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 ІМ-42
Федоренко Іван Русланович
номер у списку групи: 29

Перевірив:

Сергієнко А. М.

Київ 2025

Постановка задачі

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

Відмінність: коефіцієнт k = 1.0 - n3 * 0.01 - n4 * 0.005 - 0.15.

Отже, матриця суміжності Adir напрямленого графа за варіантом формується таким чином:

1) встановлюється параметр (seed) генератора випадкових чисел, рівне номеру варіанту n1n2n3n4;

2) матриця розміром n n заповнюється згенерованими випадковими числами в діапазоні [0, 2.0);

3) обчислюється коефіцієнт k = 1.0 - n3 * 0.01 - n4 * 0.005 - 0.15, кожен елемент матриці множиться на коефіцієнт k;

4) елементи матриці округлюються: 0 — якщо елемент менший за 1.0,

1 — якщо елемент більший або дорівнює 1.0.

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.

При проєктуванні програми також слід врахувати наступне:

1) мова програмування обирається студентом самостійно;

2) графічне зображення усіх графів має формуватися програмою з тими ж вимогами, як у Лабораторній роботі номер 3;

3) всі графи обов'язково зображувати у графічному вікні;

4) типи та структури даних для внутрішнього представлення всіх даних у програмі слід вибрати самостійно.

Варіант: 4229.

Кількість вершин = 12.

Розміщення: прямокутником з вершиною в центрі.

Код програми лабораторної роботи №3:

Програма складається з наступних файлів:

Main.py, drawing_methods.py, matrix_methods.py, traversal_methods.py.

Main.py

```
import matplotlib.pyplot as plt
from matrix_methods import generate_adjacency_matrix, print_matrix
from drawing_methods import (
    draw_graph,
    get_vertex_positions
)
from traversal_methods import draw_traversal_graph

def main():
    variant_number = 4229
    n3 = 2
    n4 = 9
    n = 10 + n3
    k = 1.0 - n3 * 0.01 - n4 * 0.005 - 0.15

    print(f"Variant number: {variant_number}")
    print(f"Number of vertices n = 10 + {n3} = {n}")

    # PART 5: Graph Traversal - New for Laboratory Work #5
    print("\n=== PART 5: Graph Traversal Analysis ===")

    # Generate matrix using existing function with correct coefficient
    directed_matrix = generate_adjacency_matrix(n, variant_number, k)

    print_matrix(directed_matrix, f"Directed Graph Adjacency Matrix ({n}x{n}) for
Traversal")

    # Get vertex positions
    positions = get_vertex_positions(n, n4)

    # Draw original graph before traversal
    fig = draw_graph(directed_matrix, positions, is_directed=True, title="Graph for
Traversal")
    fig.savefig('graph_for_traversal.png')

    # Perform BFS traversal with visualization
    print("\nStarting Breadth-First Search (BFS)...")
    print("Press any key in plot window to continue step by step")
    bfs_fig, bfs_forest, bfs_order = draw_traversal_graph(directed_matrix, positions,
"BFS")
```

```
    bfs_fig.savefig('bfs_traversal.png')

    # Perform DFS traversal with visualization
    print("\nStarting Depth-First Search (DFS)...")
    print("Press any key in plot window to continue step by step")
    dfs_fig, dfs_forest, dfs_order = draw_traversal_graph(directed_matrix, positions,
"DFS")
    dfs_fig.savefig('dfs_traversal.png')

    # Analyze and compare traversals
    print("\nTraversal Analysis:")
    print(f"BFS visited {len(bfs_order)} vertices")
    print(f"DFS visited {len(dfs_order)} vertices")

    # BFS tree properties
    bfs_tree_edges = sum(len(tree) for tree in bfs_forest)
    print(f"BFS created {len(bfs_forest)} tree(s) with a total of {bfs_tree_edges}
edges")

    # DFS tree properties
    dfs_tree_edges = sum(len(tree) for tree in dfs_forest)
    print(f"DFS created {len(dfs_forest)} tree(s) with a total of {dfs_tree_edges}
edges")

    plt.show()

if __name__ == "__main__":
    main()
```

Drawing_methods.py

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import math

def rotate_around_center(x, y, cx, cy, angle):
    """Обертання точки (x, y) навколо центру (cx, cy) на кут angle (в радіанах)"""
    dx = x - cx
    dy = y - cy
    cos_a = math.cos(angle)
    sin_a = math.sin(angle)
    x_new = cx + dx * cos_a - dy * sin_a
    y_new = cy + dx * sin_a + dy * cos_a
    return (x_new, y_new)

def draw_self_loop(ax, pos, color='blue', linewidth=1.5, is_directed=True):
    """Draw a self-loop using polygonal lines under the vertex (with rotation)"""
    cx, cy = pos
    R = 0.5
    index_angle = 0
    theta = index_angle

    cx += R * math.sin(theta)
    cy -= R * math.cos(theta)

    dx = 3 * R / 4
    dy = R * (1 - math.sqrt(7)) / 4

    p1 = (cx - dx, cy - dy)
    p2 = (cx - 3 * dx / 2, cy - R / 2)
    p3 = (cx + 3 * dx / 2, cy - R / 2)
    p4 = (cx + dx, cy - dy)

    p1 = rotate_around_center(p1[0], p1[1], cx, cy, theta)
    p2 = rotate_around_center(p2[0], p2[1], cx, cy, theta)
    p3 = rotate_around_center(p3[0], p3[1], cx, cy, theta)
    p4 = rotate_around_center(p4[0], p4[1], cx, cy, theta)

    ax.plot([p1[0], p2[0]], [p1[1], p2[1]], color=color, linewidth=linewidth)
    ax.plot([p2[0], p3[0]], [p2[1], p3[1]], color=color, linewidth=linewidth)

    if is_directed:
        dx_arrow = p4[0] - p3[0]
        dy_arrow = p4[1] - p3[1]
        ax.arrow(p3[0], p3[1], dx_arrow, dy_arrow,
                 head_width=0.15, head_length=0.15,
                 fc=color, ec=color, linewidth=linewidth,
```

```python
                    length_includes_head=True)
    else:
        ax.plot([p3[0], p4[0]], [p3[1], p4[1]], color=color, linewidth=linewidth)

def draw_edge(ax, start, end, is_directed=True, color='blue', linewidth=1.5):
    """Draw an edge between two vertices"""
    dx = end[0] - start[0]
    dy = end[1] - start[1]

    dist = np.sqrt(dx**2 + dy**2)
    if dist < 0.001:
        return

    vertex_radius = 0.5
    ratio = vertex_radius / dist
    start_x = start[0] + dx * ratio
    start_y = start[1] + dy * ratio
    end_x = end[0] - dx * ratio
    end_y = end[1] - dy * ratio

    rad = 0.2

    if is_directed:
        arrow = patches.FancyArrowPatch(
            (start_x, start_y), (end_x, end_y),
            arrowstyle='->',
            color=color,
            linewidth=linewidth,
            connectionstyle=f'arc3,rad={rad}',
            mutation_scale=15
        )
    else:
        arrow = patches.FancyArrowPatch(
            (start_x, start_y), (end_x, end_y),
            arrowstyle='-',
            color=color,
            linewidth=linewidth,
            connectionstyle=f'arc3,rad={rad}'
        )

    ax.add_patch(arrow)
    return arrow

def draw_graph(adjacency_matrix, positions, is_directed=True, title="Graph"):
    """Draw a graph based on adjacency matrix and vertex positions"""
    n = adjacency_matrix.shape[0]

    fig, ax = plt.subplots(figsize=(12, 10))
```

```python
    rect = patches.Rectangle((-6, -4), 12, 8, linewidth=1, edgecolor='gray',
                              facecolor='none', linestyle='--')
    ax.add_patch(rect)

    for i in range(n):
        for j in range(n):
            if adjacency_matrix[i, j] == 1:
                if i == j:
                    draw_self_loop(ax, positions[i], color='black', linewidth=1.5,
is_directed=is_directed)
                else:
                    draw_edge(ax, positions[i], positions[j], is_directed,
color='blue')

    for i, pos in enumerate(positions):
        circle = plt.Circle(pos, 0.5, fill=True, color='lightblue', edgecolor='blue')
        ax.add_patch(circle)

        ax.text(pos[0], pos[1], str(i+1), horizontalalignment='center',
                verticalalignment='center', fontsize=10, color='black',
fontweight='bold')

    ax.set_aspect('equal')
    margin = 2
    ax.set_xlim(min(positions[:, 0])-margin, max(positions[:, 0])+margin)
    ax.set_ylim(min(positions[:, 1])-margin, max(positions[:, 1])+margin)

    graph_type = "Directed" if is_directed else "Undirected"
    plt.title(f"{title} - {graph_type} Graph - {n} vertices")

    plt.axis('off')
    return fig

def get_component_positions(components):
    """Create positions for components in condensation graph"""
    n_components = len(components)
    radius = 5

    positions = []
    for i in range(n_components):
        angle = 2 * np.pi * i / n_components
        x = radius * np.cos(angle)
        y = radius * np.sin(angle)
        positions.append([x, y])

    return np.array(positions)

def draw_condensation_graph(condensation_matrix, components, positions=None):
    """Draw the condensation graph"""
```

```python
    n_components = len(components)

    if positions is None:
        positions = get_component_positions(components)

    fig, ax = plt.subplots(figsize=(12, 10))

    for i in range(n_components):
        for j in range(n_components):
            if condensation_matrix[i, j] == 1:
                draw_edge(ax, positions[i], positions[j], is_directed=True,
color='red')

    for i, pos in enumerate(positions):
        radius = 0.8  # Larger radius for component nodes
        circle = plt.Circle(pos, radius, fill=True, color='lightgreen',
edgecolor='green')
        ax.add_patch(circle)

        # Label with component number and list of vertices
        component_label = f"C{i+1}: {components[i]}"
        ax.text(pos[0], pos[1], component_label, horizontalalignment='center',
                verticalalignment='center', fontsize=9, color='black')

    ax.set_aspect('equal')
    margin = 3
    ax.set_xlim(min(positions[:, 0])-margin, max(positions[:, 0])+margin)
    ax.set_ylim(min(positions[:, 1])-margin, max(positions[:, 1])+margin)

    plt.title(f"Condensation Graph - {n_components} components")
    plt.axis('off')
    return fig

def get_vertex_positions(n, n4):
    """Get vertex positions based on n4 value"""
    positions = np.zeros((n, 2))

    if n4 in [8, 9]:
        width, height = 12, 8
        positions[n-1] = [0, 0]  # Center vertex

        perimeter_vertices = n - 1
        sides = [0, 0, 0, 0]
        remaining = perimeter_vertices - 4

        for i in range(remaining):
            sides[i % 4] += 1

        vertex_index = 0
```

```
    positions[vertex_index] = [-width/2, height/2]
    vertex_index += 1

    for i in range(sides[0]):
        x = -width/2 + (i+1) * width / (sides[0]+1)
        positions[vertex_index] = [x, height/2]
        vertex_index += 1

    positions[vertex_index] = [width/2, height/2]
    vertex_index += 1

    for i in range(sides[1]):
        y = height/2 - (i+1) * height / (sides[1]+1)
        positions[vertex_index] = [width/2, y]
        vertex_index += 1

    positions[vertex_index] = [width/2, -height/2]
    vertex_index += 1

    for i in range(sides[2]):
        x = width/2 - (i+1) * width / (sides[2]+1)
        positions[vertex_index] = [x, -height/2]
        vertex_index += 1

    positions[vertex_index] = [-width/2, -height/2]
    vertex_index += 1

    for i in range(sides[3]):
        y = -height/2 + (i+1) * height / (sides[3]+1)
        positions[vertex_index] = [-width/2, y]
        vertex_index += 1

return positions
```

Matrix_methods.py:

```
import numpy as np

def calculate_degrees(matrix, is_directed=True):
    """Calculate vertex degrees
    For directed graphs, returns in-degrees and out-degrees
    For undirected graphs, returns degrees
    """
    n = matrix.shape[0]

    if is_directed:
        in_degrees = np.sum(matrix, axis=0)
        out_degrees = np.sum(matrix, axis=1)
        return in_degrees, out_degrees
    else:
        degrees = np.sum(matrix, axis=1)
        return degrees

def generate_adjacency_matrix(n, variant_number, k):
    """Generate directed adjacency matrix based on variant number and k"""
    np.random.seed(variant_number)
    T = np.random.random((n, n)) * 2.0

    print(f"Using k coefficient: {k}")

    A = np.zeros((n, n), dtype=int)
    for i in range(n):
        for j in range(n):
            A[i, j] = 1 if T[i, j] * k >= 1.0 else 0

    return A

def get_undirected_matrix(directed_matrix):
    """Convert directed adjacency matrix to undirected"""
    n = directed_matrix.shape[0]
    undirected_matrix = np.zeros((n, n), dtype=int)

    for i in range(n):
        for j in range(n):
            if directed_matrix[i, j] == 1 or directed_matrix[j, i] == 1:
                undirected_matrix[i, j] = 1
                undirected_matrix[j, i] = 1

    return undirected_matrix

def calculate_strong_connectivity_matrix(reachability):
    """Calculate strong connectivity matrix from reachability matrix"""
    n = reachability.shape[0]
```

```
    strong_connectivity = np.zeros((n, n), dtype=int)

    for i in range(n):
        for j in range(n):
            if reachability[i, j] == 1 and reachability[j, i] == 1:
                strong_connectivity[i, j] = 1

    return strong_connectivity

def print_matrix(matrix, title):
    """Print the adjacency matrix in a readable format"""
    print(f"\n{title}:")
    for row in matrix:
        print(" ".join(map(str, row)))

def calculate_reachability_matrix(matrix):
    """Calculate reachability matrix using transitive closure"""
    n = matrix.shape[0]

    # Initialize reachability with the adjacency matrix
    reachability = matrix.copy()

    # Add self-loops
    for i in range(n):
        reachability[i, i] = 1

    # Warshall's algorithm for transitive closure
    for k in range(n):
        for i in range(n):
            for j in range(n):
                reachability[i, j] = reachability[i, j] or (reachability[i, k] and
reachability[k, j])

    return reachability
```

Traversal_methods.py:

```
import numpy as np
import matplotlib.pyplot as plt
from collections import deque
from matplotlib.patches import Patch
from drawing_methods import draw_edge, draw_self_loop

def find_start_vertex(adjacency_matrix):
    """Find the vertex with smallest index that has at least one outgoing edge"""
    n = adjacency_matrix.shape[0]
    for i in range(n):
        if np.sum(adjacency_matrix[i]) > 0:
            return i
    return 0  # Default to vertex 0 if no vertices have outgoing edges

def wait_for_key(fig):
    """Wait for a key press event"""
    def on_key(event):
        if event.key:
            fig.canvas.stop_event_loop()

    fig.canvas.mpl_connect('key_press_event', on_key)
    fig.canvas.start_event_loop()

def bfs(adjacency_matrix, start_vertex=None, ax=None, positions=None,
        vertex_circles=None, edge_arrows=None, step_by_step=False):
    """Performs a breadth-first search traversal of the graph."""
    n = adjacency_matrix.shape[0]

    if start_vertex is None:
        start_vertex = find_start_vertex(adjacency_matrix)

    visited = [False] * n
    queue = deque([start_vertex])
    visited[start_vertex] = True
    parent = {start_vertex: None}
    traversal_edges = []
    traversal_order = []

    if ax is not None and positions is not None and vertex_circles is not None:
        vertex_circles[start_vertex].set_facecolor('yellow')
        ax.text(positions[start_vertex][0], positions[start_vertex][1] - 0.8,
f"Start",
                horizontalalignment='center', verticalalignment='center', fontsize=8)
        if step_by_step:
            plt.title("BFS: Initial vertex selected (press any key to continue)")
            wait_for_key(plt.gcf())

    while queue:
```

```python
        current = queue.popleft()
        traversal_order.append(current)

        if ax is not None and positions is not None and vertex_circles is not None:
            vertex_circles[current].set_facecolor('red')
            if step_by_step:
                plt.title(f"BFS: Processing vertex {current+1} (press any key to continue)")
                wait_for_key(plt.gcf())

        for neighbor in range(n):
            if adjacency_matrix[current, neighbor] == 1 and not visited[neighbor]:
                queue.append(neighbor)
                visited[neighbor] = True
                parent[neighbor] = current
                traversal_edges.append((current, neighbor))

                if ax is not None and positions is not None and vertex_circles is not None and edge_arrows is not None:
                    vertex_circles[neighbor].set_facecolor('yellow')
                    edge_key = (current, neighbor)
                    if edge_key in edge_arrows:
                        edge_arrows[edge_key].set_color('green')
                        edge_arrows[edge_key].set_linewidth(2.5)
                    if step_by_step:
                        plt.title(f"BFS: Discovered vertex {neighbor+1} (press any key to continue)")
                        wait_for_key(plt.gcf())

        if ax is not None and positions is not None and vertex_circles is not None:
            vertex_circles[current].set_facecolor('lightgreen')
            if step_by_step:
                plt.title(f"BFS: Finished processing vertex {current+1} (press any key to continue)")
                wait_for_key(plt.gcf())

    return visited, parent, traversal_edges, traversal_order

def dfs(adjacency_matrix, start_vertex=None, ax=None, positions=None,
        vertex_circles=None, edge_arrows=None, step_by_step=False):
    """Performs a depth-first search traversal of the graph."""
    n = adjacency_matrix.shape[0]

    if start_vertex is None:
        start_vertex = find_start_vertex(adjacency_matrix)

    visited = [False] * n
    parent = {start_vertex: None}
    traversal_edges = []
```

```
    traversal_order = []

    if ax is not None and positions is not None and vertex_circles is not None:
        vertex_circles[start_vertex].set_facecolor('yellow')
        ax.text(positions[start_vertex][0], positions[start_vertex][1] - 0.8,
f"Start",
                horizontalalignment='center', verticalalignment='center', fontsize=8)
        if step_by_step:
            plt.title("DFS: Initial vertex selected (press any key to continue)")
            wait_for_key(plt.gcf())

    def dfs_recursive(current):
        traversal_order.append(current)
        visited[current] = True

        if ax is not None and positions is not None and vertex_circles is not None:
            vertex_circles[current].set_facecolor('red')
            if step_by_step:
                plt.title(f"DFS: Processing vertex {current+1} (press any key to
continue)")
                wait_for_key(plt.gcf())

        # Try to visit neighbors in numeric order
        for neighbor in range(n):
            if adjacency_matrix[current, neighbor] == 1 and not visited[neighbor]:
                parent[neighbor] = current
                traversal_edges.append((current, neighbor))

                # Update visualization for neighbor and edge
                if ax is not None and positions is not None and vertex_circles is not
None and edge_arrows is not None:
                    edge_key = (current, neighbor)
                    if edge_key in edge_arrows:
                        edge_arrows[edge_key].set_color('green')
                        edge_arrows[edge_key].set_linewidth(2.5)
                    vertex_circles[neighbor].set_facecolor('yellow')
                    if step_by_step:
                        plt.title(f"DFS: Discovered vertex {neighbor+1} (press any key
to continue)")
                        wait_for_key(plt.gcf())

                dfs_recursive(neighbor)

        # Update visualization for visited vertex (processed)
        if ax is not None and positions is not None and vertex_circles is not None:
            vertex_circles[current].set_facecolor('lightgreen')
            if step_by_step:
                plt.title(f"DFS: Finished processing vertex {current+1} (press any key
to continue)")
```

```
                wait_for_key(plt.gcf())

    # Start DFS from the designated vertex
    dfs_recursive(start_vertex)

    return visited, parent, traversal_edges, traversal_order

def complete_graph_traversal(adjacency_matrix, traversal_func, ax=None,
positions=None,
                            vertex_circles=None, edge_arrows=None,
step_by_step=False):
    """Completes a full traversal of the graph, handling disconnected components."""
    n = adjacency_matrix.shape[0]
    all_visited = [False] * n
    forest = []
    all_traversal_order = []

    while False in all_visited:
        # Find the smallest unvisited vertex with outgoing edges
        start_vertex = None
        for i in range(n):
            if not all_visited[i] and np.sum(adjacency_matrix[i]) > 0:
                start_vertex = i
                break

        # If no unvisited vertices with outgoing edges, choose any unvisited vertex
        if start_vertex is None:
            for i in range(n):
                if not all_visited[i]:
                    start_vertex = i
                    break

        if step_by_step:
            plt.title(f"Starting new traversal from vertex {start_vertex+1} (press any
key to continue)")
            wait_for_key(plt.gcf())

        # Run traversal from start vertex
        visited, parent, traversal_edges, traversal_order = traversal_func(
            adjacency_matrix, start_vertex, ax, positions, vertex_circles,
edge_arrows, step_by_step
        )

        # Update all_visited
        for i in range(n):
            all_visited[i] = all_visited[i] or visited[i]

        # Add traversal tree to forest
        if traversal_edges:
```

```
            forest.append(traversal_edges)

        all_traversal_order.extend(traversal_order)

    return forest, all_traversal_order

def draw_traversal_graph(adjacency_matrix, positions, traversal_func_name):
    """Draw graph and perform traversal visualization step by step"""
    n = adjacency_matrix.shape[0]

    fig, ax = plt.subplots(figsize=(12, 10))
    plt.ion()  # Turn on interactive mode

    vertex_circles = {}
    edge_arrows = {}

    # Draw all edges first
    for i in range(n):
        for j in range(n):
            if adjacency_matrix[i, j] == 1:
                if i == j:
                    draw_self_loop(ax, positions[i], color='black', linewidth=1.5,
is_directed=True)
                else:
                    # Store edge object for later coloring
                    arrow = draw_edge(ax, positions[i], positions[j],
is_directed=True, color='blue')
                    edge_arrows[(i, j)] = arrow

    # Draw all vertices
    for i in range(n):
        circle = plt.Circle(positions[i], 0.5, fill=True, facecolor='lightblue',
edgecolor='blue')
        ax.add_patch(circle)
        vertex_circles[i] = circle

        ax.text(positions[i][0], positions[i][1], str(i+1),
horizontalalignment='center',
                verticalalignment='center', fontsize=10, color='black',
fontweight='bold')

    ax.set_aspect('equal')
    margin = 2
    ax.set_xlim(min(positions[:, 0])-margin, max(positions[:, 0])+margin)
    ax.set_ylim(min(positions[:, 1])-margin, max(positions[:, 1])+margin)

    plt.title(f"Graph Traversal - {traversal_func_name}")
    plt.axis('off')
```

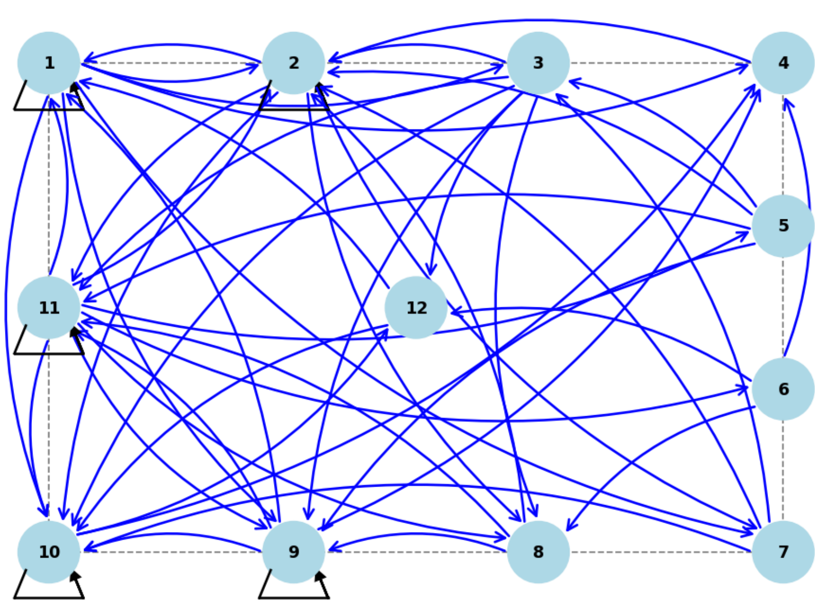
```
    # Add legend
    legend_elements = [
        Patch(facecolor='lightblue', edgecolor='blue', label='Unvisited'),
        Patch(facecolor='yellow', edgecolor='blue', label='Discovered'),
        Patch(facecolor='red', edgecolor='blue', label='Processing'),
        Patch(facecolor='lightgreen', edgecolor='blue', label='Processed')
    ]
    ax.legend(handles=legend_elements, loc='upper right')

    plt.draw()
    plt.pause(1.0)

    # Select traversal function
    if traversal_func_name == "BFS":
        traversal_func = bfs
    else:  # "DFS"
        traversal_func = dfs

    # Run the traversal with visualization
    forest, traversal_order = complete_graph_traversal(
        adjacency_matrix, traversal_func, ax, positions, vertex_circles, edge_arrows,
step_by_step=True
    )

    print(f"\n{traversal_func_name} Traversal Order:", " -> ".join(str(v+1) for v in
traversal_order))

    print("\nTraversal Tree Edges:")
    for i, tree in enumerate(forest):
        print(f"Tree {i+1}:")
        for edge in tree:
            print(f"  {edge[0]+1} -> {edge[1]+1}")

    plt.title(f"Graph Traversal - {traversal_func_name} (Completed)")
    plt.ioff()  # Turn off interactive mode

    return fig, forest, traversal_order
```

Вивід програми:

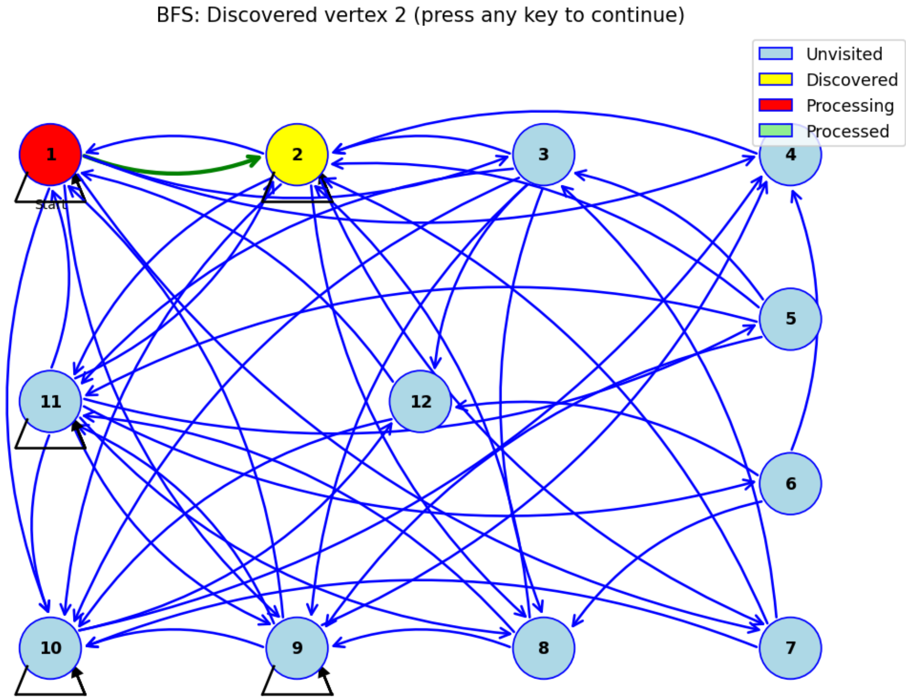
```
Variant number: 4229
Number of vertices n = 10 + 2 = 12

=== PART 5: Graph Traversal Analysis ===
Using k coefficient: 0.7849999999999999

Directed Graph Adjacency Matrix (12x12) for Traversal:
1 1 1 1 0 0 1 0 1 1 0 0
1 1 0 0 0 0 1 1 0 1 1 0
0 1 0 0 0 0 0 1 1 1 1 1
0 1 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 1 0 1 0
0 0 0 1 0 0 0 1 0 0 0 1
0 1 1 0 0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0 1 0 1 0
1 0 0 1 0 0 0 0 1 1 1 0
0 0 0 1 0 0 0 0 0 1 0 1
1 1 0 0 1 1 0 1 1 1 1 0
1 0 0 0 0 0 0 0 0 1 0 0

Starting Breadth-First Search (BFS)...
Press any key in plot window to continue step by step

BFS Traversal Order: 1 -> 2 -> 3 -> 4 -> 7 -> 9 -> 10 -> 8 -> 11 -> 12 -> 5 -> 6

Traversal Tree Edges:
Tree 1:
  1 -> 2
  1 -> 3
  1 -> 4
  1 -> 7
  1 -> 9
  1 -> 10
  2 -> 8
  2 -> 11
  3 -> 12
  11 -> 5
  11 -> 6
```

```
Starting Depth-First Search (DFS)...
Press any key in plot window to continue step by step

DFS Traversal Order: 1 -> 2 -> 7 -> 3 -> 8 -> 9 -> 4 -> 10 -> 12 -> 11 -> 5 -> 6

Traversal Tree Edges:
Tree 1:
  1 -> 2
  2 -> 7
  7 -> 3
  3 -> 8
  8 -> 9
  9 -> 4
  9 -> 10
  10 -> 12
  9 -> 11
  11 -> 5
  11 -> 6

Traversal Analysis:
BFS visited 12 vertices
DFS visited 12 vertices
BFS created 1 tree(s) with a total of 11 edges
DFS created 1 tree(s) with a total of 11 edges
```

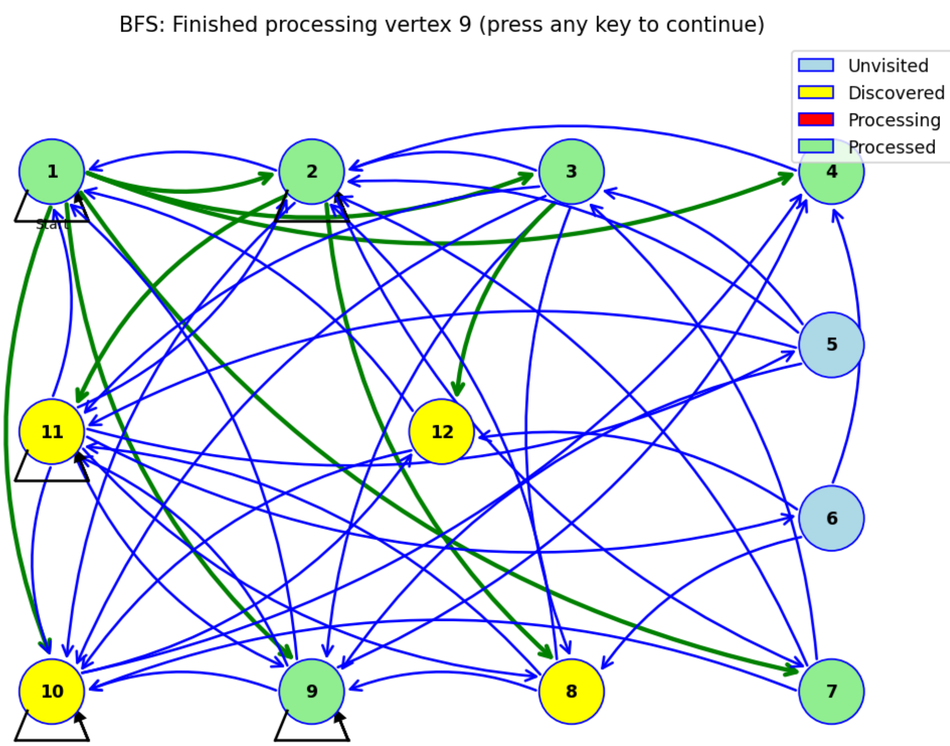
Graph for Traversal - Directed Graph - 12 vertices



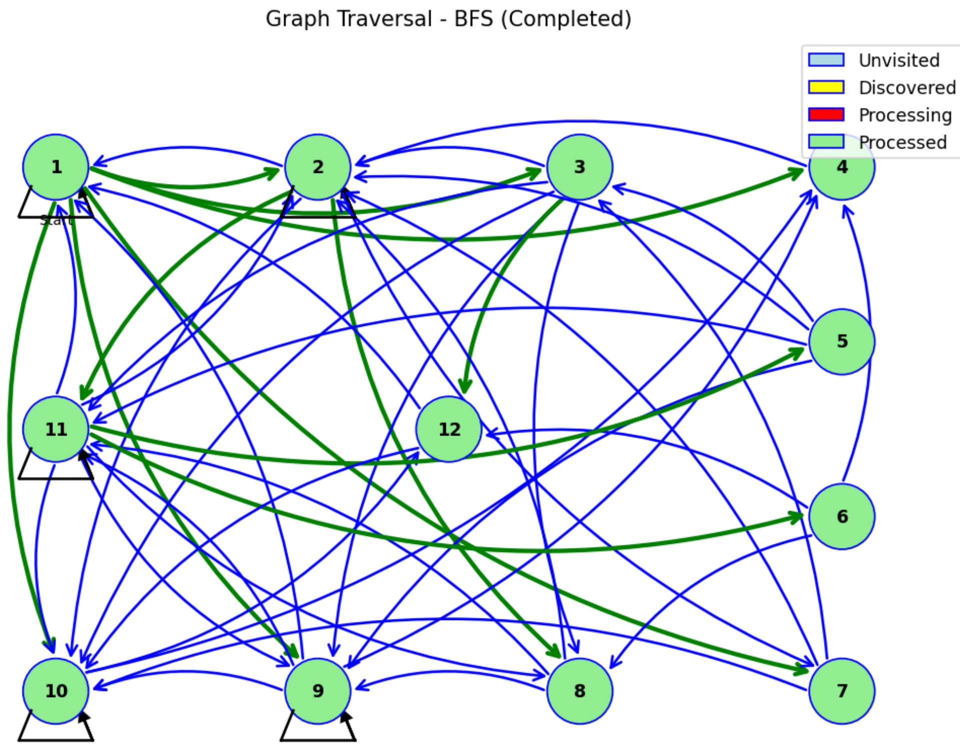
На початку:



В середині:

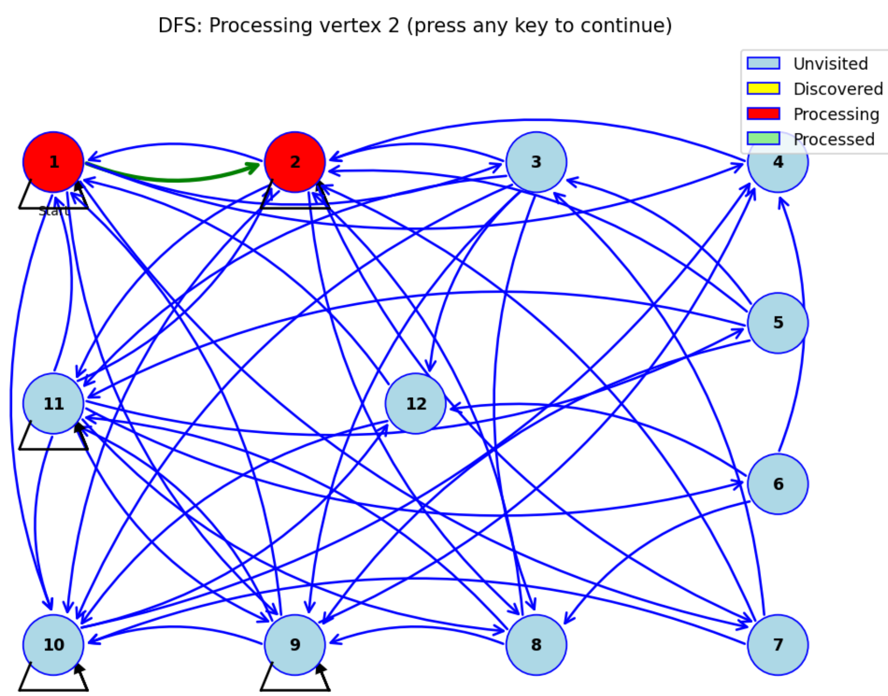


Наприкінці:

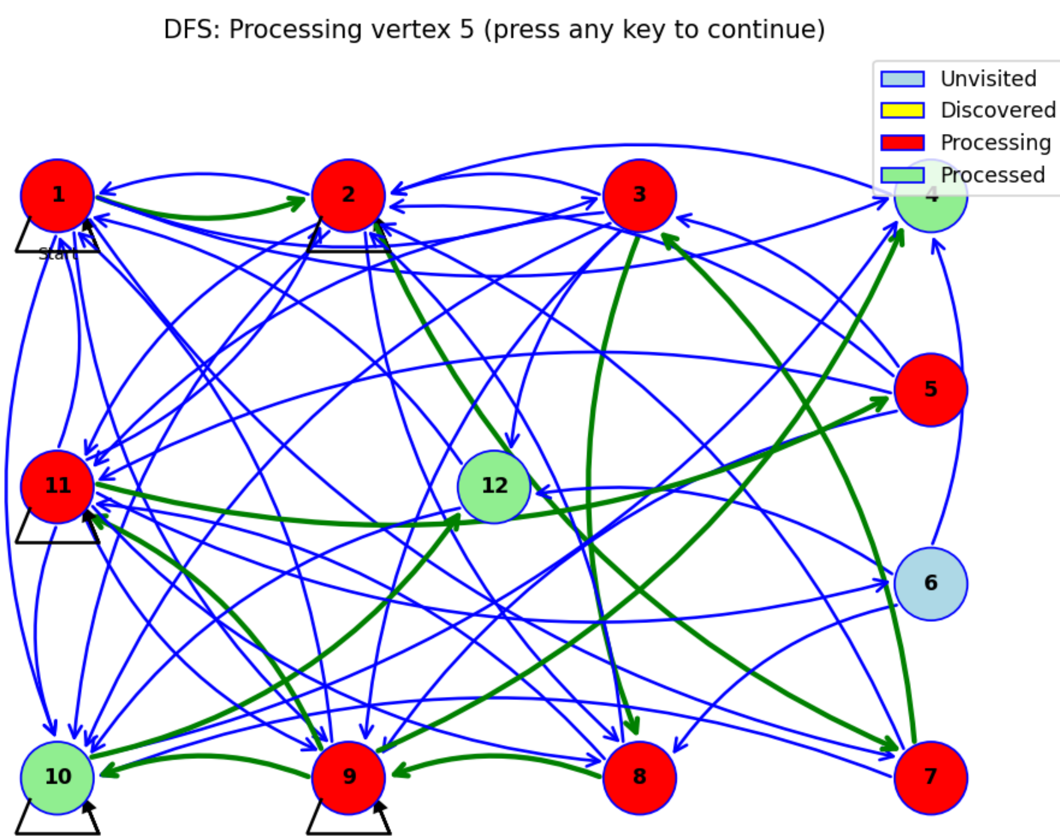


Graph Traversal - BFS (Completed)
Unvisited
Discovered
Processing
Processed
1
2
3
4
5
6
7
8
9
10
11
12
Start

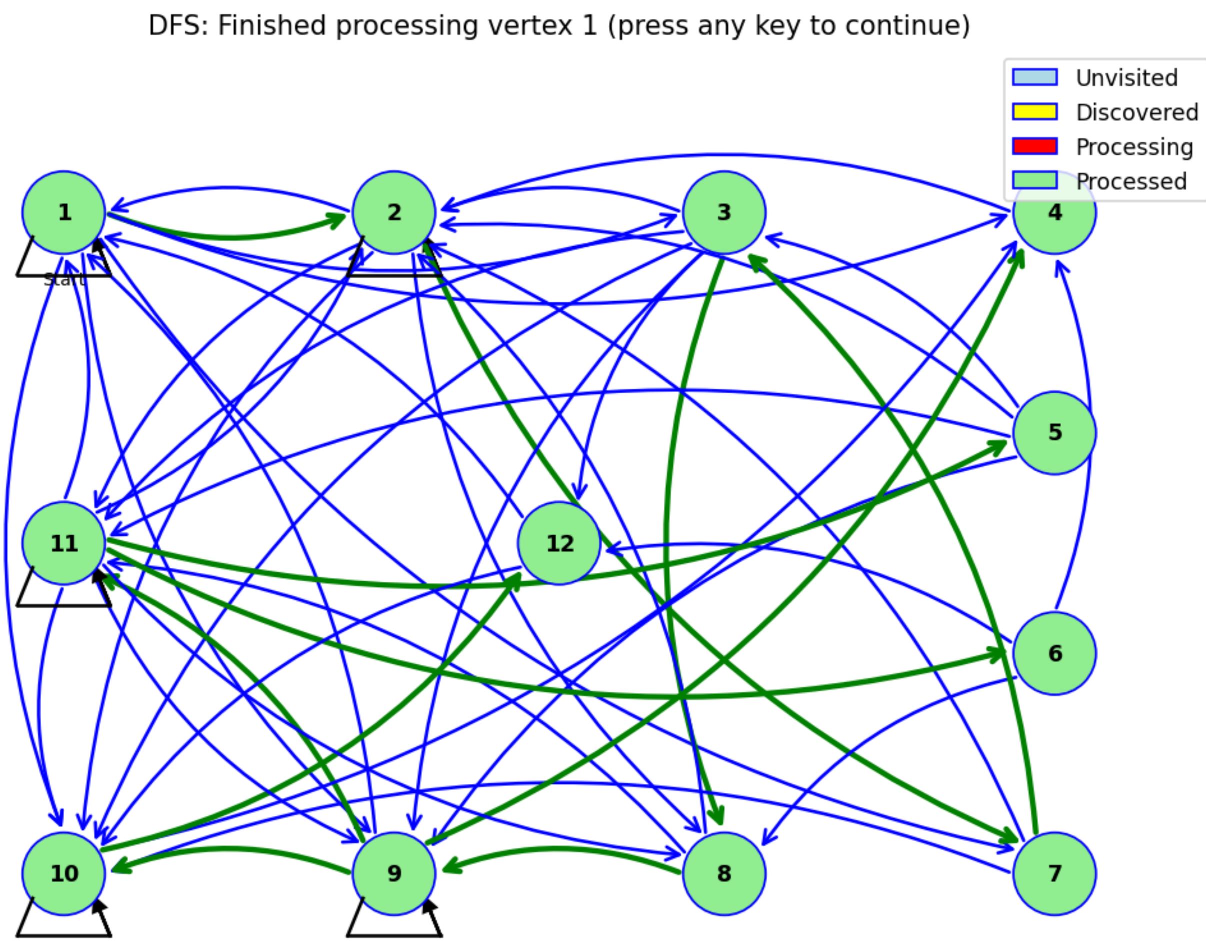
На початку:



В середині:



Наприкінці:



Висновки:

Було успішно реалізував програму для візуалізації та аналізу алгоритмів обходу графа. Побудував напрямлений граф, реалізував алгоритми обходу графа вшир (BFS) та вглиб (DFS). Процеси обходу візуалізував.