Algorytmy i struktury danych (studia stacjonarne)

Dr Anna Muranova
Semestr zimowy 2025/2026, UWM w Olsztynie

Ćwiczenie 9 (Graf nieskierowany) 10.12.2025

Zadanie 9.1. Graf to struktura danych przeznaczona do badania zależności pomiędzy obiektami. Graf składa się z węzłów (wierzchołków) połączonych krawędziami, w taki sposób że każda z nich łączy dwa wierzchołki. Każdy z wierzchołków grafu reprezentuje pewną wartość. Każda krawędź grafu może być oznaczona pewną wartością numeryczną (np. koszt podróży z wierzchołka A do wierzchołka B), która jest wagą tej krawędzi. Takie grafy nazywamy nieskierowanymi

Krawędź może również mieć dodatkowo nadany kierunek, a graf składający się z takich krawędzi jest grafem skierowanym.

Przygotować implementację grafa wg następujących wymagań:

(A) Klasa przechowująca węzły grafu:

```
from typing import Any

class Vertex:
    data: Any
```

gdzie data oznacza wartość przechowywaną w wierzchołku, domyślnie None. Stwórz konstruktor. Metoda `__str__` zwraca data w postaci napisu.

(B) Klasa przechowująca krawędzie grafu:

```
class Edge:
    vertices: set[Vertex]
    weight: float
```

gdzie `vertices` będzie zbiorem rozmiaru 2. Waga domyślnie będzie `1.0`. Stwórz konstruktor. Metoda `__str__` zwraca data z wierzchołków ze znakiem – pomiędzy nimi.

(C) Klasa przechowująca strukturę grafu:

```
class Graph:
    vertices: list[Vertex],
    edges: set[Edge]
```

W klasie należy umieścić następujące metody:

(a) konstruktor `__init__`, który stworze pusty graf

(b) metoda `__str__`, zwraca listą wartości wierzchołków, oraz zbiór krawędzi w postaci napisów

(c) funkcja `add_vertex(value)` dodaje wierzchołek do grafa

(d) funkcja `size()` zwraca ilość wierzchołków w grafie

(e) funkcja `edge_size()` zwraca ilość krawędziowi w grafie

(f) funkcja `add_edge(vertex1, vertex2, weight = 1.0)` dodaje krawędź do grafa

(g) funkcja `is_edge(vertex1, vertex2)` sprawdza, czy jest krawędź pomiędzy podanymi wierzchołkami

(h) funkcja `cone( value)`, która doda wierzchołek do grafa i połączy go ze wszystkimi wierzchołkami grafa

(i) funkcja `complete_graph(*values)`, która stworzy pełny graf (wszystkie możliwe krawędzi) z wierzchołków z podanymi wartościami

(j) funkcja `adjacency_matrix()`, zwróci macierz sąsiedztwa grafu.

(k) funkcja `neighbours_dict()`, zwróci słownik, gdzie kluczami będą wszystkie wierzchołki na liście, a danymi – lista wszystkich wierzchołków, sąsiednich z kluczem (zachowaj porządek!).

(l) funkcja `neighbours_dict_data()`, przekształcicie w poprzednim slowniku wierzchołki na dany w nich.

(m) `breadth_ first_traversal(visit)` (przechodzenie wszerz):

- odwiedzanie rozpoczynamy od pierwszego wierzchołka i dodajemy go do listy odwiedzonych wierzchołków oraz do kolejki
- dopóki kolejka nie jest pusta:
  - niech v będzie pierwszym wierzchołkiem pobranym z kolejki
  - odwiedzamy wierzchołek v
  - dla każdego wierzchołka sąsiadującego z v: jeżeli nie został odwiedzony i nie jest w kolejce, to umieszczamy go w kolejce FIFO (w kolejności numeru).

(n) `deep_first_traversal` (przechodzenie w głąb) Odwiedzanie rozpoczynamy od wierzchołka początkowego i oznaczamy go jako odwiedzone. Następnie przechodzimy do sąsiada wierzchołka początkowego z najmniejszym numerem i je również oznaczamy jako odwiedzone. Operację kończymy gdy zostaną odwiedzone tym sposobem wszystkie wierzchołki.

(o) funkcja `from_adjacency_matrix(matrix)`, stworzy graf według macierze sąsiedztwa z wartościami $'v0', 'v1', 'v2', ...$ w wierzchołkach.

Testy:

```
G = Graph()
assert G.size() == 0
assert G.edge_size() == 0
G.add_vertex('v0')
assert G.size() == 1
assert G.edge_size() == 0
G.add_vertex('v1')
G.cone('v2')
assert G.edge_size() == 2
```

```
assert G.is_edge(G.V[0], G.V[1]) == False
G.add_vertex('v3')
G.add_edge(G.V[0],G.V[3])
G_complete = Graph()
G_complete.complete_graph('x0','x1','x2','x3')
assert G_complete.size() == 4
assert G_complete.edge_size() == 6
assert G_complete.is_edge(G_complete.V[0], G_complete.V[1])
assert G.adjacency_matrix()==[[0, 0, 1, 1], [0, 0, 1, 0],
             [1, 1, 0, 0], [1, 0,0,0]]
assert G_complete.adjacency_matrix()==[[0, 1, 1, 1,], [1, 0, 1, 1],
       [1, 1, 0, 1], [1, 1, 1, 0]]
assert G.neighbours_dict_data() == {'v0': ['v2', 'v3'], 'v1': ['v2'],
                         'v2': ['v0', 'v1'], 'v3': ['v0']}
assert G_complete.neighbours_dict_data() == {'x0': ['x1','x2', 'x3'],
         'x1': ['x0', 'x2', 'x3'], 'x2': ['x0', 'x1', 'x3'],
         'x3': [ 'x0',   'x1', 'x2']}
G_breadth = []
G.breadth_first_traversal(G_breadth.append)
assert [x.data for x in G_breadth] == ['v0', 'v2', 'v3', 'v1']
G_deep = []
G.deep_first_traversal(G_deep.append)
assert [x.data for x in G_deep] == ['v0', 'v2', 'v1', 'v3']

G = Graph()
G.from_adjacency_matrix([[0,1,1,0,0],[1,0,1,0,0],
              [1,1,0,1,1],[0,0,1,0,0],[0,0,1,0,0]])
assert G.size() == 5

G_breadth = []
G.breadth_first_traversal(G_breadth.append)
assert [x.data for x in G_breadth] == ['v0', 'v1', 'v2', 'v3', 'v4']
G_deep = []
G.deep_first_traversal(G_deep.append)
assert [x.data for x in G_deep] == ['v0', 'v1', 'v2', 'v3', 'v4']
```