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 №1

з дисципліни
«Алгоритми і структури даних»

Виконала:

студентка групи ІМ - 43
Хоанг Чан Кам Лі
номер у списку групи: 29

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 напрямленого графів у графічному вікні.

Варіант 29:
Номер варіанту: 4329

Кількість вершин n = 10 + 2 = 12

Розміщення вершин: квадратом (прямокутником) з вершиною в центрі, бо n4 = 9.

Коефіцієнт k = 0.665

Текст програми

graph_generation.py

```
import numpy as np


def generate_matrices(SEED=4329, NUM_NODES=12):
    np.random.seed(SEED)
    random_matrix = 2.0 * np.random.rand(NUM_NODES, NUM_NODES)


    coeff_k = 1.0 - int(str(SEED)[2]) * 0.02 - int(str(SEED)[3]) * 0.005 - 0.25
    adjacency_matrix = (random_matrix * coeff_k >= 1.0).astype(int)


    adjacency_matrix_undir = adjacency_matrix + adjacency_matrix.T
```

```
    adjacency_matrix_undir[adjacency_matrix_undir > 1] =
1


    return adjacency_matrix, adjacency_matrix_undir


if __name__ == "__main__":
    adj_matrix_dir, adj_matrix_undir =
generate_matrices()
    print("Directed Adjacency Matrix:\n",
adj_matrix_dir)
    print("\nUndirected Adjacency Matrix:\n",
adj_matrix_undir)
```

draw.py

```
def draw_graph(adj_matrix, directed):
    import numpy as np
    import matplotlib.pyplot as plt


    NODE_RADIUS = 0.05
    LINE_THICKNESS = 1.5


    NUM_NODES = adj_matrix.shape[0]


    center_node = np.array([[1, 1]])


    num_boundary_nodes = NUM_NODES - 1


    nodes_per_side = num_boundary_nodes // 4
```

```
    # Calculate the remainder nodes
    remainder_nodes = num_boundary_nodes % 4

    boundary_nodes = []

    # Distribute nodes to sides
    sides = [0, 1, 2, 3] # 0: bottom, 1: right, 2: top,
3: left
    side_node_counts = [nodes_per_side] * 4

    # Distribute remainder nodes
    for i in range(remainder_nodes):
      side_node_counts[i] += 1

    # Place nodes on each side
    for side, count in zip(sides, side_node_counts):
      if count > 0:
        # Calculate spacing between nodes (between 0.5
and 1.5)
        spacing = np.linspace(0.3, 1.7, count)

        if side == 0: # bottom side (y=0)
          boundary_nodes.extend([[x, 0] for x in
spacing])
        elif side == 1: # right side (x=2)
```

```
          boundary_nodes.extend([[2, y] for y in
spacing])
        elif side == 2: # top side (y=2)
          boundary_nodes.extend([[x, 2] for x in
spacing])
        elif side == 3: # left side (x=0)
          boundary_nodes.extend([[0, y] for y in
spacing])


    all_nodes = np.array(boundary_nodes)
    # Combine center and boundary nodes, if center node
exists
    if NUM_NODES > 0:
      all_nodes = np.vstack((center_node, all_nodes))



    plt.figure(figsize=(8, 8)) # Increased figure size


    # Draw edges first
    focus_point = np.array([1, 1])


    if directed:
      title = 'Directed Graph Representation'
      for i in range(NUM_NODES):
        for j in range(NUM_NODES):
          if adj_matrix[i, j] == 1:
            node1 = all_nodes[i]
```

```
            node2 = all_nodes[j]


            if i == 0 or j == 0: # If either node is the
center node (index 0)
              plt.arrow(node1[0], node1[1], node2[0] -
node1[0], node2[1] - node1[1],
                        head_width=0.05,
head_length=0.08, fc='k', ec='k',
linewidth=LINE_THICKNESS, length_includes_head=True)
            else:
              # Draw a quadratic Bezier curve with arrow
              midpoint = (node1 + node2) / 2
              control_point = midpoint * 0.3 +
focus_point * 0.7 # Adjust weights for desired curve
shape


              t = np.linspace(0, 1, 100)
              # Bezier curve formula: B(t) = (1-t)^2 *
P0 + 2*(1-t)*t * P1 + t^2 * P2
              curve_x = (1-t)**2 * node1[0] + 2*(1-t)*t
* control_point[0] + t**2 * node2[0]
              curve_y = (1-t)**2 * node1[1] + 2*(1-t)*t
* control_point[1] + t**2 * node2[1]


              plt.plot(curve_x, curve_y, 'r-',
linewidth=LINE_THICKNESS)


              # Add arrow head near the end of the curve
```

```
            # Find the position and direction at the
end of the curve
            end_point_idx = -2 # Get a point near the
end
            arrow_point =
np.array([curve_x[end_point_idx],
curve_y[end_point_idx]])
            direction = np.array([curve_x[-1] -
curve_x[end_point_idx], curve_y[-1] -
curve_y[end_point_idx]])
            direction = direction /
np.linalg.norm(direction) # Normalize direction vector


            # Position the arrow head slightly before
the end point
            arrow_base = np.array([curve_x[-1],
curve_y[-1]]) - direction * NODE_RADIUS * 1.4
            plt.arrow(arrow_base[0], arrow_base[1],
direction[0] * 0.01, direction[1] * 0.01, # Draw a small
arrow pointing in the direction
                      head_width=0.05,
head_length=0.08, fc='r', ec='r', linewidth=0,
length_includes_head=True)


    else: # Undirected graph
      title = 'Undirected Graph Representation'
      adj_matrix_undir = adj_matrix + adj_matrix.T
      adj_matrix_undir[adj_matrix_undir > 1] = 1
```

```
      for i in range(NUM_NODES):
        for j in range(i + 1, NUM_NODES): # Iterate
through the upper triangle to avoid duplicate lines
          if adj_matrix_undir[i, j] == 1:
            node1 = all_nodes[i]
            node2 = all_nodes[j]

            if i == 0 or j == 0: # If either node is the
center node (index 0)
              plt.plot([node1[0], node2[0]], [node1[1],
node2[1]], 'k-', linewidth=LINE_THICKNESS)
            else:
              # Draw a quadratic Bezier curve
              midpoint = (node1 + node2) / 2
              control_point = midpoint * 0.3 +
focus_point * 0.7
              t = np.linspace(0, 1, 100)

              curve_x = (1-t)**2 * node1[0] + 2*(1-t)*t
* control_point[0] + t**2 * node2[0]
              curve_y = (1-t)**2 * node1[1] + 2*(1-t)*t
* control_point[1] + t**2 * node2[1]
              plt.plot(curve_x, curve_y, 'r-',
linewidth=LINE_THICKNESS)



    # Draw nodes as circles with numbers
```

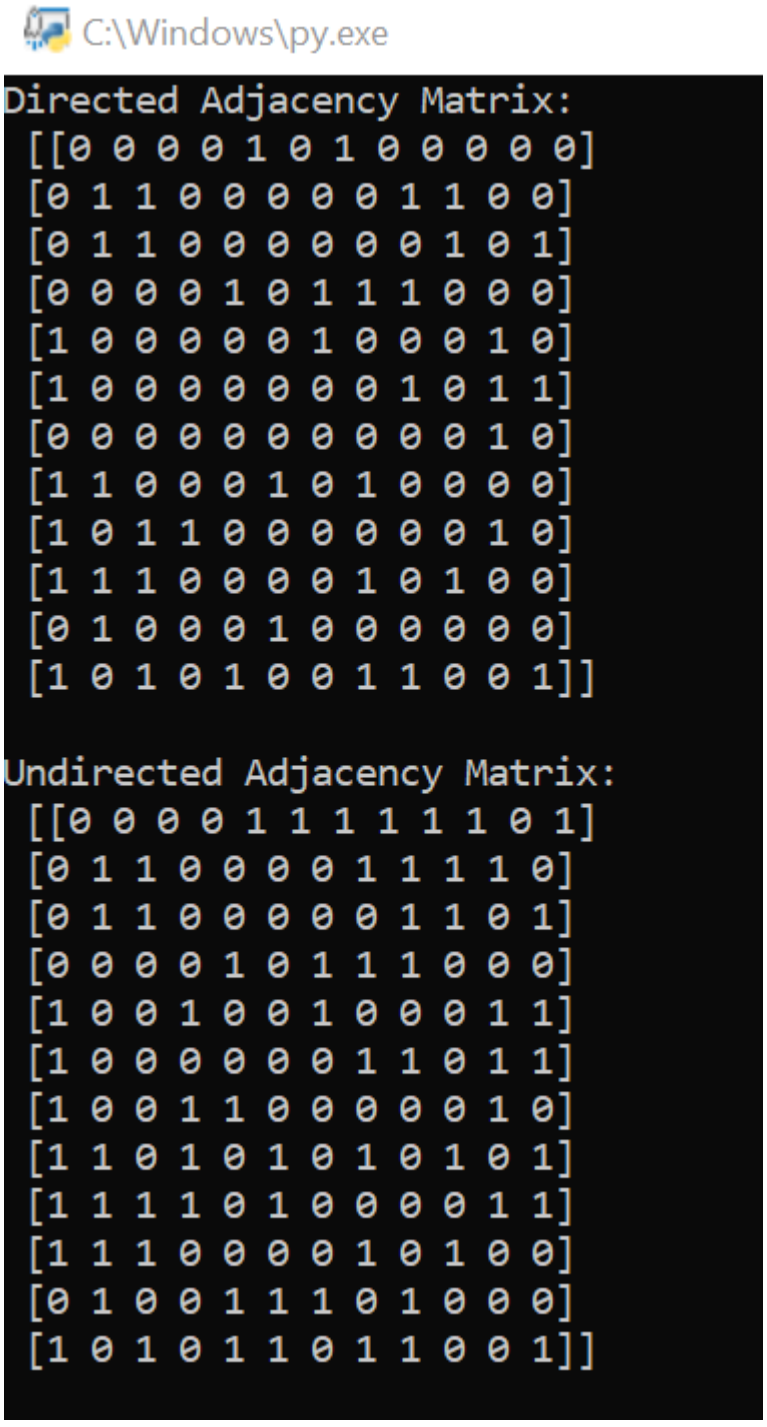
```
    for i, (x, y) in enumerate(all_nodes):
        circle = plt.Circle((x, y), NODE_RADIUS,
color='skyblue', zorder=5)
        plt.gca().add_patch(circle)
        plt.text(x, y, str(i), ha='center', va='center',
color='black', fontsize=8, zorder=6)
    # Remove axis
    plt.axis('off')
    # Set plot limits and aspect ratio
    plt.xlim(-0.2, 2.2)
    plt.ylim(-0.2, 2.2)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.title(title)


    plt.show()
```

main.py

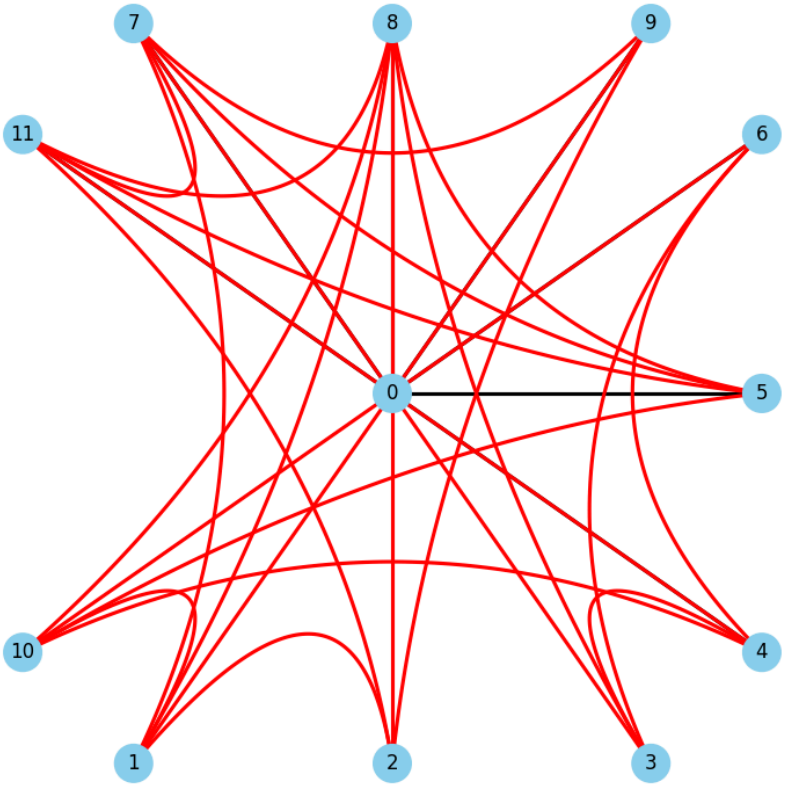
```
import graph_generation
import draw


adj_matrix_dir, adj_matrix_undir =
graph_generation.generate_matrices()


print("Directed Adjacency Matrix:\n", adj_matrix_dir)
print("\nUndirected Adjacency Matrix:\n",
adj_matrix_undir)


draw.draw_graph(adj_matrix_undir, directed=False)
```

```
draw.draw_graph(adj_matrix_dir, directed=True)
```

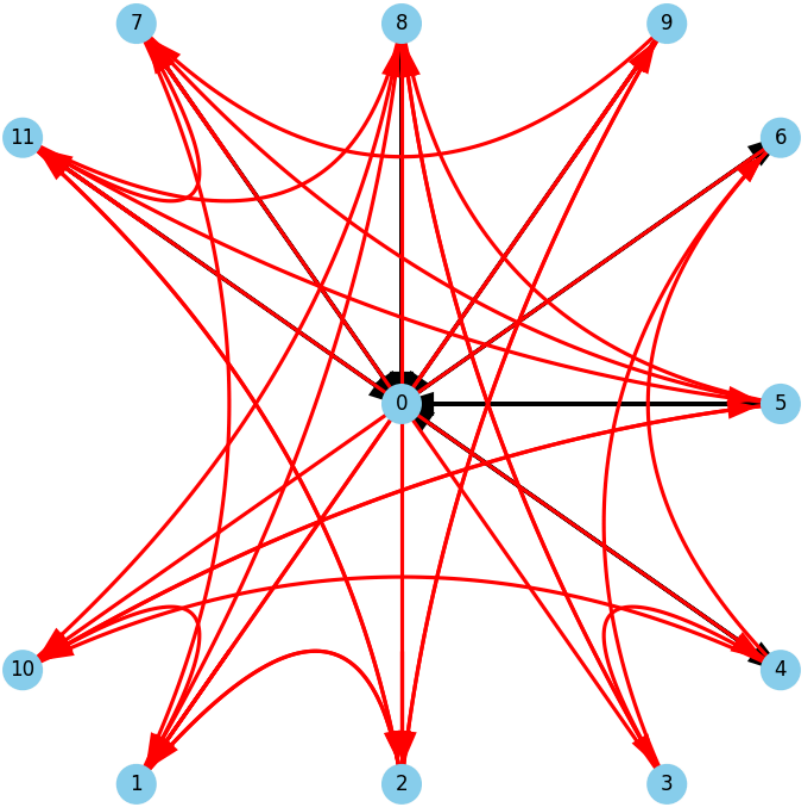
Результати тестування програми та перевірки



Undirected Graph Representation



Directed Graph Representation



Висновки

Лабораторна робота була присвячена генерації та візуалізації графів, що є важливою складовою математичного моделювання та аналізу структур даних. Графи широко використовуються у багатьох сферах, від комп'ютерних наук до логістики, транспортних систем та аналізу соціальних мереж. Саме тому їх правильне представлення та обробка мають велике значення.

У ході виконання завдання була реалізована система для автоматичного створення матриць суміжності, яка забезпечує коректну взаємодію між вершинами. Використання випадкових чисел у процесі генерації дало можливість моделювати різні варіанти зв'язків. Окремо було розглянуто два типи графів - напрямлений та ненапрямлений. Це дозволило дослідити відмінності між ними та правильність їх математичного опису.

Після побудови матриць суміжності був реалізований алгоритм розташування вершин на площині. Це дало змогу створити графічне представлення структури графа, де всі вузли та зв'язки були правильно відображені. У випадку ненапрямленого графа зв'язки між вершинами були показані як прямі або вигнуті лінії, а в напрямленому графі додатково використовувалися стрілки для позначення спрямованості зв'язків.

На основі отриманих результатів було зроблено висновок, що алгоритми обробки графів дозволяють ефективно аналізувати складні мережі та моделювати зв'язки між елементами. Використання бібліотек `numpy` та `matplotlib` значно полегшує роботу з графами, спрощуючи генерацію даних і візуалізацію. Правильне програмне розбиття на модулі забезпечило зручність використання та читабельність коду, що важливо для подальшого вдосконалення роботи.

Загалом лабораторна робота продемонструвала практичне застосування теорії графів та можливості їх використання в аналізі даних і розробці алгоритмів. Отримані знання можна застосовувати у широкому спектрі завдань, зокрема в оптимізації процесів, моделюванні зв'язків у великих системах та розробці ефективних структур для роботи з даними. Робота не тільки допомогла глибше зрозуміти принципи побудови графів, а й дала можливість закріпити практичні навички програмування та аналізу складних структур.