного вибору.
- Підходять лише для задач, де локальний оптимум веде до глобального.
6. Чому жадібні алгоритми часто використовуються для наближеного розв'язання екстремальних задач?
- Вони прості і швидкі у реалізації.

- Дають хороші приблизні рішення, якщо точний розв’язок занадто складний або неможливий за прийнятний час.
- Дозволяють отримати результат з прийнятною точністю у великих або складних задачах.