**Міністерство освіти і науки України**
**Національний технічний університет України**
**«Київський політехнічний інститут імені Ігоря Сікорського»**
**Факультет інформатики та обчислювальної техніки**
**Кафедра обчислювальної техніки**

# Лабораторна робота №3

з дисципліни
«Алгоритми і структури даних»

Виконав:

Студент групи ІМ-41

Димура Ілля Олександрович
Номер у списку групи: 7

Перевірив:

Сергієнко А. М.

Київ 2025

## Завдання

1. Представити у програмі напрямлений і ненапрямлений графи з заданими параметрами:

    • кількість вершин n;

    • розміщення вершин;

    • матриця суміжності A.

2. Створити програму для формування зображення напрямленого і ненапрямленого графів у графічному вікні.

## Варіант 7:

$n_1n_2n_3n_4 = 4107$

Розміщення вершин: колом з вершиною в центрі

Кількість вершин: $10 + n_3 = 10$

## Текст програми

```python
import math
import random
import tkinter as tk
from typing import List, Tuple, Set, Sequence


# types
Coord = Tuple[float, float]
Matrix = List[List[int]]


# config
VARIANT: int = 4107
PANEL_SIZE: int = 600  # graph render size
PANEL_GAP: int = 40  # gap between graphs
```

```
OUTER_RADIUS: float = 0.40 * PANEL_SIZE
NODE_RADIUS: int = 22
EDGE_WIDTH: int = 3



def generate_directed_matrix(size: int,
                             seed: int,
                             n3: int,
                             n4: int) -> Matrix:
    random.seed(seed)
    k: float = 1.0 - n3 * 0.02 - n4 * 0.005 - 0.25
    return [
        [1 if random.uniform(0.0, 2.0) * k >= 1.0 else 0 for _ in
range(size)]
        for _ in range(size)
    ]



def to_undirected(matrix: Matrix) -> Matrix:
    n: int = len(matrix)
    result: Matrix = [row[:] for row in matrix]
    for i in range(n):
        for j in range(i + 1, n):
            result[i][j] = result[j][i] = 1 if matrix[i][j] or
matrix[j][i] else 0
    return result



def print_matrix(matrix: Matrix, title: str) -> None:
```

```
    n: int = len(matrix)
    print(title)
    print("   " + "".join(f"{j + 1:>2}" for j in range(n)))
    print("   " + "-" * (3 * n))
    for i in range(n):
        row: str = " ".join(
            f"\033[31m{matrix[i][j]}\033[0m" if i == j else
str(matrix[i][j])
            for j in range(n)
        )
        print(f"{i + 1:>2}| {row}")
    print()


def node_positions(count: int,
                   center_idx: int,
                   offset_x: int) -> List[Coord]:
    cx: float = offset_x + PANEL_SIZE / 2
    cy: float = PANEL_SIZE / 2
    positions: List[Coord] = [None] * count  # type: ignore
    positions[center_idx] = (cx, cy)

    outer: List[int] = [i for i in range(count) if i != center_idx]
    for k, idx in enumerate(outer):
        angle: float = 2 * math.pi * k / len(outer)
        positions[idx] = (cx + OUTER_RADIUS * math.cos(angle),
                          cy + OUTER_RADIUS * math.sin(angle))
    return positions
```

```
def shift_point(p: Coord, q: Coord, distance: float) -> Coord:
   dx: float = q[0] - p[0]
   dy: float = q[1] - p[1]
   length: float = math.hypot(dx, dy)
   return (p[0] + dx / length * distance,
           p[1] + dy / length * distance)



def draw_node(canvas: tk.Canvas, x: float, y: float, label: str) ->
None:
   canvas.create_oval(x - NODE_RADIUS, y - NODE_RADIUS,
                      x + NODE_RADIUS, y + NODE_RADIUS,
                      fill="coral", outline="black", width=2)
   canvas.create_text(x, y, text=label, font=("Arial", 12, "bold"))



def draw_straight_edge(canvas: tk.Canvas,
                       p_from: Coord,
                       p_to: Coord,
                       with_arrow: bool) -> None:
   a: Coord = shift_point(p_from, p_to, NODE_RADIUS)
   b: Coord = shift_point(p_to, p_from, NODE_RADIUS * (1.25 if
with_arrow else 1))
   canvas.create_line(*a, *b,
                      width=EDGE_WIDTH, fill="black",
                      arrow=tk.LAST if with_arrow else tk.NONE,
                      arrowshape=(12, 14, 6), capstyle=tk.ROUND)
```

```
def draw_angled_edge(canvas: tk.Canvas,
                     p_from: Coord,
                     p_to: Coord,
                     vertices: Sequence[Coord],
                     with_arrow: bool) -> None:
    dx, dy = p_to[0] - p_from[0], p_to[1] - p_from[1]
    length: float = math.hypot(dx, dy)
    a: Coord = (p_from[0] + dx / length * NODE_RADIUS,
                p_from[1] + dy / length * NODE_RADIUS)
    b: Coord = (p_to[0] - dx / length * NODE_RADIUS * 1.25,
                p_to[1] - dy / length * NODE_RADIUS * 1.25)

    mid: Coord = ((a[0] + b[0]) / 2, (a[1] + b[1]) / 2)
    offset: float = length * math.tan(0.035 * math.pi)

    best_midpoint: Coord | None = None
    best_clearance: float = -1.0
    for sign in (1, -1):
        perp: Coord = (-dy / length * offset * sign,
                       dx / length * offset * sign)
        m: Coord = (mid[0] + perp[0], mid[1] + perp[1])
        clearance: float = min(
            math.hypot(m[0] - vx, m[1] - vy)
            for (vx, vy) in vertices if (vx, vy) not in (p_from, p_to)
        )
        if clearance > best_clearance:
            best_clearance, best_midpoint = clearance, m  # type:
ignore
```

```
   canvas.create_line(*a, *best_midpoint, width=EDGE_WIDTH,
                      fill="black", capstyle=tk.ROUND)
   canvas.create_line(*best_midpoint, *b, width=EDGE_WIDTH,
                      fill="black",
                      arrow=tk.LAST if with_arrow else tk.NONE,
                      arrowshape=(12, 14, 6), capstyle=tk.ROUND)


def draw_loop(canvas: tk.Canvas,
              x: float, y: float,
              with_arrow: bool) -> None:
   points: int = 16
   radius: float = NODE_RADIUS * 0.9
   start_a, end_a = -0.9 * math.pi, 0.55 * math.pi
   side: int = -1  # left
   sx, sy = x + side * NODE_RADIUS, y - 0.85 * NODE_RADIUS

   coords: List[float] = []
   for i in range(points):
       ang: float = start_a + (end_a - start_a) * i / (points - 1)
       coords += [sx + side * radius * math.cos(ang),
                  sy + radius * math.sin(ang)]

   canvas.create_line(*coords, smooth=True,
                      width=EDGE_WIDTH, fill="black",
                      arrow=tk.LAST if with_arrow else tk.NONE,
                      arrowshape=(12, 14, 6), capstyle=tk.ROUND)
```

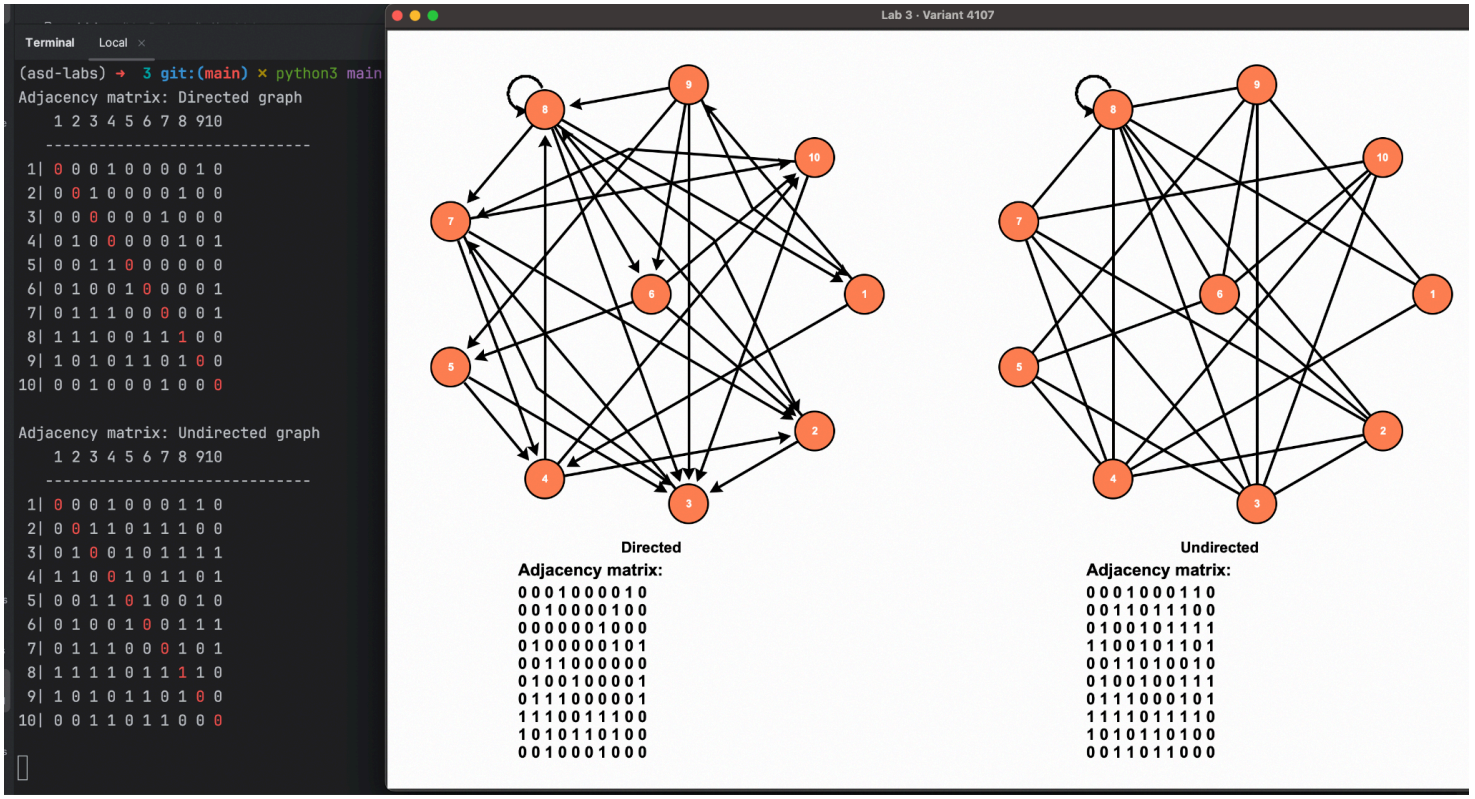
```
def render_matrix(c: tk.Canvas,
                  mx: Matrix,
                  origin: Coord,
                  ):
   ox, oy = origin
   n = len(mx)
   line_h = 20
   c.create_text(ox, oy, anchor="w", fill="black", text="Adjacency
matrix:", font=("Arial", 19, "bold"))
   oy += line_h + 5
   for i in range(n):
       row_str = " ".join(
           f"{mx[i][j]}" for j in range(n)
       )
       c.create_text(ox, oy + i * line_h, anchor="w",
                     text=row_str, fill="black", font=("Arial", 18,
"bold"))



def draw_graph(canvas: tk.Canvas,
               positions: List[Coord],
               matrix: Matrix,
               offset_x: int,
               directed: bool) -> None:
   processed: Set[Tuple[int, int]] = set()
   count: int = len(matrix)

   for i in range(count):
```

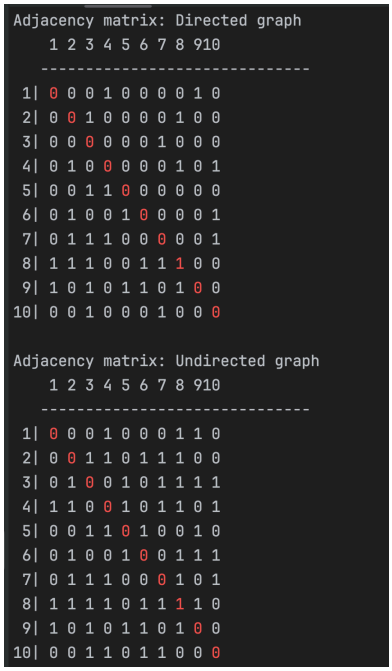
```
        traversal = range(count) if directed else range(i, count)
        for j in traversal:
            if not matrix[i][j]:
                continue
            if i == j:
                draw_loop(canvas, *positions[i], with_arrow=True)
                continue

            if directed and matrix[j][i] and (j, i) not in processed:
                draw_straight_edge(canvas, positions[i], positions[j],
True)
                draw_angled_edge(canvas, positions[j], positions[i],
                                 positions, True)
                processed.update({(i, j), (j, i)})
            elif not directed or (i, j) not in processed:
                draw_straight_edge(canvas, positions[i], positions[j],
directed)
                if directed:
                    processed.add((i, j))

    for idx, (x, y) in enumerate(positions):
        draw_node(canvas, x, y, str(idx + 1))

    caption: str = "Directed" if directed else "Undirected"
    canvas.create_text(offset_x + PANEL_SIZE / 2, PANEL_SIZE - 15,
                       text=caption, fill="black", font=("Arial", 17,
"bold"))

    render_matrix(canvas, matrix,
                  origin=(offset_x + PANEL_SIZE / 4, PANEL_SIZE + 10),
```

```
                )



if __name__ == "__main__":
    n1, n2, n3, n4 = map(int, str(VARIANT).zfill(4))
    vertex_count: int = 10 + n3

    directed_mx: Matrix = generate_directed_matrix(vertex_count,
VARIANT, n3, n4)
    undirected_mx: Matrix = to_undirected(directed_mx)

    print_matrix(directed_mx, "Adjacency matrix: Directed graph")
    print_matrix(undirected_mx, "Adjacency matrix: Undirected graph")

    canvas_width: int = 2 * PANEL_SIZE + PANEL_GAP
    root = tk.Tk()
    root.title(f"Lab 3 · Variant {VARIANT}")
    canvas = tk.Canvas(root, width=canvas_width, height=PANEL_SIZE +
330, bg="white")
    canvas.pack()

    pos_directed: List[Coord] = node_positions(vertex_count,
vertex_count // 2, offset_x=0)
    pos_undirected: List[Coord] = node_positions(vertex_count,
vertex_count // 2,
                                                 offset_x=PANEL_SIZE +
PANEL_GAP)
    draw_graph(canvas, pos_directed, directed_mx, offset_x=0,
directed=True)
    draw_graph(canvas, pos_undirected, undirected_mx,
offset_x=PANEL_SIZE + PANEL_GAP, directed=False)
```

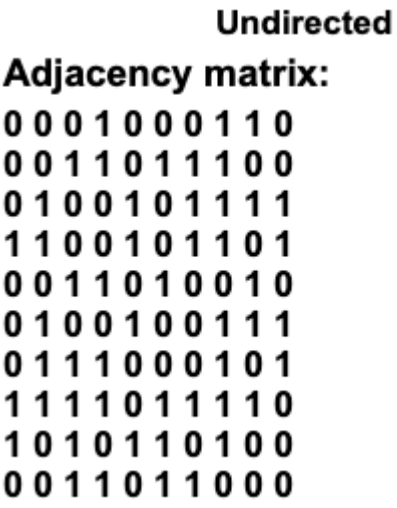
```
root.mainloop()
```

# Тести програми:







# Висновок:

Лабораторна робота була виконана на мові програмування Python, використовуючи кросплатформну бібліотеку tkinter. Для покращення якості зображенна і можливості краще розрізняти ребра, під час малювання окрім прямих також використовуються ламані.