

**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 №5

з дисципліни
«Алгоритми і структури даних»

Виконав:

Студент групи IM-41

Димура Ілля Олександрович

Номер у списку групи: 7

Перевірив:

Сергієнко А. М.

Київ 2025

Завдання

1. Представити напрямлений граф із заданими параметрами так само, як у лабораторній роботі 3.

Відмінність: коефіцієнт $k = 1.0 - n_3 * 0.01 - n_4 * 0.005 - 0.15$
2. Створити програму, яка виконує обхід напрямленого графа вшир (BFS) та вглиб (DFS).
 - обхід починати з вершини із найменшим номером, яка має щонайменше одну вихідну дугу;
 - при обході враховувати порядок нумерації;
 - у програмі виконання обходу відображати покроково, черговий крок виконувати за натисканням кнопки у вікні або на клавіатурі.
3. Під час обходу графа побудувати дерево обходу. У програмі дерево обходу виводити покроково у процесі виконання обходу графа. Це можна виконати одним із двох способів:
 - або виділяти іншим кольором ребра графа;
 - або будувати дерево обходу поряд із графом.
4. Зміну статусів вершин у процесі обходу продемонструвати зміною кольорів вершин, графічними позначками тощо, або ж у процесі обходу виводити протокол обходу у графічне вікно або в консоль.
5. Якщо після обходу графа лишилися невідвідані вершини, продовжувати обхід з невідвіданої вершини з найменшим номером, яка має щонайменше одну вихідну дугу.

Варіант 7:

$$n_1n_2n_3n_4 = 4107$$

Розміщення вершин: колом з вершиною в центрі

Кількість вершин: $10 + n_3 = 10$

Текст програми

```
import math
import random
import tkinter as tk
from typing import List, Tuple, Dict, Set, Generator, Optional
```

```
VARIANT: int = 4107 # група ім41, варіант 07
PANEL_SIZE: int = 600
NODE_RADIUS: int = 22
EDGE_WIDTH: int = 3
OUTER_RADIUS: float = 0.40 * PANEL_SIZE

COLOR_NODE_DEFAULT = "coral"
COLOR_NODE_VISITED = "lightgreen"
COLOR_EDGE_DEFAULT = "gray"
COLOR_EDGE_TREE = "red"

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

def draw_angled_edge(canvas: tk.Canvas,
                     p_from: Coord,
                     p_to: Coord,
                     vertices: List[Coord],
                     ) -> list[int]:
    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

id1 = canvas.create_line(*a, *best_midpoint,
                        width=EDGE_WIDTH,
                        fill=COLOR_EDGE_DEFAULT,
                        capstyle=tk.ROUND)

id2 = canvas.create_line(*best_midpoint, *b,
                        width=EDGE_WIDTH,
                        fill=COLOR_EDGE_DEFAULT,
                        arrow=tk.LAST,
                        arrowshape=(12, 14, 6),
                        capstyle=tk.ROUND)

return [id1, id2]

```

```
def generate_directed_matrix(size: int, seed: int, n3: int, n4: int) → Matrix:
    random.seed(seed)
    k: float = 1.0 - n3 * 0.01 - n4 * 0.005 - 0.15
    return [
        [1 if random.uniform(0.0, 2.0) * k ≥ 1.0 else 0 for _ in range(size)]
        for _ in range(size)
    ]

def node_positions(count: int, center_idx: int = 0) → List[Coord]:
    cx, cy = PANEL_SIZE / 2, PANEL_SIZE / 2
    pos: List[Coord] = [None] * count # type: ignore
    pos[center_idx] = (cx, cy)

    outer = [i for i in range(count) if i ≠ center_idx]
    for k, idx in enumerate(outer):
        ang = 2 * math.pi * k / len(outer)
        pos[idx] = (cx + OUTER_RADIUS * math.cos(ang), cy + OUTER_RADIUS * math.sin(ang))

    return pos

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

```
def draw_node(cv: tk.Canvas, x: float, y: float, label: str) → int:
    oid = cv.create_oval(
        x - NODE_RADIUS,
        y - NODE_RADIUS,
        x + NODE_RADIUS,
        y + NODE_RADIUS,
        fill=COLOR_NODE_DEFAULT,
        outline="black",
        width=2,
    )
    cv.create_text(x, y, text=label, font=("Arial", 12, "bold"))
    return oid

def draw_straight_edge(cv: tk.Canvas,
                      p_from: Coord, p_to: Coord,
                      both: bool = False) → int:
    a = shift_point(p_from, p_to, NODE_RADIUS)
    b = shift_point(p_to, p_from, NODE_RADIUS * 1.25)
    return cv.create_line(
        *a, *b,
        width=EDGE_WIDTH,
        fill=COLOR_EDGE_DEFAULT,
        arrow=tk.BOTH if both else tk.LAST, # ← ***
        arrowshape=(12, 14, 6),
        capstyle=tk.ROUND,
    )
```

```

def draw_loop(cv: tk.Canvas, x: float, y: float) → int:
    pts, r = 16, NODE_RADIUS * 0.9
    start_a, end_a = -0.9 * math.pi, 0.55 * math.pi
    side = -1
    sx, sy = x + side * NODE_RADIUS, y - 0.85 * NODE_RADIUS
    coords: List[float] = []
    for i in range(pts):
        ang = start_a + (end_a - start_a) * i / (pts - 1)
        coords += [sx + side * r * math.cos(ang), sy + r * math.sin(ang)]
    eid = cv.create_line(
        *coords,
        smooth=True,
        width=EDGE_WIDTH,
        fill=COLOR_EDGE_DEFAULT,
        arrow=tk.LAST,
        arrowshape=(12, 14, 6),
        capstyle=tk.ROUND,
    )
    return eid

def draw_graph(
    cv: tk.Canvas,
    positions: List[Coord],
    mx: Matrix
) → Tuple[Dict[int, int], Dict[Tuple[int, int], List[int]]]:
    n = len(mx)
    node_items: Dict[int, int] = {}

```

```

edge_items: Dict[Tuple[int, int], List[int]] = {}

for i, (x, y) in enumerate(positions):
    node_items[i] = draw_node(cv, x, y, str(i + 1))

for i in range(n):
    for j in range(n):
        if mx[i][j] == 0 or (i, j) in edge_items:
            continue

        if i == j:
            eid = draw_loop(cv, *positions[i])
            edge_items[(i, j)] = [eid]
            continue

        if mx[j][i]:
            # u → v
            eids_uv = draw_straight_edge(cv, positions[i],
positions[j])
            edge_items[(i, j)] = [eids_uv]
            # v → u
            eids_vu = draw_angled_edge(cv, positions[j],
positions[i], positions)
            edge_items[(j, i)] = eids_vu

        else:
            eid = draw_straight_edge(cv, positions[i],
positions[j])
            edge_items[(i, j)] = [eid]

return node_items, edge_items

```

```

class TraversalVisualizer:

    def __init__(self, cv: tk.Canvas, mx: Matrix, nodes: Dict[int, int], edges: Dict[Tuple[int, int], List[int]], mode: str):
        self.cv, self.mx, self.nodes, self.edges = cv, mx, nodes, edges
        self.n = len(mx)
        self.mode = mode # "BFS" or "DFS"
        self.path: List[int] = []
        self.visited = [False] * self.n
        self.order: List[int] = [-1] * self.n
        self.tree_mx = [[0] * self.n for _ in range(self.n)]
        self.gen = self._bfs() if mode == "BFS" else self._dfs()

    def _next_start(self, after: int = -1) → Optional[int]:
        for v in range(after + 1, self.n):
            if not self.visited[v] and any(self.mx[v]):
                return v
        return None

    def _bfs(self) → Generator[Tuple[str, Tuple[int, int]], None, None]:
        from collections import deque
        start = self._next_start()
        while start is not None:
            q: deque[int] = deque([start])
            self.visited[start] = True
            yield ("vertex", (start, -1))
            while q:
                v = q.popleft()
                for w in self.edges[v]:
                    if not self.visited[w]:
                        self.order[w] = self.order[v] + 1
                        self.tree_mx[v][self.order[w]] = w
                        self.visited[w] = True
                        q.append(w)

```

```

while q:

    v = q.popleft()

    for w in range(self.n):

        if self.mx[v][w] and not self.visited[w]:

            yield ("edge", (v, w))

            self.tree_mx[v][w] = 1

            self.visited[w] = True

            yield ("vertex", (w, v))

            q.append(w)

    start = self._next_start(start)

def _dfs(self) → Generator[Tuple[str, Tuple[int, int]], None, None]:
    stack: List[Tuple[int, int]] = []
    start = self._next_start()
    while start is not None:
        stack.append((start, 0))
        self.visited[start] = True
        yield ("vertex", (start, -1))
        while stack:
            v, nxt = stack[-1]
            w = None
            for j in range(nxt, self.n):
                if self.mx[v][j] and not self.visited[j]:
                    w = j
                    break
            stack[-1] = (v, (w + 1) if w is not None else self.n)
            if w is None:
                stack.pop()

```

```

        continue

    yield ("edge", (v, w))
    self.visited[w] = True
    yield ("vertex", (w, v))
    stack.append((w, 0))

start = self._next_start(start)

def step(self) → bool:
    try:
        kind, (u, parent) = next(self.gen)
    except StopIteration:
        return False

    if kind == "vertex":
        self.order[u] = len(self.path)
        self.path.append(u)
        self.cv.itemconfig(self.nodes[u], fill=COLOR_NODE_VISITED)
        if parent == -1:
            print(f"Starting from {u + 1}")
        else:
            print(f"{parent + 1} → {u + 1}")
    else:
        for eid in self.edges[(u, parent)]:
            self.cv.itemconfig(eid, fill=COLOR_EDGE_TREE,
width=EDGE_WIDTH + 1)
            self.cv.tag_raise(eid)

    return True

```

```

def print_matrix(mx: Matrix, label: str):
    n = len(mx)
    print(f"{label}:")
    for i in range(n):
        row_str = " ".join(
            f"{mx[i][j]}" for j in range(n)
        )
        print(row_str)
    print()

def print_mapping(mapping: list[int]):
    pairs = [(v, num) for v, num in enumerate(mapping) if num != -1]
    pairs.sort(key=lambda x: x[1])

    print("Vertex-to-order mapping:")
    for old, new in pairs:
        print(f"new {new + 1:2d} ← old {old + 1:2d}")
    print()

class App:
    def __init__(self, root: tk.Tk):
        root.title(f"BFS / DFS Traversal · Variant {VARIANT}")

        n1, n2, n3, n4 = map(int, str(VARIANT).zfill(4))
        self.n = 10 + n3

        self.mx = generate_directed_matrix(self.n, VARIANT, n3, n4)

```

```
print_matrix(self.mx, "Directed graph adjacency matrix")

        self.canvas = tk.Canvas(root, width=PANEL_SIZE,
height=PANEL_SIZE, bg="white")
        self.canvas.grid(row=0, column=0, columnspan=4, padx=8, pady=8)

        self.nodes, self.edges = draw_graph(self.canvas,
node_positions(self.n, self.n // 2), self.mx)

        self.mode_var = tk.StringVar(value="BFS")
        tk.Radiobutton(root, text="BFS", variable=self.mode_var,
value="BFS", font=("Arial", 12, "bold")).grid(row=1,
column=0)

        tk.Radiobutton(root, text="DFS", variable=self.mode_var,
value="DFS", font=("Arial", 12, "bold")).grid(row=1,
column=1)

        tk.Button(root, text="Start", width=9,
command=self.start).grid(row=1, column=2, padx=4)

        self.btn_next = tk.Button(root, text="Next step", width=11,
state=tk.DISABLED, command=self.next_step)
        self.btn_next.grid(row=1, column=3, padx=4)

        self.status = tk.StringVar(value="Choose BFS or DFS → Start")
        tk.Label(root, textvariable=self.status, font=("Arial",
12)).grid(row=2, column=0, columnspan=4, pady=6)

root.bind("<space>", lambda _: self.next_step())
self.vis: Optional[TraversalVisualizer] = None
```

```

def reset_colors(self):
    for oid in self.nodes.values():
        self.canvas.itemconfig(oid, fill=COLOR_NODE_DEFAULT)
    for ids in self.edges.values():
        for eid in ids:
            self.canvas.itemconfig(eid, fill=COLOR_EDGE_DEFAULT,
width=EDGE_WIDTH)

def start(self):
    self.reset_colors()
    mode = self.mode_var.get()
    self.vis = TraversalVisualizer(self.canvas, self.mx,
self.nodes, self.edges, mode)
    self.btn_next.config(state=tk.NORMAL)
    self.status.set(f"{mode} started - Space / Next step ...")
    print(f"{mode} traversal started")

def next_step(self):
    if not self.vis:
        return
    if not self.vis.step():
        self.btn_next.config(state=tk.DISABLED)
        text = f"{self.mode_var.get()} traversal finished"
        self.status.set(text)
        print(text)

    print_matrix(self.vis.tree_mx, "Adjacency matrix of the
traversal tree")
    print_mapping(self.vis.order)

```

```

if __name__ == "__main__":
    tk_root = tk.Tk()
    App(tk_root)
    tk_root.mainloop()

```

Тести програми:

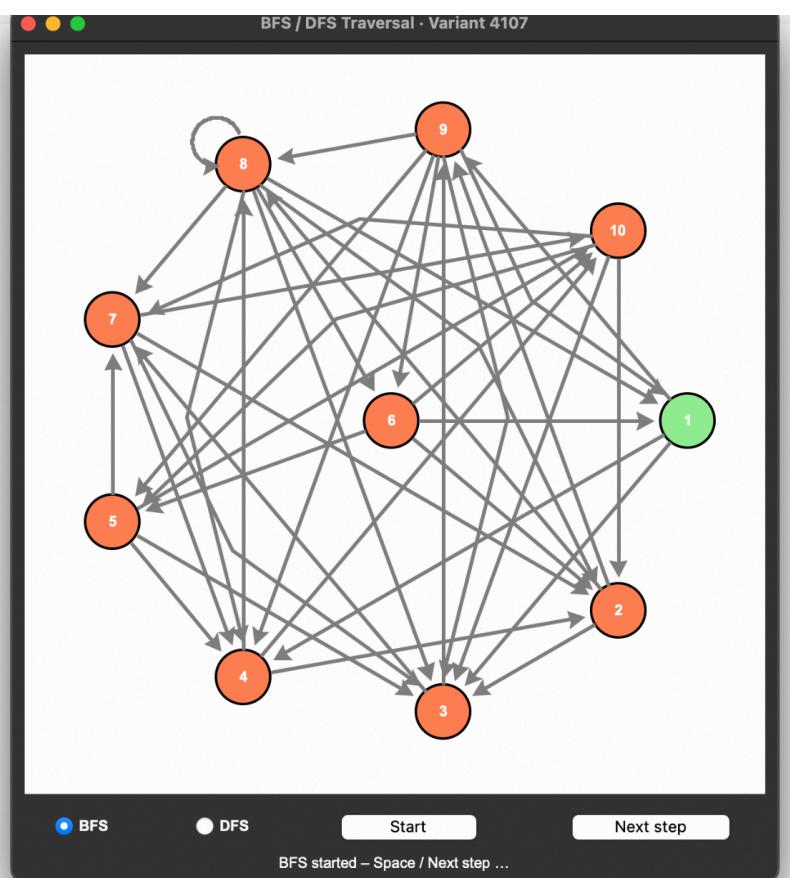
1. BFS

```

(asd-labs) → 5 git:(main) ✘ python3 main.py
Directed graph adjacency matrix:
0 0 1 1 0 0 0 1 0
0 0 1 0 0 0 0 1 1 0
0 0 0 0 0 0 1 0 1 0
0 1 0 0 0 0 0 1 0 1
0 0 1 1 0 0 1 0 0 1
1 1 0 0 1 0 0 0 0 1
0 1 1 1 0 0 0 0 0 1
1 1 1 1 0 1 1 1 0 0
1 0 1 1 1 1 0 1 0 0
0 1 1 0 1 0 1 0 0 0

```

BFS traversal started
Starting from 1
]



Terminal Local × + ▾

(asd-labs) → 5 git:(main) ✘ python3 main.py

Directed graph adjacency matrix:

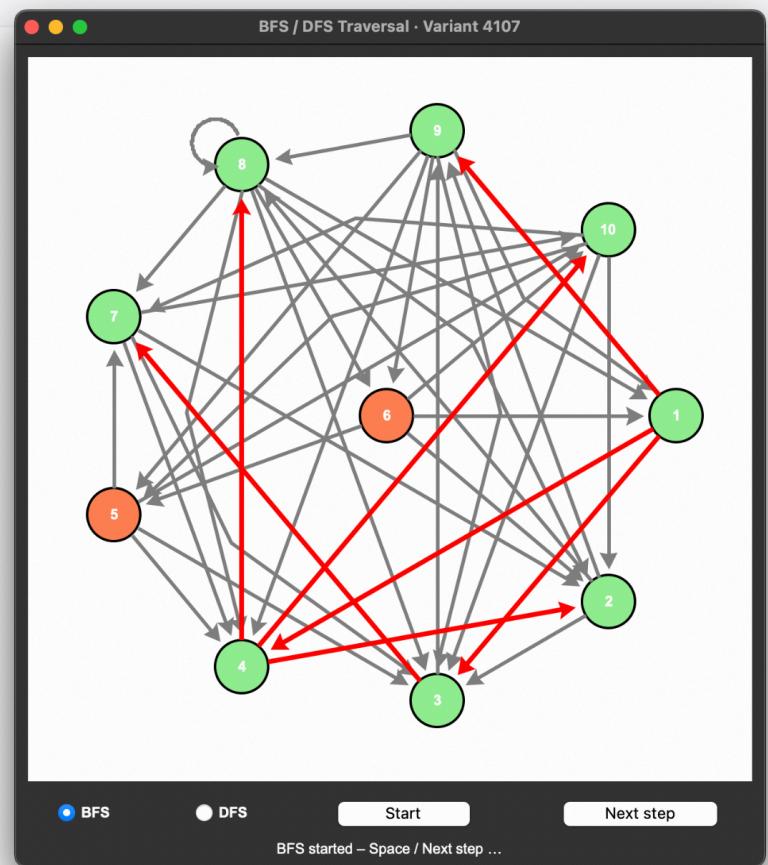
```
0 0 1 1 0 0 0 0 1 0  
0 0 1 0 0 0 0 1 1 0  
0 0 0 0 0 0 1 0 1 0  
0 1 0 0 0 0 0 1 0 1  
0 0 1 1 0 0 1 0 0 1  
1 1 0 0 1 0 0 0 0 1  
0 1 1 1 0 0 0 0 0 1  
1 1 1 1 0 1 1 1 0 0  
1 0 1 1 1 1 0 1 0 0  
0 1 1 0 1 0 1 0 0 0
```

BFS traversal started

Starting from 1

```
1 -> 3  
1 -> 4  
1 -> 9  
3 -> 7  
4 -> 2  
4 -> 8  
4 -> 10
```

█



```
4 -> 2  
4 -> 8  
4 -> 10  
9 -> 5  
9 -> 6
```

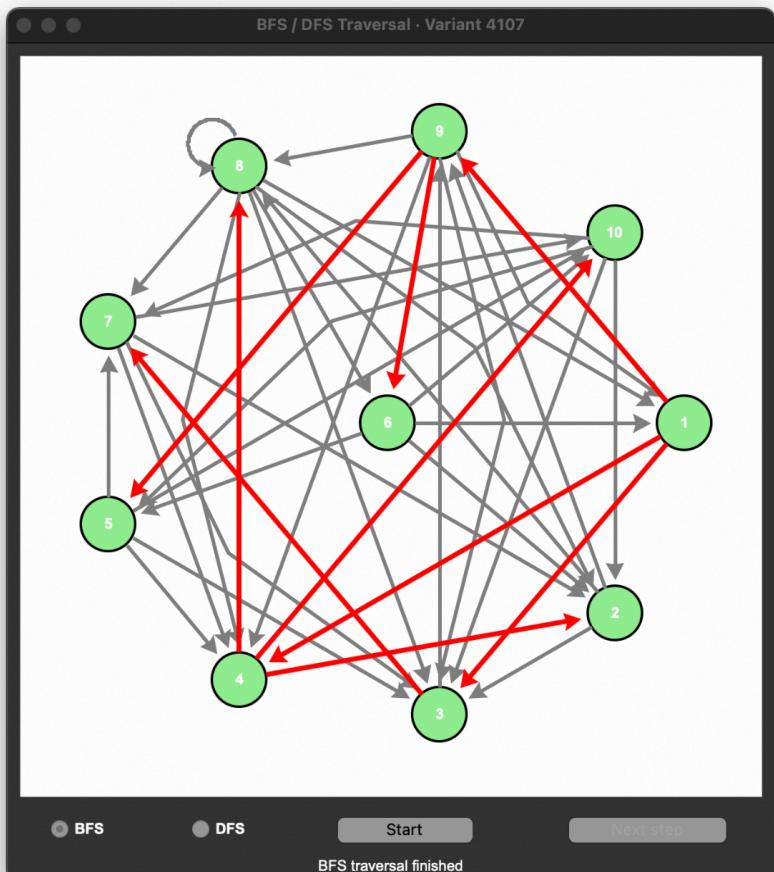
BFS traversal finished

Adjacency matrix of the traversal tree:

```
0 0 1 1 0 0 0 0 1 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 1 0 0 0  
0 1 0 0 0 0 0 1 0 1  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 1 1 0 0 0 0  
0 0 0 0 0 0 0 0 0 0
```

Vertex-to-order mapping:

```
new 1 ← old 1  
new 2 ← old 3  
new 3 ← old 4  
new 4 ← old 9  
new 5 ← old 7  
new 6 ← old 2  
new 7 ← old 8  
new 8 ← old 10  
new 9 ← old 5  
new 10 ← old 6
```



2. DFS

Terminal Local × +

(asd-labs) → 5 git:(main) ✘ python3 main.py

Directed graph adjacency matrix:

```
0 0 1 1 0 0 0 0 1 0
0 0 1 0 0 0 0 1 1 0
0 0 0 0 0 0 1 0 1 0
0 1 0 0 0 0 0 1 0 1
0 0 1 1 0 0 1 0 0 1
1 1 0 0 1 0 0 0 0 1
0 1 1 1 0 0 0 0 0 1
1 1 1 1 0 1 1 1 0 0
1 0 1 1 1 1 0 1 0 0
0 1 1 0 1 0 1 0 0 0
```

DFS traversal started
Starting from 1

BFS DFS Start Next step

DFS started – Space / Next step ...

(asd-labs) → 5 git:(main) ✘ python3 main.py

Directed graph adjacency matrix:

0	0	1	1	0	0	0	1	0	
0	0	1	0	0	0	0	1	1	0
0	0	0	0	0	0	1	0	1	0
0	1	0	0	0	0	0	1	0	1
0	0	1	1	0	0	1	0	0	1
1	1	0	0	1	0	0	0	0	1
0	1	1	1	0	0	0	0	0	1
1	1	1	1	0	1	1	1	0	0
1	0	1	1	1	1	0	1	0	0
0	1	1	0	1	0	1	0	0	0

DFS traversal started

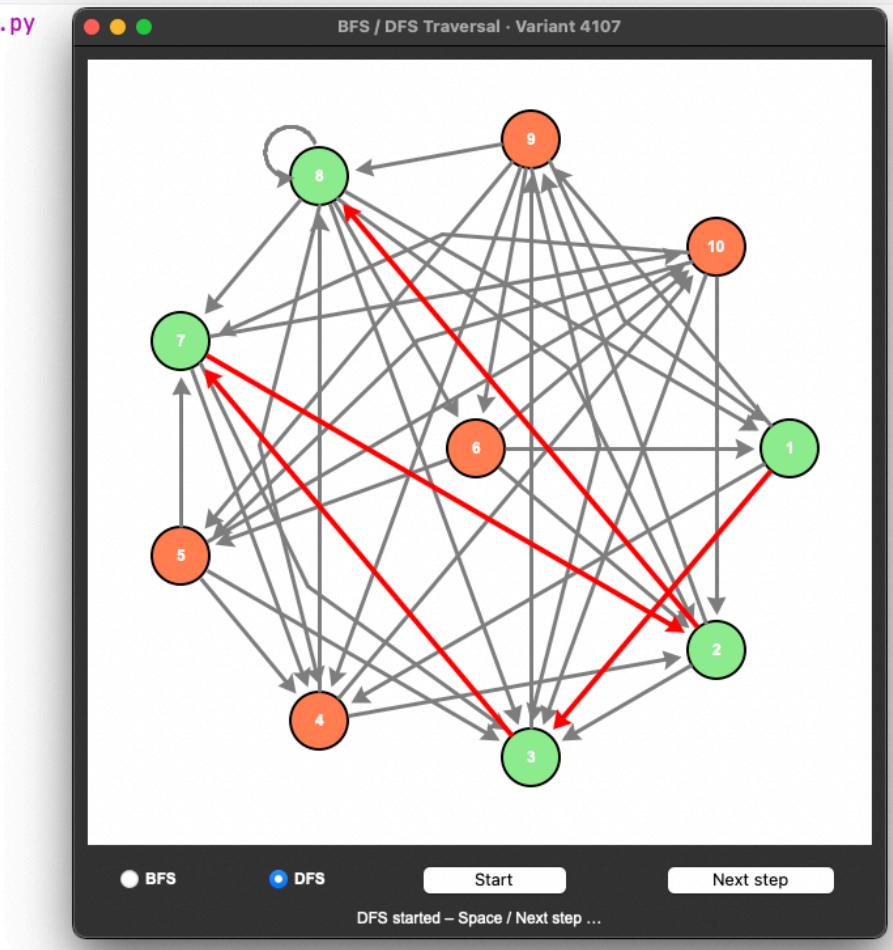
Starting from 1

1 -> 3

3 -> 7

7 -> 2

2 -> 8

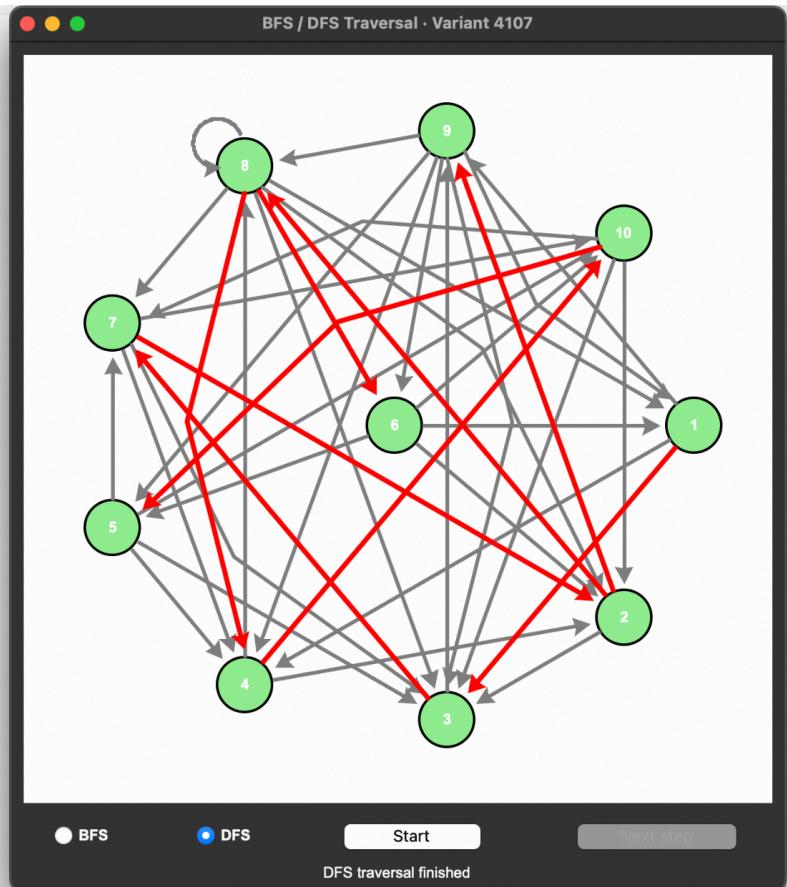


```

2 -> 8
8 -> 4
4 -> 10
10 -> 5
8 -> 6
2 -> 9
DFS traversal finished
Adjacency matrix of the traversal tree:
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

Vertex-to-order mapping:
new 1 ← old 1
> new 2 ← old 3
vi new 3 ← old 7
n new 4 ← old 2
h new 5 ← old 8
new 6 ← old 4
new 7 ← old 10
new 8 ← old 5
) new 9 ← old 6
b new 10 ← old 9

```



Висновок:

Лабораторна робота була виконана на мові програмування Python, використовуючи кросплатформну бібліотеку [tkinter](#). Були імплементовані алгоритми BFS та DFS. Спільна логіка (оновлення кольорів, запису шляху тощо) була винесена окремо в клас, а крок з обходом - абстрагований для можливості змінювати алгоритми обходу. Знання цих алгоритмів є обов'язковим для вирішення більшості задач з графами.