Графи etc

Виконала команда 8

Оксана Савчук, Анастасія Пелех, Микола Висоцький, Северин Шикула, Дарина Кузишин

1. Зчитування графу

```python
from collections import defaultdict

def read_graph(filename: str) -> dict[int, list[int]]:
    """
    Function reads graph from .csv file.
    File have to contain edges on each line in format [source, destination].
    >>> read_graph('gr.csv', False)
    """
    graph = defaultdict(lambda: [])

    try:
        with open(filename, encoding="utf-8") as src_file:
            for line in src_file.readlines():
                pair = tuple(map(int, line.strip().split(",")))

                graph[pair[0]].append(pair[1])
                if pair[1] not in graph:
                    graph[pair[1]] = []
    except FileNotFoundError:
        print(f"'{filename}' file is not found.")
        return {}

    return dict(graph)
```

Зчитуємо всі ребра графу. Записуємо їх у словник.
Вертаємо вхідний граф у вигляді словника.

2. Гамільтоновий цикл

```
if not graph:
    return []

vertices = list(graph.keys())
cycle = [vertices[0]]
ban = set()

while len(cycle) != (len(vertices) + 1) and cycle:
    vertex = cycle[-1]
    adj = {ver for ver in graph[vertex] if not(ver in cycle[1:] or tuple(cycle + [ver]) in ban)}

    if len(cycle) < len(graph.keys()):
        adj = {ver for ver in adj if ver != cycle[0]}

    if not adj:
        vertex = cycle.pop()
        if cycle:
            ban.add(tuple(cycle + [vertex]))
            continue
        return None

    cycle.append(next(iter(adj)))

return cycle
```

Виконано за допомогою бектрекінгу
Пробуєм заповнити цикл, поки він не стане довжиною у всі вершини +1.
Для цього ми беремо останню вершину і відбираємо вершини, до яких воно може піти - шлях не має бути в бані і вершина не має бути використана.
Якщо прилягаючих немає - вертаємось на крок назад і поки є цикл добавляємо цикл в бан.
Якщо прилягаючих немає і цикл вже нуль - вертаємо None - тому що циклу немає.

3. Ейлерів цикл

```
def euler_cycle(graph: dict[int, list[int]]) -> list[int]:
    """
    Function takes graph and returns Euler's cycle if exists.
    If not returns None.
    >>> euler_cycle({0: [1], 1: [2], 2: [3], 3: [0]})
    [0, 1, 2, 3, 0]
    """
    if not is_strongly_connected(graph, consider_isolated_vertices=False):
        return None

    if not has_euler_cycle(graph):
        return None

    graph = deepcopy(graph)
    circuit = []

    current_vertex = list(graph.keys())[0]
    current_path = [current_vertex]

    while current_path:
        if graph[current_vertex]:
            current_vertex = graph[current_vertex].pop()
            current_path.append(current_vertex)
        else:
            circuit.append(current_path.pop())

    circuit.reverse()
    return circuit
```

Спочатку перевіряємо чи граф є сильнозв’язний, перевіряємо чи степені входу дорівнюють степеням виходу. Далі використовуємо Hierholzer’s alghorithm для знаходження циклу. Проходимось ланцюгами, як ДФС. Шлях може закритися і не містити всіх вершин.Записуємо вершини на якій зупинились.

Тоді вертаємось до наступної можливої для відвідування вершини.

```
def dfs(graph: dict[int, list[int]], start_in: int = 0, terminate_in: int = -1) -> list:
    """
    Function computed Depth First Search on 'graph'.
    >>> dfs({0: [2, 5, 7], 1: [2, 6, 7], 2: [0, 1, 4, 5, 6, 7],\
    3: [6, 7], 4: [2, 5, 7], 5: [0, 2, 4, 7],\
    6: [1, 2, 3, 7], 7: [0, 1, 2, 3, 4, 5, 6]})
    [0, 2, 1, 6, 3, 7, 4, 5]
    """
    if start_in not in graph.keys():
        return None

    stack = [start_in]
    result = [start_in]

    while stack:
        for adj_vertex in graph[stack[-1]]:
            if adj_vertex not in result:
                result.append(adj_vertex)
                stack.append(adj_vertex)

                if terminate_in == adj_vertex:
                    return result

                break
        else:
            stack.pop()

    return result
```

Реалізація пошуку вглиб ітеративно.

```python
def get_transposed(graph: dict[int, list[int]]):
    """
    Function returns transpised graph.
    >>> get_transposed({0: [1], 1: [2], 2: [3], 3: [0], 4: []})
    {0: [3], 1: [0], 2: [1], 3: [2], 4: []}
    """
    result = defaultdict(list)

    for key, value in graph.items():
        if not result.get(key, []):
            result[key] = []
        for ver in value:
            result[ver].append(key)

    return dict(result)
```

Функція приймає граф і повертає транспонований граф(дуги протилежної орієнтації)

функція перевіряє чи граф є сильнозв’язний. Алгоритм- ДФС Косараджу

```python
def is_strongly_connected(graph: dict[int, list[int]]):
    """
    Function checks is graph strongly connected or not.
    WE DO NOT CONSIDER ISOLATED VERTICES.
    >>> is_strongly_connected({0: [1], 1: [2], 2: [3], 3: [0], 4: []})
    True
    >>> is_strongly_connected({0: [1], 1: [2], 2: [3], 3: [0], 4: []})
    True
    >>> is_strongly_connected({0: [1], 1: [2], 2: [0], 10: [11], 11: [12], 12: [10]})
    False
    """
    transposed_graph = get_transposed(graph)
    isolated_vertices = {key for key in graph if not graph[key] and not transposed_graph[key]}
    vertices = set(graph.keys()) - isolated_vertices

    direct_dfs = dfs(graph, next(iter(vertices)))
    reverse_dfs = dfs(transposed_graph, next(iter(vertices)))

    return set(direct_dfs) == vertices and vertices == set(reverse_dfs)
```

4. Дводольність графа

```
def is_bipartite(graph: dict[int, list[int]]) -> bool:
    """
    Function checks if graph is bipartite.
    >>> is_bipartite({1: [2, 3], 4: [2, 3], 2: [], 3: []})
    True
    >>> is_bipartite({1: [2], 2: [3], 3:[4], 4: [5], 5: [6], 6: [1]})
    True
    >>> is_bipartite({0: [3], 1: [2, 3], 2: [1, 3, 5, 6, 7],\
    3:[0, 1, 2], 4: [5, 6], 5: [2, 4, 6], 6: [2, 4, 5], 7: [2]})
    False
    """
    graph = deepcopy(graph)
    for key, value in graph.items():
        for adj in value:
            graph[adj].append(key)

    initial_vertex = next(iter(graph.keys()))
    queue = [initial_vertex]
    coloring = {0: {initial_vertex}, 1: set()}

    while queue:
        vertex = queue.pop(0)
        current_color = vertex in coloring[1]

        for adj_vertex in graph[vertex]:
            if adj_vertex in coloring[current_color]:
                return False
            if adj_vertex in coloring[not current_color]:
                continue

            queue.append(adj_vertex)
            coloring[not current_color].add(adj_vertex)

    return True
```

Реалізовуємо це за допомогою BFS. Початкову вершину замальовуємо одним кольором, всі сусідні малюємо в інший колір. Якщо вершина має сусідню вершину замальовану в її колір, то вертаємо False. Якщо ж такого за обхід не сталось, то вертаємо True, бо ми маємо розмалювання на два кольори.

5. Ізоморфізм графів

```python
def is_isomorphic(graph1: dict[int, list[int]], graph2: dict[int, list[int]]) -> bool:
    """
    Function takes two graphs and determines
    are they isomorphic of not.
    >>> is_isomorphic({1: [3, 5], 2: [7], 5: [2, 3], 5: [], 7: [], 3: []},\
    {5: [], 7: [], 2: [3, 5], 1: [7], 3: [1, 5]})
    False
    >>> is_isomorphic({'a': ['c', 'b'], 'b': ['d'], 'c': [], 'd': []\
    }, {0: [1, 2], 1: [3], 2: [], 3: []})
    True
    >>> is_isomorphic({'a': ['b', 'd'], 'd': ['b'], 'b': ['c'], 'c': ['a', 'd']},\
    {'a': ['b', 'd'], 'd': ['b', 'c'], 'b': ['c'], 'c': ['a']})
    True
    >>> is_isomorphic({'a': ['b'], 'd': [], 'b': ['d'], 'c': ['a', 'b']},\
    {'a': ['c', 'd'], 'd': ['b'], 'b': [], 'c': ['b']})
    False
    """
    if len(graph1.keys()) != len(graph2.keys()):
        return False

    original = {(frm, to_ver) for frm, to in graph1.items() for to_ver in to}

    for permitation in permutations(graph1.keys()):
        bjection = {key: value for key, value in zip(graph2.keys(), permitation)}
        mutated = {(bjection[frm], bjection[to_ver]) for frm, to in graph2.items() for to_ver in to}

        if mutated == original:
            return True

    return False
```

Генеруєм всі можливі бієкції і накладаємо на другий граф, якщо бієкція співпадає з першим графом вертаємо True, якщо жодна не співпала, то False. Додатково перевіряєм чи кількість вершин однакова.