Практична робота № 5

Тема. Графи. Ациклічні графи

Мета: набути практичних навичок розв’язання задач топографічного сортування та оцінювання їх асимптотичної складності.

Постановка задачі.

Виконати індивідуальне завдання. Завдання полягає у розв’язанні задачі, яку потрібно вибрати зі списку, наведеного нижче. Номер варіанта відповідає номеру студента у списку групи. У разі, якщо було досягнуто кінця списку задач, потрібно циклічно повернутися на його початок.

Завдання.

Ациклічний граф може бути представлений у вигляді списку суміжності:

- Вершини: {1, 2, 3, 4, 5, 6, 7}
- Ребра: {(1, 2), (1, 3), (1, 6), (3, 7), (1, 4), (4, 5)}

Графічно це можна уявити так:

```
   1

  / | \ \

 2 3 6 4

   |     |

   7    5
```

Алгоритм топологічного сортування за допомогою DFS

Для топологічного сортування графа за допомогою алгоритму DFS (глибини в перший пошук) ми виконуємо такі кроки:

1. **Ініціалізуємо всі вершини як неперевірені (unvisited).**
2. **Виконуємо DFS для кожної неперевіреної вершини, додаючи вершини до стека після відвідування всіх їхніх сусідів.**
3. **Витягуємо вершини зі стека для отримання топологічно відсортованого порядку.**

Псевдокод

4. Створити список для відмічення відвіданих вершин.
5. Створити стек для зберігання топологічного порядку.

6. Написати функцію DFS, яка:
   - Відмічає поточну вершину як відвідану.
   - Рекурсивно викликає DFS для кожної сусідньої неперевіреної вершини.
   - Додає поточну вершину до стека після відвідування всіх сусідів.
7. Виконати DFS для кожної вершини, яка ще не була відвідана.
8. Витягнути вершини зі стека для отримання топологічного порядку.

Реалізація на Python

```
from collections import defaultdict



# Граф, представлений у вигляді списку суміжності

graph = defaultdict(list)

graph[1] = [2, 3, 6, 4]

graph[2] = []

graph[3] = [7]

graph[4] = [5]

graph[5] = []

graph[6] = []

graph[7] = []



# Функція для виконання DFS

def dfs(v, visited, stack):

    visited[v] = True

    for neighbour in graph[v]:

        if not visited[neighbour]:

            dfs(neighbour, visited, stack)

    stack.append(v)
```

```
# Функція для топологічного сортування

def topological_sort(graph):

    visited = {v: False for v in graph}



    stack = []



    for v in graph:

        if not visited[v]:

            dfs(v, visited, stack)



    stack.reverse()

    return stack



# Виконуємо топологічне сортування



topological_order = topological_sort(graph)

print("Топологічно відсортований порядок:", topological_order)
```

Виконання алгоритму

9. **Починаємо з вершини 1**:
   - Відвідуємо сусідів: 2, 3, 6, 4.
   - Виконуємо рекурсивний DFS для кожного з них.
10. **DFS для вершини 2**:
    - Вершина 2 не має сусідів.
    - Додаємо 2 до стека.
11. **DFS для вершини 3**:
    - Відвідуємо сусіда 7.
    - Виконуємо DFS для вершини 7.

12. **DFS для вершини 7**:
    - Вершина 7 не має сусідів.
    - Додаємо 7 до стека.
    - Додаємо 3 до стека.
13. **DFS для вершини 6**:
    - Вершина 6 не має сусідів.
    - Додаємо 6 до стека.
14. **DFS для вершини 4**:
    - Відвідуємо сусіда 5.
    - Виконуємо DFS для вершини 5.
15. **DFS для вершини 5**:
    - Вершина 5 не має сусідів.
    - Додаємо 5 до стека.
    - Додаємо 4 до стека.
16. **Додаємо вершину 1 до стека після відвідування всіх сусідів**.

Порядок у стеці буде: [1, 4, 5, 6, 3, 7, 2]

Топологічно відсортований порядок: [1, 4, 5, 6, 3, 7, 2]

Контрольні питання

17. **Переваги і недоліки алгоритму Кана порівняно з алгоритмом DFS для топологічного сортування графа:**
    - **Алгоритм Кана**:
        - *Переваги*: Ефективний для великих графів, оскільки використовує ітераційний підхід та знаходить топологічне сортування за складність $O(V+E)O(V+E)$.
        - *Недоліки*: Складніший для реалізації порівняно з DFS; не працює з графами, які містять цикли.
    - **DFS**:
        - *Переваги*: Простота реалізації; працює з графами, які містять цикли.
        - *Недоліки*: Може бути менш ефективним для великих графів через рекурсивний характер та складність $O(V+E)O(V+E)$ у гіршому випадку.
18. **Складність часу і пам’яті для кожного з алгоритмів у найгіршому і найкращому випадках:**
    - **Алгоритм Кана**:
        - *Час*: $O(V+E)O(V+E)$ у найгіршому випадку, де VV - кількість вершин, EE - кількість ребер.
        - *Пам'ять*: $O(V)O(V)$, оскільки потрібно зберігати список вершин, які не мають входящих ребер.
    - **DFS**:
        - *Час*: $O(V+E)O(V+E)$ у найгіршому випадку.
        - *Пам'ять*: $O(V)O(V)$ у найгіршому випадку для зберігання стеку відвіданих вершин.

19. **Застосування алгоритму Кана до графів з вагами на ребрах та порівняння з DFS:**
    - Алгоритм Кана застосовується лише до ациклічних графів, тому для графів з вагами на ребрах потрібно перевіряти відсутність циклів перед застосуванням алгоритму. DFS може бути застосований до будь-якого типу графа, включаючи ті, що містять цикли.
    - Порівняно з DFS, алгоритм Кана потребує додаткового перевірки графу на ацикличність, що може бути витратним за ресурсами.
20. **Вплив структури графа на швидкість роботи кожного з цих алгоритмів:**
    - **Алгоритм Кана**: Працює ефективно для ациклічних графів, де кожна вершина має визначений порядок виконання.
    - **DFS**: Може бути ефективним для графів будь-якої структури, але може бути менш ефективним для графів зі складною структурою через рекурсивний характер.
21. **Обмеження використання кожного алгоритму для певних типів графів або завдань:**
    - **Алгоритм Кана**: Застосовується лише до ациклічних графів.
    - **DFS**: Може бути використаний для будь-якого типу графа, включаючи графи з циклами.
22. **Варіанти оптимізації для кожного алгоритму з метою поліпшення його продуктивності:**
    - **Алгоритм Кана**: Використання структур даних, таких як стеки або черги, для зменшення часу та пам'яті при перевірці на ациклічність.
    - **DFS**: Використання ітеративного підходу замість рекурсії для зменшення витрат пам'яті та уникнення перевищення стеку.