Лабораторна робота №2

Виконали Депутат Антон і Михалищук Назар

Наш код реалізує кілька функцій для роботи з графами у різних формах представленості: матриця суміжності, матриця інцидентності та список суміжності. Код містить функції для пошуку в глибину (DFS), пошук в ширину (BFS) та обчислення радіусу графа. Давайте розглянемо кожну функцію детальніше.

```python
def adjacency_matrix_transformer(lst: list[list]) -> dict[int, list[int]]:
    """
    Transforms matrix to dict
    >>> adjacency_matrix_transformer([[0, 1, 1], [1, 0, 1], [1, 1, 0]])
    {0: [1, 2], 1: [0, 2], 2: [0, 1]}
    """
    adjacency = {}
    mid = []
    q = 0
    for i, k in enumerate(lst):
        for e, _ in enumerate(k):
            if k[e] == 1:
                if adjacency.get(i):
                    mid = adjacency.get(i)
                    adjacency.pop(i)

                mid.append(e)
                adjacency.setdefault(i, mid)
                mid = []
                q += 1
        if q == 0:
            adjacency.setdefault(i, mid)
        q = 0
    return adjacency
```

1. adjacency_matrix_transformer(lst: list[list]) -> dict[int, list[int]]

Опис: Ця функція перетворює матрицю суміжності у словник списків суміжності.

Вхід: Матриця суміжності, де lst[i][j] = 1 означає наявність ребра між вершинами i та j.
Вихід: Словник, де ключем є вершина, а значенням — список її сусідів.

Алгоритм:

Пройшовшись по всіх елементах матриці, для кожного елемента 1 додається вершина до списку сусідів відповідної вершини.

```python
def read_incidence_matrix(filename: str) -> list[list]:
    """
    :param str filename: path to file
    :returns list[list]: the incidence matrix of a given graph
    """
    edge = []
    final = []
    skip = 0
    q = 0
    with open(filename, "r", encoding="utf") as file:
        while True:
            edgecontent = file.readline().replace("\n", "")
            if skip == 0:
                skip += 1
                continue
            if edgecontent == "}":
                break
            mid = edgecontent.split()
            edge.append((int(mid[0]), int(mid[-1][0])))
    mid = []
    for i in edge:
        for e in i:
            if q < e:
                q = e
    for _ in range(len(edge)):
        mid.append(0)
    for _ in range(q + 1):
        final.append(mid.copy())
    for i, e in enumerate(edge):
        for k in e:
            final[k][i] = 1
    return final
```

2. read_incidence_matrix(filename: str) -> list[list]

Опис: Ця функція читає матрицю інцидентності з файлу.

Вхід: Ім'я файлу, що містить опис графа.
Вихід: Матриця інцидентності графа.

Алгоритм:

Функція зчитує файл, створює список ребер, а потім будує матрицю інцидентності, де кожен стовпець представляє ребро, а кожен рядок — вершину. Якщо вершина інцидентна ребру, то на відповідній позиції буде стояти 1.

```python
def read_adjacency_matrix(filename: str) -> list[list]:
    """
    :param str filename: path to file
    :returns list[list]: the adjacency matrix of a given graph
    """
    edge = []
    final = []
    skip = 0
    q = 0
    with open(filename, "r", encoding="utf") as file:
        while True:
            edgecontent = file.readline().replace("\n", "")
            if skip == 0:
                skip += 1
                continue
            if edgecontent == "}":
                break
            mid = edgecontent.split()
            edge.append((int(mid[0]), int(mid[-1][0])))
    mid = []
    for i in edge:
        for e in i:
            if q < e:
                q = e
    for _ in range(q + 1):
        mid.append(0)
    for _ in range(q + 1):
        final.append(mid.copy())
    for i in edge:
        final[i[0]][i[1]] = 1
    return final
```

3. read_adjacency_matrix(filename: str) -> list[list]

Опис: Ця функція зчитує матрицю суміжності з файлу.

Вхід: Ім'я файлу, що містить опис графа.
Вихід: Матриця суміжності графа.

Алгоритм:

Читає файл, збирає ребра і на основі них створює матрицю суміжності.

```python
def read_adjacency_dict(filename: str) -> dict[int, list[int]]:
    """
    :param str filename: path to file
    :returns dict: the adjacency dict of a given graph
    """
    edge = []
    final = {}
    skip = 0
    with open(filename, "r", encoding="utf") as file:
        while True:
            edgecontent = file.readline().replace("\n", "")
            if skip == 0:
                skip += 1
                continue
            if edgecontent == "}":
                break
            mid = edgecontent.split()
            edge.append((mid[0], mid[-1][0]))
    mid = []
    for i in edge:
        if final.get(int(i[0])):
            mid = final.get(int(i[0]))
            final.pop(int(i[0]))
        mid.append(int(i[1]))
        final.setdefault(int(i[0]), mid)
        mid = []
    return final
```

4. read_adjacency_dict(filename: str) -> dict[int, list[int]]

Опис: Ця функція зчитує словник суміжності з файлу.

Вхід: Ім'я файлу, що містить опис графа.
Вихід: Словник суміжності графа.

Алгоритм:

Читає файл і будує словник, де кожному ключу (вершині) відповідає список суміжних до неї вершин.

```python
def iterative_adjacency_dict_dfs(graph: dict[int, list[int]], start: int) -> list[int]:
    """
    :param list[list] graph: the adjacency list of a given graph
    :param int start: start vertex of search
    :returns list[int]: the dfs traversal of the graph
    >>> iterative_adjacency_dict_dfs({0: [1, 2], 1: [0, 2], 2: [0, 1]}, 0)
    [0, 1, 2]
    >>> iterative_adjacency_dict_dfs({0: [1, 2], 1: [0, 2, 3], 2: [0, 1], 3: []}, 0)
    [0, 1, 2, 3]
    """
    visited = set()
    stack = [start]
    result = []
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            result.append(node)
        stack.extend(
            neighbor for neighbor in graph[node][::-1] if neighbor not in visited
        )
    return result


def iterative_adjacency_matrix_dfs(graph: list[list], start: int) -> list[int]:
    """
    :param dict graph: the adjacency matrix of a given graph
    :param int start: start vertex of search
    :returns list[int]: the dfs traversal of the graph
    >>> iterative_adjacency_matrix_dfs([[0, 1, 1], [1, 0, 1], [1, 1, 0]], 0)
    [0, 1, 2]
    >>> iterative_adjacency_matrix_dfs([[0, 1, 1, 0], [1, 0, 1, 1], [1, 1, 0, 0], [0, 0, 0, 0]], 0)
    [0, 1, 2, 3]
    """
    graph = adjacency_matrix_transformer(graph)
    return iterative_adjacency_dict_dfs(graph, start)
```

5. iterative_adjacency_dict_dfs(graph: dict[int, list[int]], start: int) -> list[int]

Опис: Ця функція здійснює ітеративний пошук в глибину (DFS) для графа, представленого як словник суміжності.

Вхід: Граф у вигляді словника суміжності, стартова вершина.
Вихід: Список вершин у порядку відвідування.

Алгоритм:

Використовує стек для обробки вершин. Пройшовшись по всіх сусідах поточної вершини, додає їх у стек, доки не буде оброблено всі вершини.

6. iterative_adjacency_matrix_dfs(graph: list[list], start: int) -> list[int]

Опис: Ця функція виконує ітеративний DFS для графа, представленого матрицею суміжності.

Вхід: Граф у вигляді матриці суміжності, стартова вершина.
Вихід: Список вершин у порядку відвідування.

Алгоритм:

Спочатку перетворює матрицю суміжності в словник суміжності, а потім викликає iterative_adjacency_dict_dfs.

```python
def recursive_adjacency_dict_dfs(
    graph: dict[int, list[int]], start: int, visited: set = None
) -> list[int]:
    """
    :param list[list] graph: the adjacency list of a given graph
    :param int start: start vertex of search
    :returns list[int]: the dfs traversal of the graph
    >>> recursive_adjacency_dict_dfs({0: [1, 2], 1: [0, 2], 2: [0, 1]}, 0)
    [0, 1, 2]
    >>> recursive_adjacency_dict_dfs({0: [1, 2], 1: [0, 2, 3], 2: [0, 1], 3: []}, 0)
    [0, 1, 2, 3]
    """
    if visited is None:
        visited = []
    if start not in visited:
        visited.append(start)
        for next_el in graph.get(start):
            recursive_adjacency_dict_dfs(graph, next_el, visited)
    return visited


def recursive_adjacency_matrix_dfs(
    graph: list[list[int]], start: int, visited: set = None
) -> list[int]:
    """
    :param dict graph: the adjacency matrix of a given graph
    :param int start: start vertex of search
    :returns list[int]: the dfs traversal of the graph
    >>> recursive_adjacency_matrix_dfs([[0, 1, 1], [1, 0, 1], [1, 1, 0]], 0)
    [0, 1, 2]
    >>> recursive_adjacency_matrix_dfs([[0, 1, 1, 0], [1, 0, 1, 1], [1, 1, 0, 0], [0, 0, 0, 0]], 0)
    [0, 1, 2, 3]
    """
    graph = adjacency_matrix_transformer(graph)
    return recursive_adjacency_dict_dfs(graph, start, visited)
```

7. recursive_adjacency_dict_dfs(graph: dict[int, list[int]], start: int, visited: set = None) -> list[int]

Опис: Ця функція реалізує рекурсивний пошук в глибину (DFS) для графа, представленого як словник суміжності.

Вхід: Граф у вигляді словника суміжності, стартова вершина.
Вихід: Список вершин у порядку відвідування.

Алгоритм:

Використовує рекурсію для відвідування всіх сусідів поточної вершини.

8. recursive_adjacency_matrix_dfs(graph: list[list], start: int, visited: set = None) -> list[int]

Опис: Ця функція виконує рекурсивний DFS для графа, представленого матрицею суміжності.

Вхід: Граф у вигляді матриці суміжності, стартова вершина.
Вихід: Список вершин у порядку відвідування.

Алгоритм:

Перетворює матрицю суміжності в словник суміжності і викликає recursive_adjacency_dict_dfs.

```python
def iterative_adjacency_dict_bfs(graph: dict[int, list[int]], start: int) -> list[int]:
    :param list[list] graph: the adjacency list of a given graph
    :param int start: start vertex of search
    :returns list[int]: the bfs traversal of the graph
    >>> iterative_adjacency_dict_bfs({0: [1, 2], 1: [0, 2], 2: [0, 1]}, 0)
    [0, 1, 2]
    >>> iterative_adjacency_dict_bfs({0: [1, 2], 1: [0, 2, 3], 2: [0, 1], 3: []}, 0)
    [0, 1, 2, 3]
    """
    visited = set()
    queue = [start]
    result = []
    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.add(node)
            result.append(node)
            queue.extend(
                neighbor for neighbor in graph.get(node, []) if neighbor not in visited
            )
    return result


def iterative_adjacency_matrix_bfs(graph: list[list[int]], start: int) -> list[int]:
    """
    :param dict graph: the adjacency matrix of a given graph
    :param int start: start vertex of search
    :returns list[int]: the bfs traversal of the graph
    >>> iterative_adjacency_matrix_bfs([[0, 1, 1], [1, 0, 1], [1, 1, 0]], 0)
    [0, 1, 2]
    >>> iterative_adjacency_matrix_bfs([[0, 1, 1, 0], [1, 0, 1, 1], [1, 1, 0, 0], [0, 0, 0, 0]], 0)
    [0, 1, 2, 3]
    """
    graph = adjacency_matrix_transformer(graph)
    return iterative_adjacency_dict_bfs(graph, start)
```

9. iterative_adjacency_dict_bfs(graph: dict[int, list[int]], start: int) -> list[int]

Опис: Ця функція здійснює ітеративний пошук в ширину (BFS) для графа, представленого як словник суміжності.

Вхід: Граф у вигляді словника суміжності, стартова вершина.
Вихід: Список вершин у порядку відвідування.

Алгоритм:

Використовує чергу для обробки вершин. Пройшовшись по всіх сусідах поточної вершини, додає їх у чергу.

10. iterative_adjacency_matrix_bfs(graph: list[list[int]], start: int) -> list[int]

Опис: Ця функція виконує ітеративний BFS для графа, представленого матрицею суміжності.

Вхід: Граф у вигляді матриці суміжності, стартова вершина.
Вихід: Список вершин у порядку відвідування.

Алгоритм:

Спочатку перетворює матрицю суміжності в словник суміжності, а потім викликає iterative_adjacency_dict_bfs.

```python
def adjacency_matrix_radius(graph: list[list]) -> int:
    >>> adjacency_matrix_radius([[0, 1, 1], [1, 0, 1], [1, 1, 0], [0, 1, 0]])
    2
    """
    graph = adjacency_matrix_transformer(graph)
    return adjacency_dict_radius(graph)


def adjacency_dict_radius(graph):
    """
    :param dict graph: the adjacency list of a given graph
    :returns int: the radius of the graph
    >>> adjacency_dict_radius({0: [1, 2], 1: [0, 2], 2: [0, 1]})
    1
    >>> adjacency_dict_radius({0: [1, 2], 1: [0, 2], 2: [0, 1], 3: [1]})
    2
    """

    def iterative_adjacency_dict_bfs_with_distances(graph, start):
        distances = {start: 0}
        queue = [start]
        while queue:
            node = queue.pop(0)
            for neighbor in graph.get(node, []):
                if neighbor not in distances:
                    distances[neighbor] = distances[node] + 1
                    queue.append(neighbor)
        return distances

    eccentricities = []
    for node in graph:
        distances = iterative_adjacency_dict_bfs_with_distances(graph, node)
        if len(distances) != len(graph):
            distances.update({v: float("inf") for v in graph if v not in distances})
        eccentricities.append(max(distances.values()))
    return min(eccentricities)
```

11. adjacency_matrix_radius(graph: list[list]) -> int

Опис: Ця функція обчислює радіус графа, представленого як матриця суміжності.

Вхід: Граф у вигляді матриці суміжності.
Вихід: Радіус графа.

Алгоритм:

Перетворює матрицю суміжності в словник суміжності і викликає функцію adjacency_dict_radius.

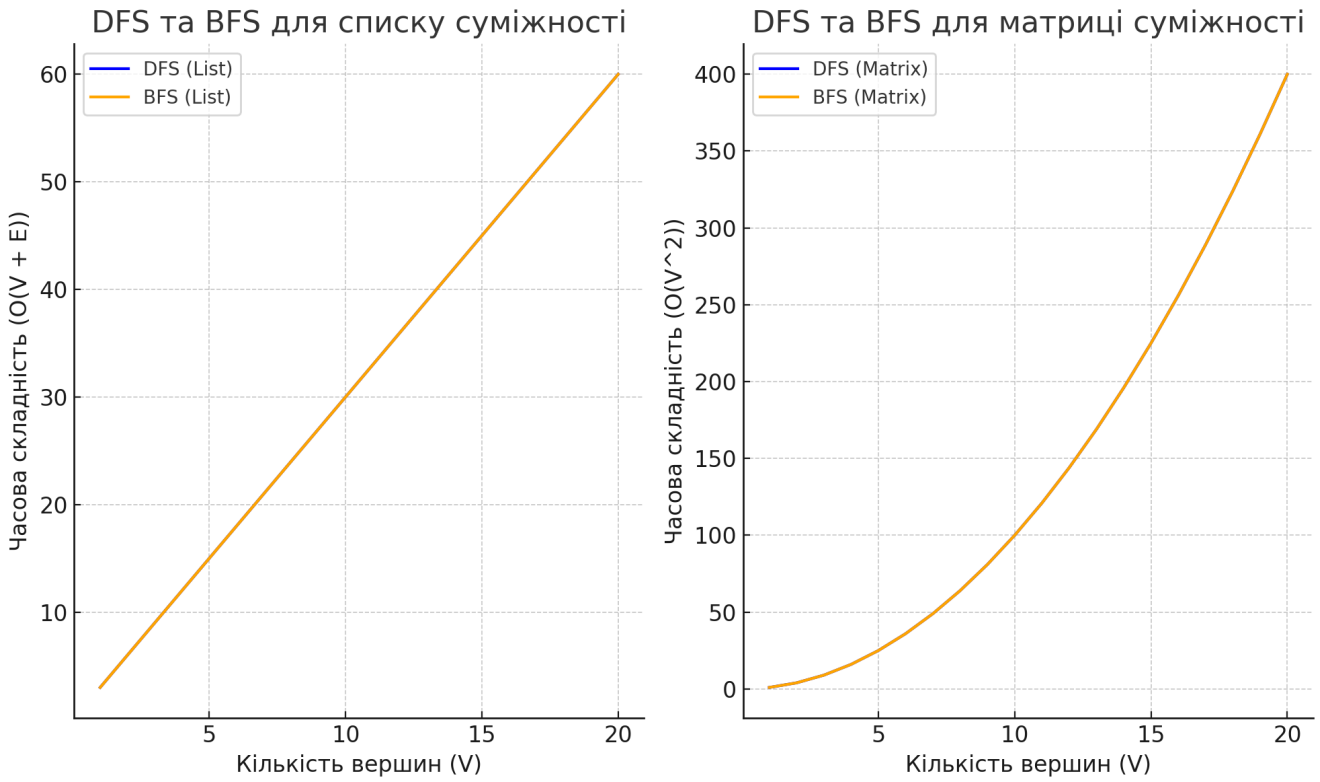
12. adjacency_dict_radius(graph)

Опис: Ця функція обчислює радіус графа, представленого як словник суміжності.

Вхід: Граф у вигляді словника суміжності.
Вихід: Радіус графа.

Алгоритм:

Для кожної вершини знаходиться відстань до всіх інших вершин (це робиться за допомогою BFS), потім обчислюється ексцентриситет для кожної вершини, і радіус визначається як мінімум з цих значень.



Графіки ефективності

Для аналізу ефективності ми можемо побудувати графіки складності алгоритмів DFS та BFS для кожної представленості графа.

Аналіз складності:

DFS та BFS для списку суміжності:
Часова складність: O(V + E), де V — кількість вершин, а E — кількість ребер.
Просторова складність: O(V), оскільки використовуються додаткові структури даних для відвідуваних вершин.
DFS та BFS для матриці суміжності:
Часова складність: O(V^2), оскільки для кожної вершини потрібно перевіряти всі її сусіди.
Просторова складність: O(V^2) для збереження матриці суміжності.