

# **GRAFY I ICH ZASTOSOWANIA**

***Rozwiązania zestawów 1-3***  
***grupa "Cynamonki"***



**Zestaw 1**

**Yuliya Zviarko**



# **zadanie 1**

Zdefiniowałem trzy sposoby reprezentacji grafu:

**listę sąsiedztwa,**

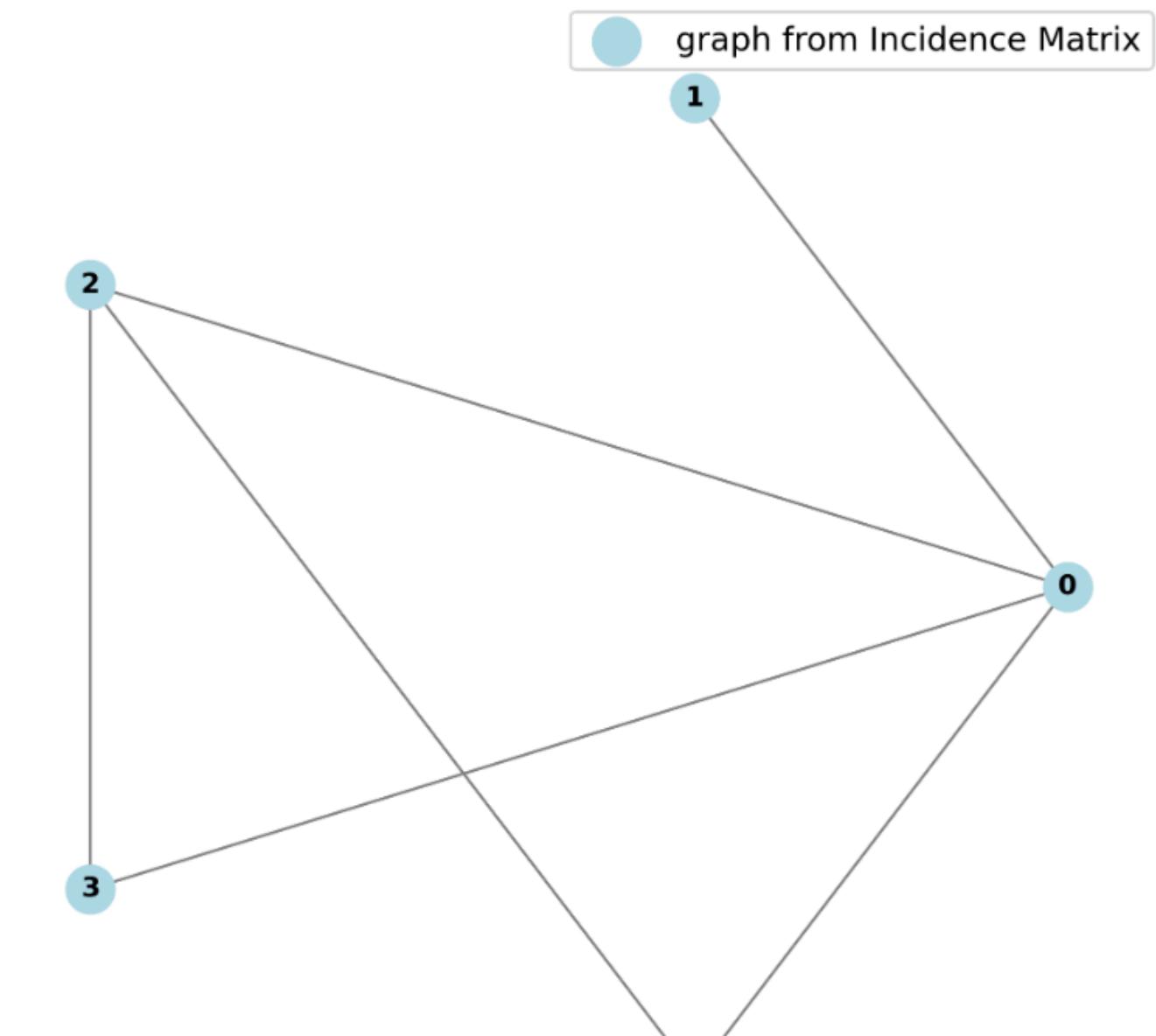
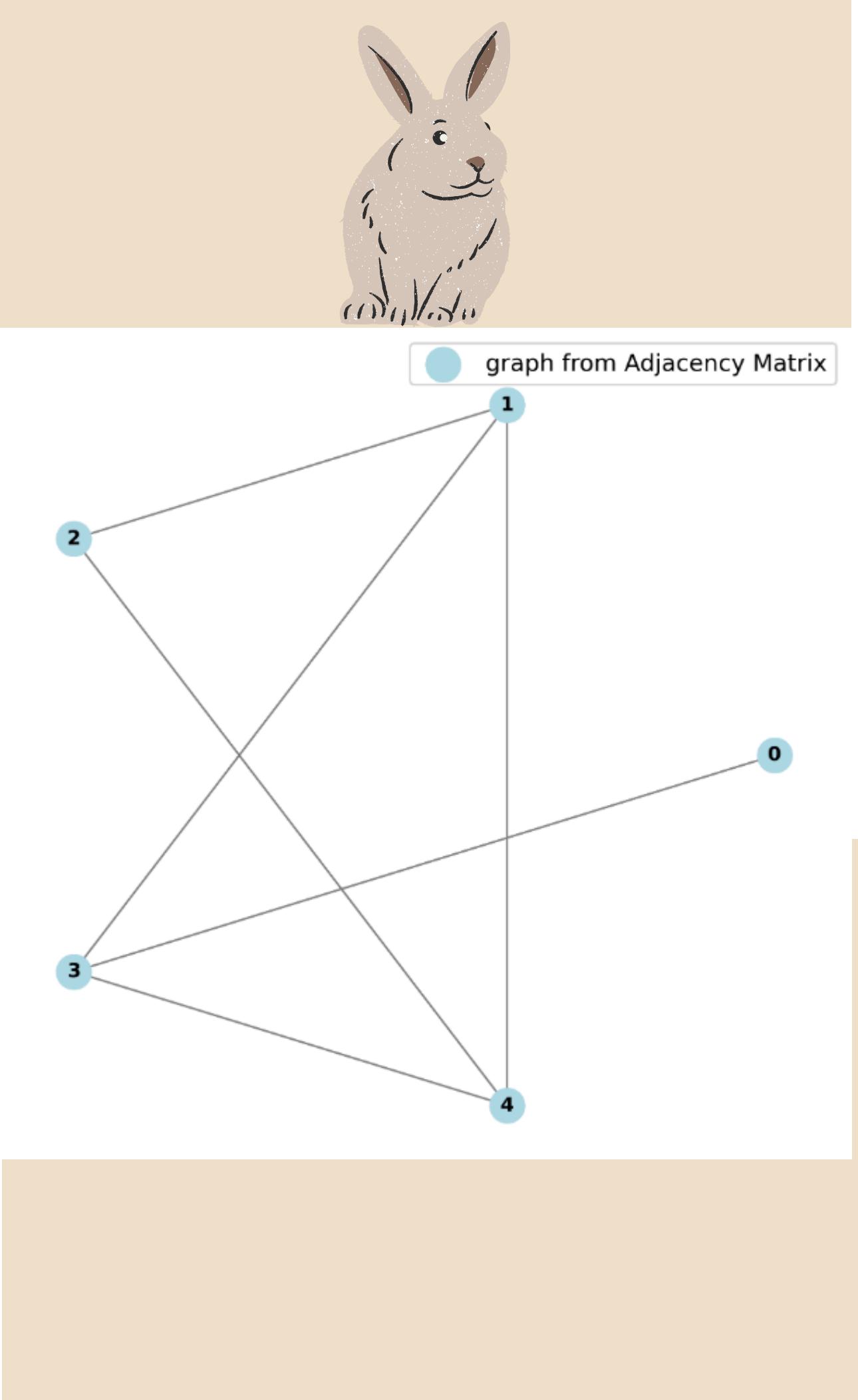
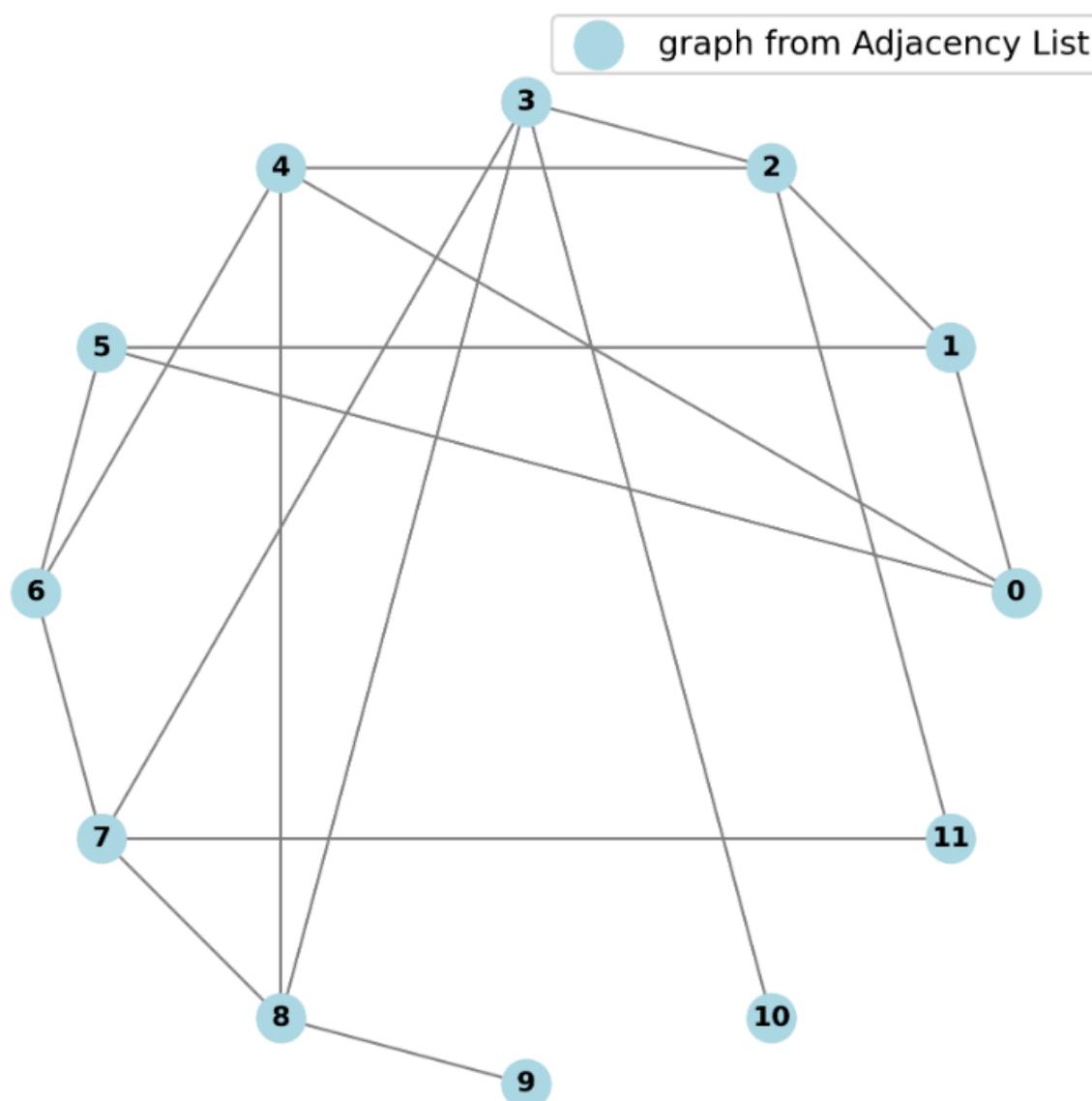
**macierz sąsiedztwa**

**oraz macierz incydencji,**

dziedziczące po klasie bazowej Graph. Każda reprezentacja zawiera własne metody konwersji do pozostałych form oraz umożliwia wczytywanie danych z pliku. Wspólne funkcjonalności, takie jak dodawanie wag czy generowanie tekstowej reprezentacji grafu, zaimplementowałem w klasie bazowej.



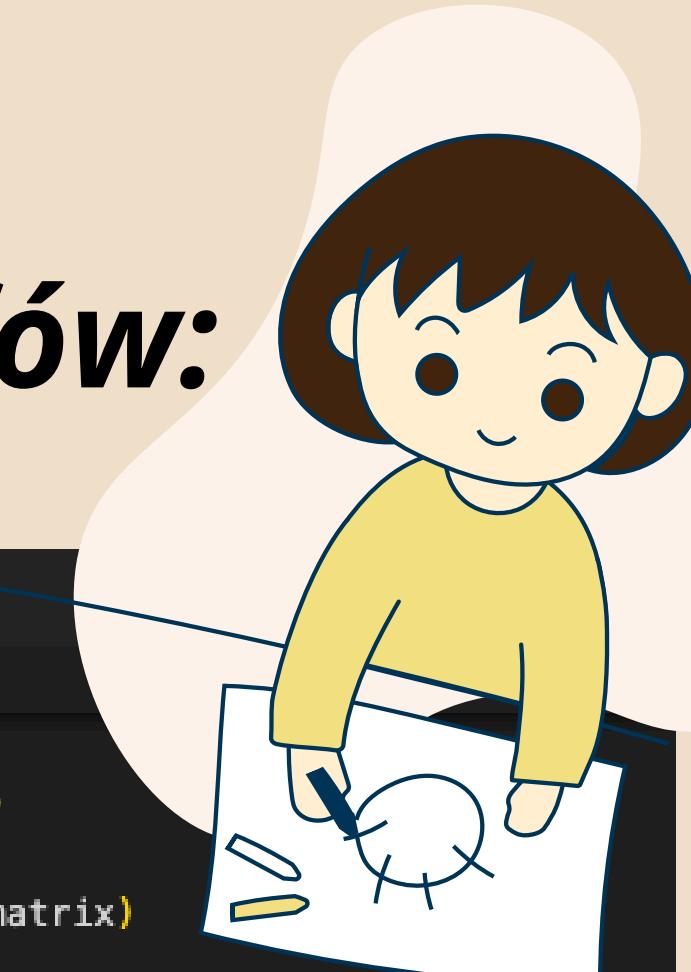
# zadanie 1



Zostawię parę wykresów, które otrzymałem w trakcie przekształceń między różnymi reprezentacjami grafu. Pokazuję one, jak zmienia się struktura danych przy konwersji z jednej formy na inną. Dzięki nim łatwiej zobaczyć, że mimo różnych formatów, graf pozostaje ten sam pod względem logicznym.

# zadanie 2

## *Wizualizacja grafów:*



```
DrawGraph.py ×
WFIS-GilZ-lab > DrawGraph.py

15
16     adjacency_matrix = np.array(graph.data)
17     try:
18         G = nx.from_numpy_array(adjacency_matrix)
19         plt.figure(figsize=(6, 6))
20
21         n = len(G.nodes)
22         r = 1
23         x0, y0 = 0, 0
24
25         alpha = 2 * np.pi / n
26         pos = {} # pozycje wierzchołków w układzie współrzędnych 2D
27
28         # i - każdy wierzchołek w grafie
29         for i, node in enumerate(G.nodes):
30             xi = x0 + r * np.cos(i * alpha)
31             yi = y0 + r * np.sin(i * alpha)
32             pos[node] = (xi, yi)
33
34         nx.draw(G, pos, with_labels=True, node_color="lightblue", edge_col
```

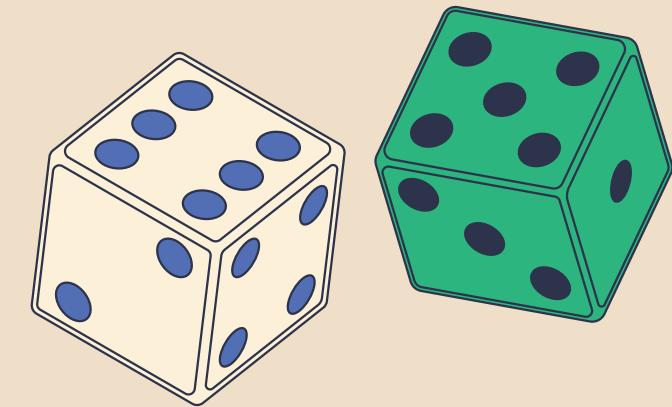
Rysowanie grafów odbywało się przy użyciu biblioteki networkx, która umożliwia tworzenie wizualizacji na podstawie macierzy sąsiedztwa.

Wierzchołki rozmieszczano na okręgu, co zapewnia czytelną i symetryczną prezentację relacji między nimi.

Dodatkowo możliwe było oznaczenie wag krawędzi oraz wyróżnienie krawędzi należących do drzewa rozpinającego (MST).

Działanie funkcji rysującej zostało zaprezentowane w zadaniu 1 i będzie również wykorzystywane w zadaniu 3.

# zadanie 3 - losowanie grafów



```
...  
Napisać program do generowania grafów losowych  $G(n, l)$  oraz  $G(n, p)$ .  
...  
# 1)  $G(n, l)$   
random_graph1 = generate_random_graph_by_edges(7, 10)  
print(random_graph1)  
Draw(random_graph1, "random_graph1", "7, 10 - random graph", "outputs/01")  
  
# 2)  $G(n, p)$   
random_graph2 = random_graph_by_probability(7, 0.5)  
print(random_graph2)  
Draw(random_graph2, "random_graph2", "7, 0.5 - random graph", "outputs/01")
```

- W przypadku  $G(n, k)$  losujemy dokładnie  $k$  unikalnych krawędzi spośród wszystkich możliwych par wierzchołków.
- W przypadku  $G(n, p)$  przechodzimy przez każdą możliwą parę wierzchołków i z prawdopodobieństwem  $p$  dodajemy między nimi krawędź. W obu przypadkach grafy są nieskierowane i reprezentowane jako lista sąsiedztwa.

# Zadanie 3 - Graf losowy $G(n,k)$

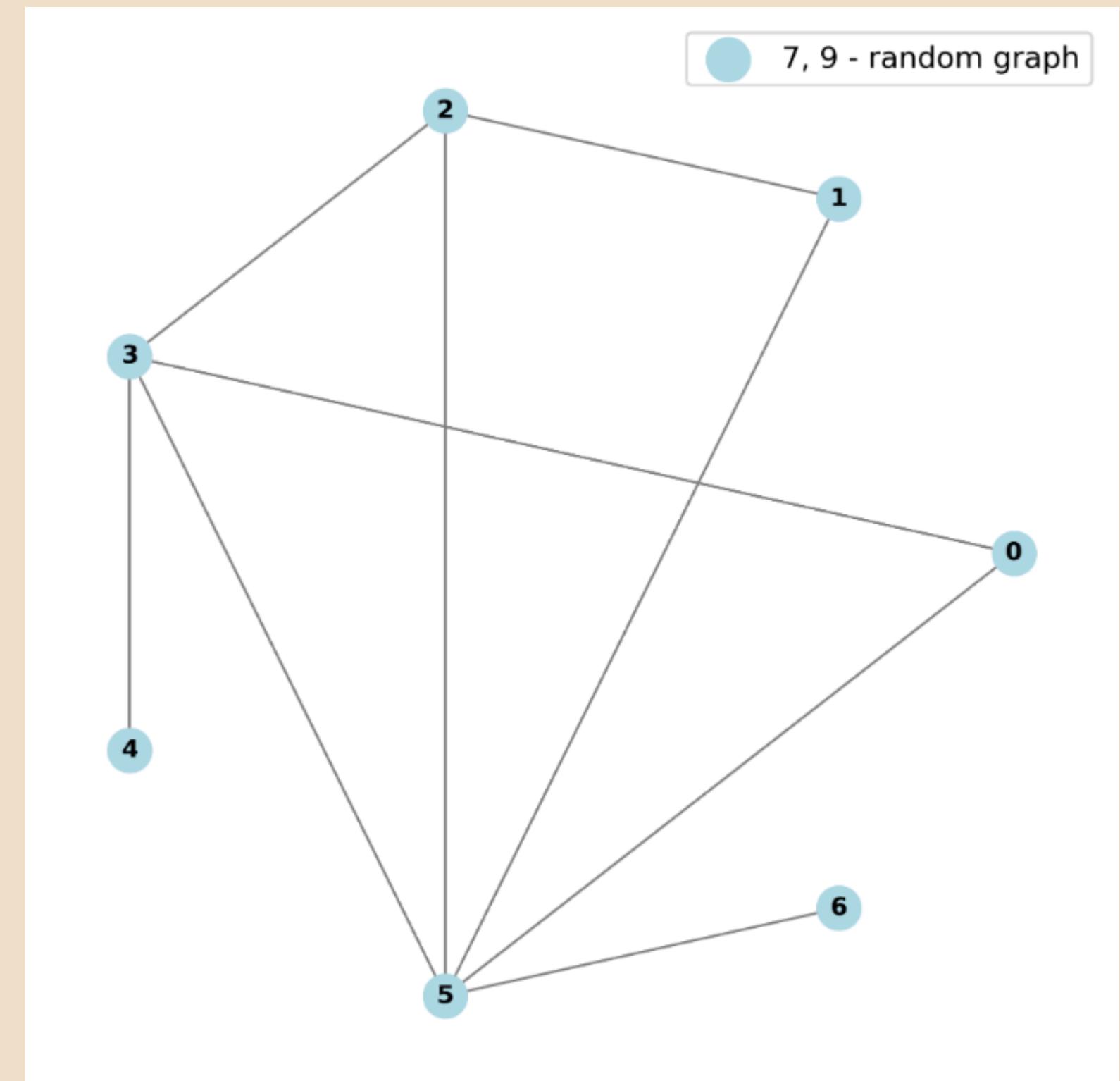
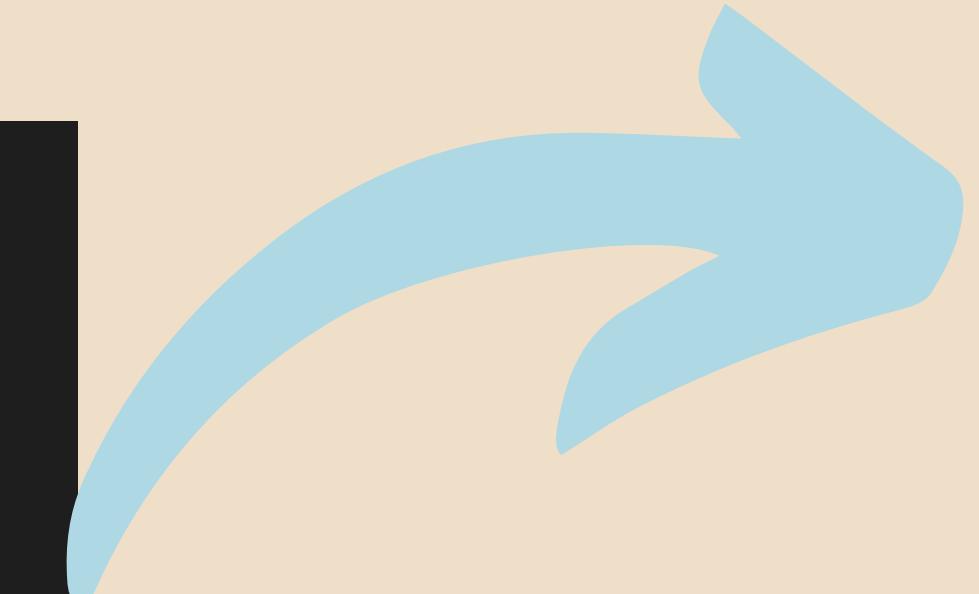
Przykładowy output i wykres dla grafu losowego o parametrach:

**7 wierzchołków, 9 krawędzi**

Na podstawie listy sąsiedztwa wygenerowano wykres, w którym relacje między wierzchołkami odpowiadają danym w grafie.

Zgodność grafu i wizualizacji potwierdza poprawność działania algorytmu generującego.

```
N: 7 L: 9
0: [3, 5]
1: [5, 2]
2: [3, 1, 5]
3: [5, 0, 2, 4]
4: [3]
5: [3, 1, 6, 0, 2]
6: [5]
```

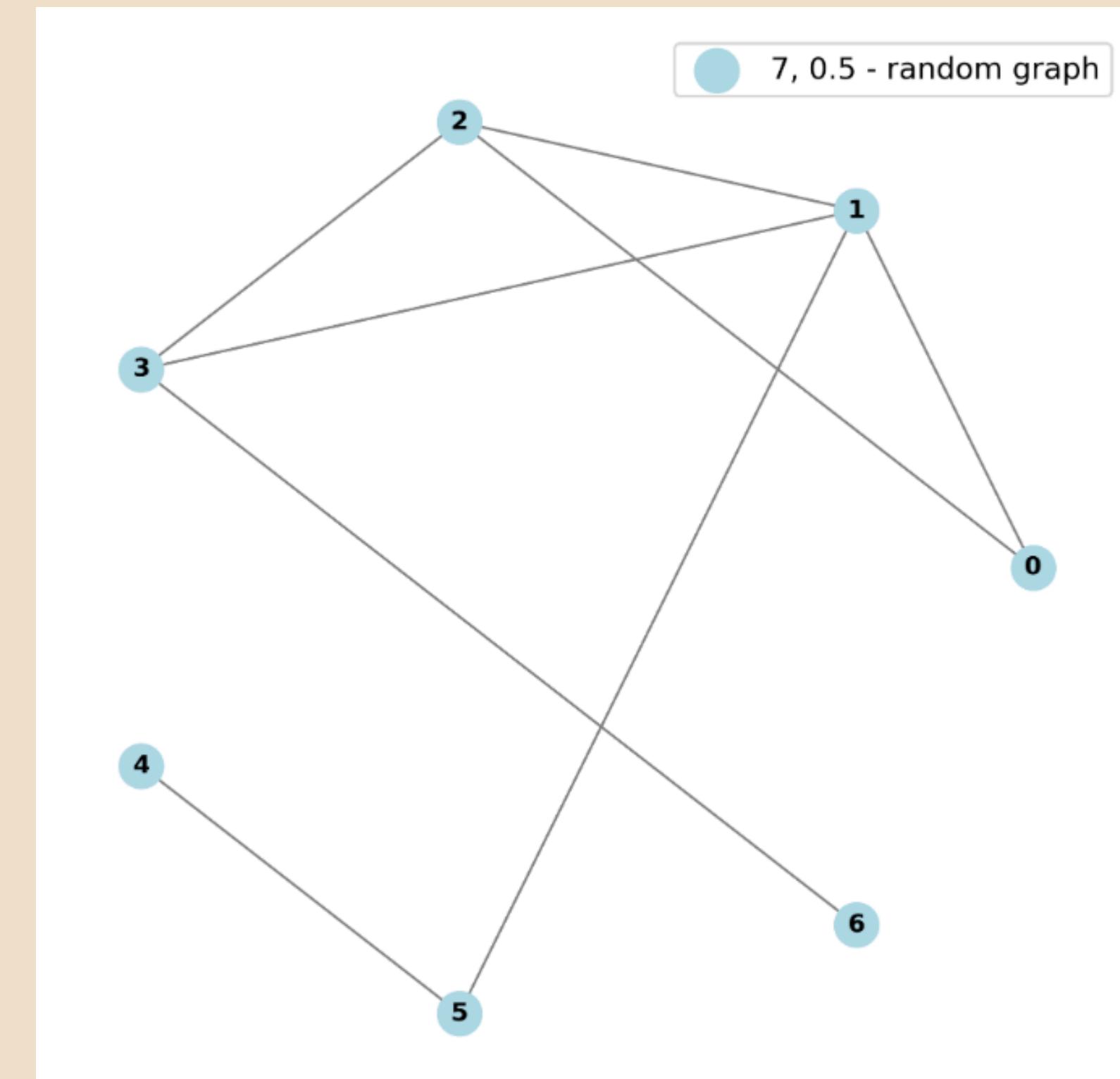
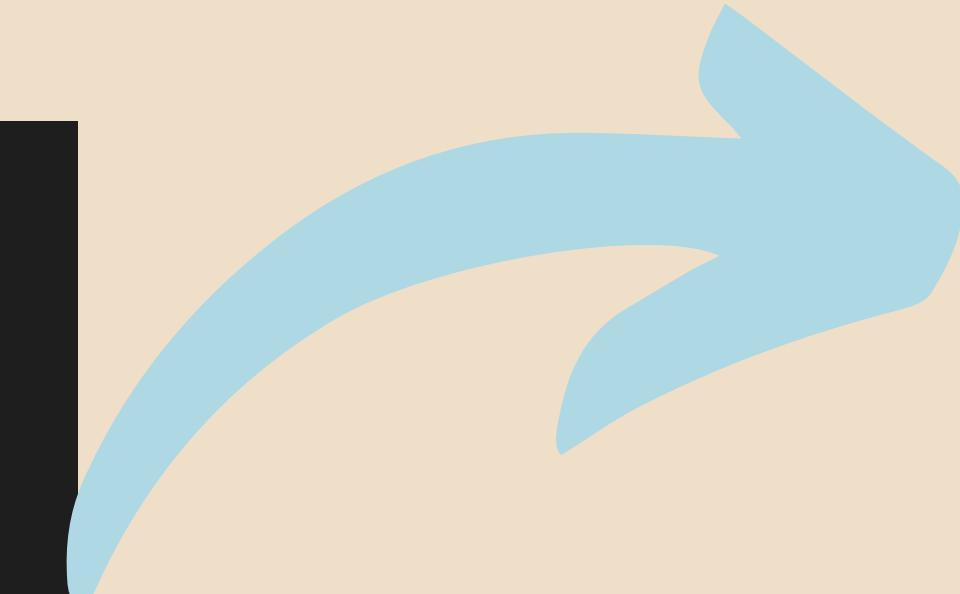


# Zadanie 3 - Graf losowy $G(n,p)$

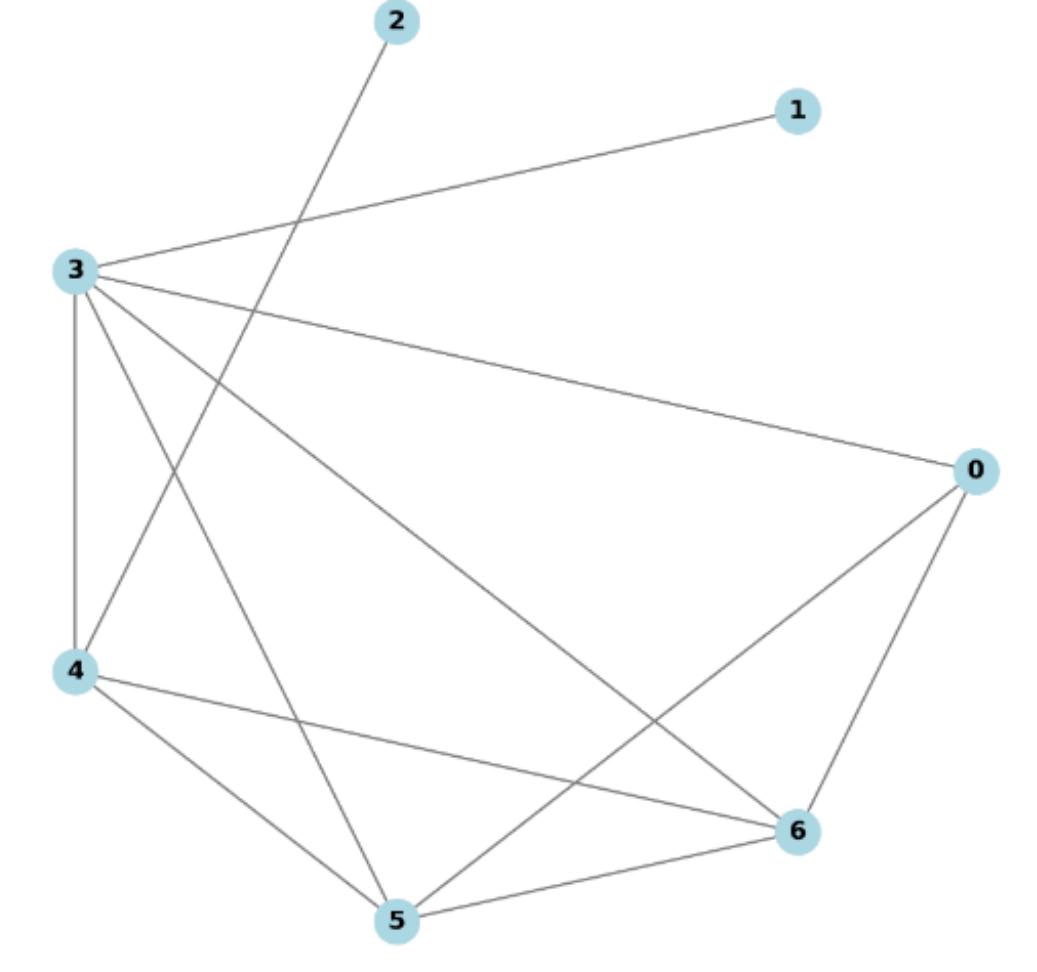
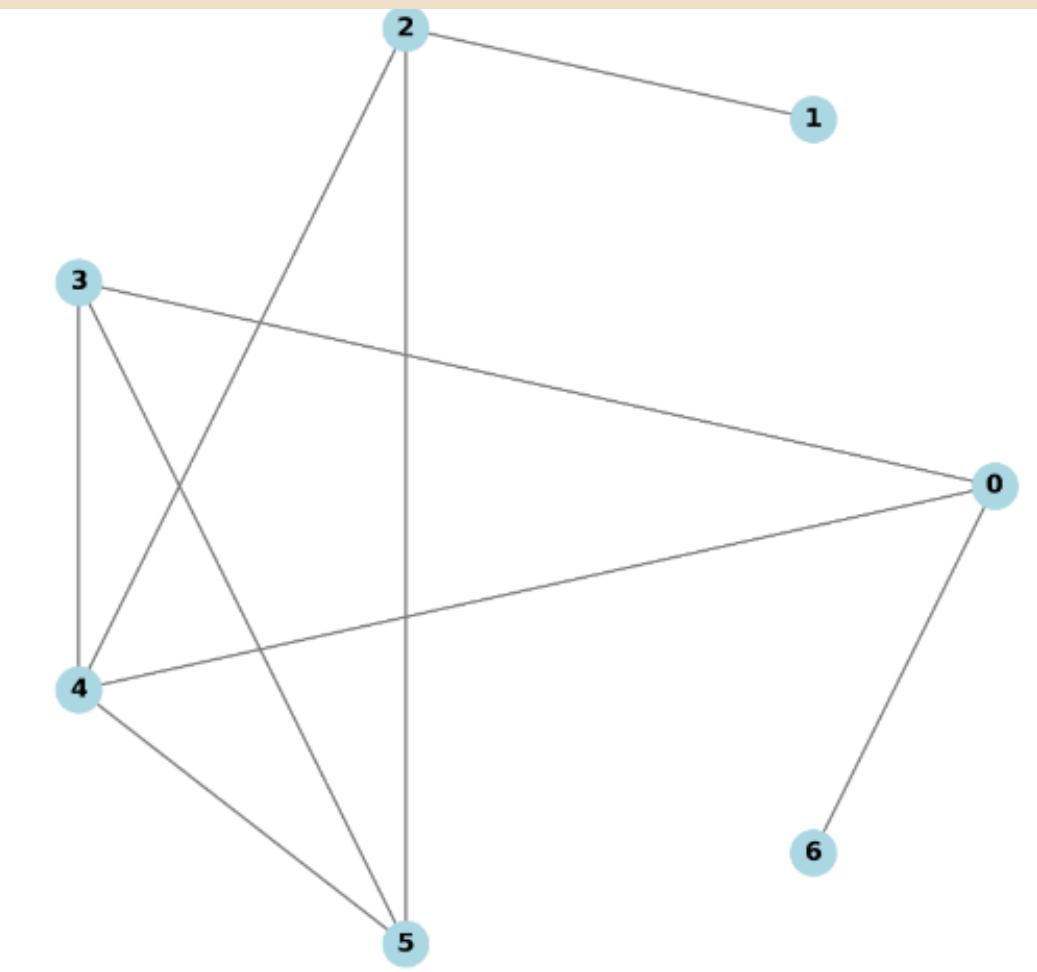
Relacje z listy sąsiedztwa i wykresu są identyczne.

Dla  $p = 0.5$  wygenerowano graf, w którym każda możliwa krawędź miała 50% szansy na wystąpienie, co skutkowało umiarkowaną gęstością połączeń między wierzchołkami.

```
N: 7 L: 8
0: [1, 2]
1: [0, 2, 3, 5]
2: [0, 1, 3]
3: [1, 2, 6]
4: [5]
5: [1, 4]
6: [3]
```

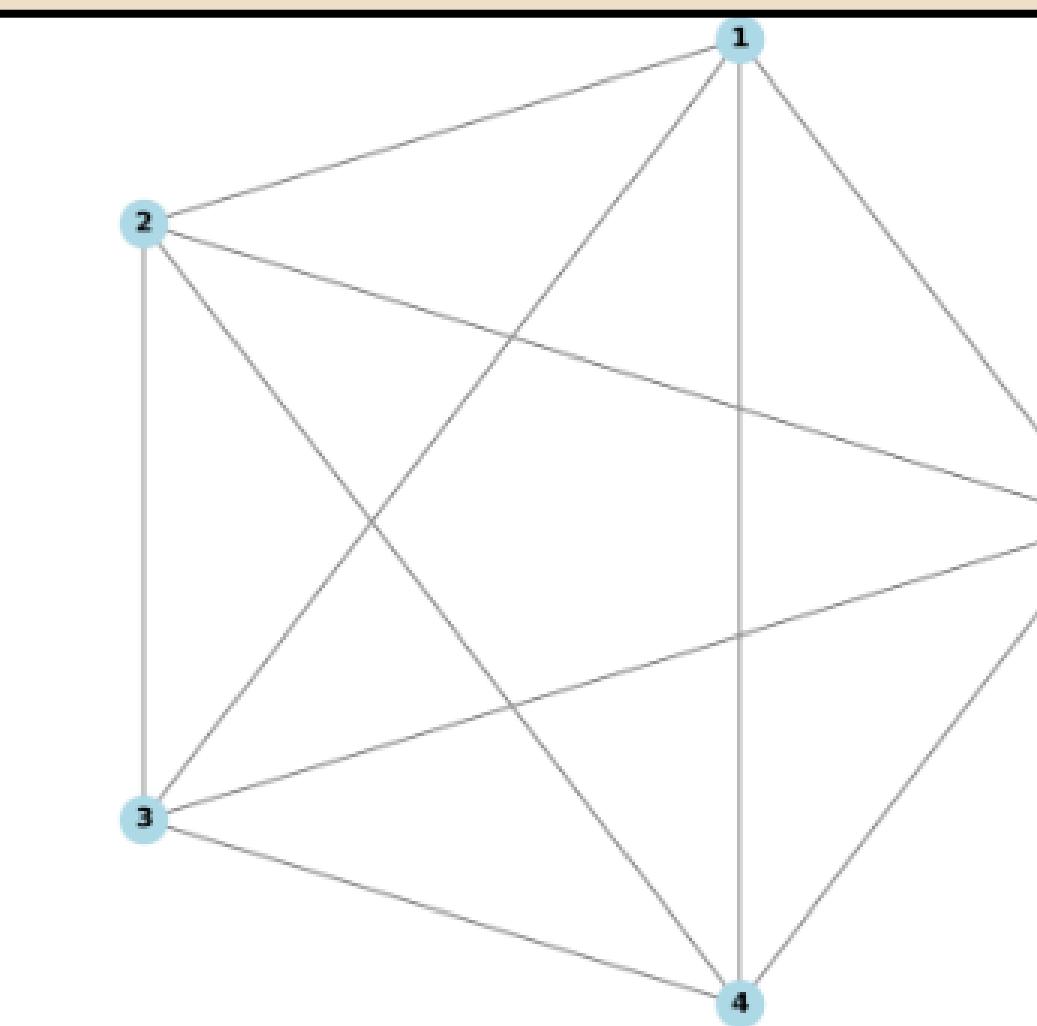
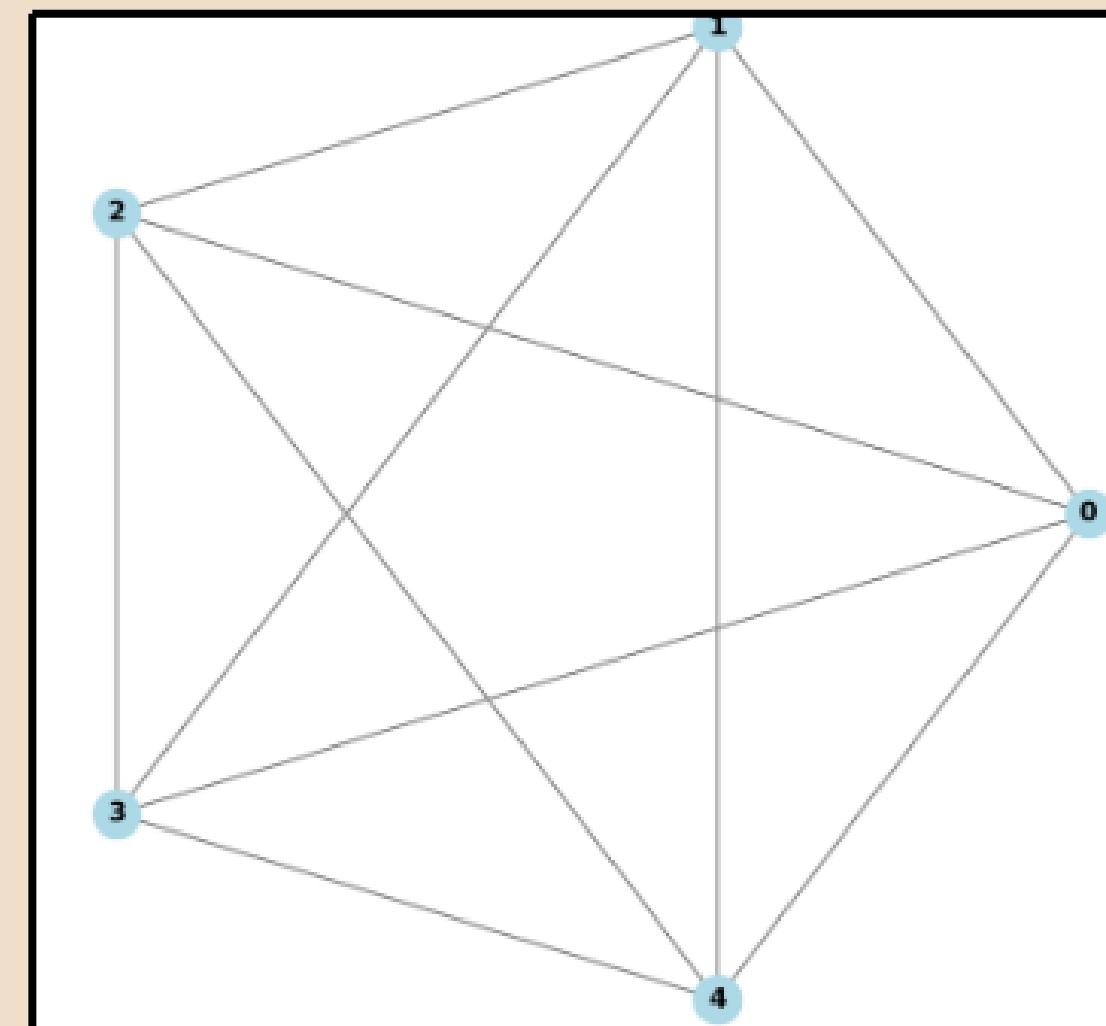


```
N: 7 L: 9  
0: [6, 3, 4]  
1: [2]  
2: [1, 4, 5]  
3: [0, 4, 5]  
4: [2, 0, 5, 3]  
5: [4, 2, 3]  
6: [0]
```



*Kolejne kompilacje*  
:D

```
N: 7 L: 11  
0: [3, 5, 6]  
1: [3]  
2: [4]  
3: [0, 1, 4, 5, 6]  
4: [2, 3, 5, 6]  
5: [0, 3, 4, 6]  
6: [0, 3, 4, 5]
```



```
N: 5 L: 10  
0: [2, 4, 1, 3]  
1: [2, 3, 4, 0]  
2: [0, 4, 3, 1]  
3: [2, 1, 4, 0]  
4: [2, 0, 3, 1]
```

```
N: 5 L: 10  
0: [1, 2, 3, 4]  
1: [0, 2, 3, 4]  
2: [0, 1, 3, 4]  
3: [0, 1, 2, 4]  
4: [0, 1, 2, 3]
```

# Zestaw 2.1

# Przemysław Ryś



# **Zestaw 2.2**

# **Szymon Łabędziewski**



# **zadanie 4**

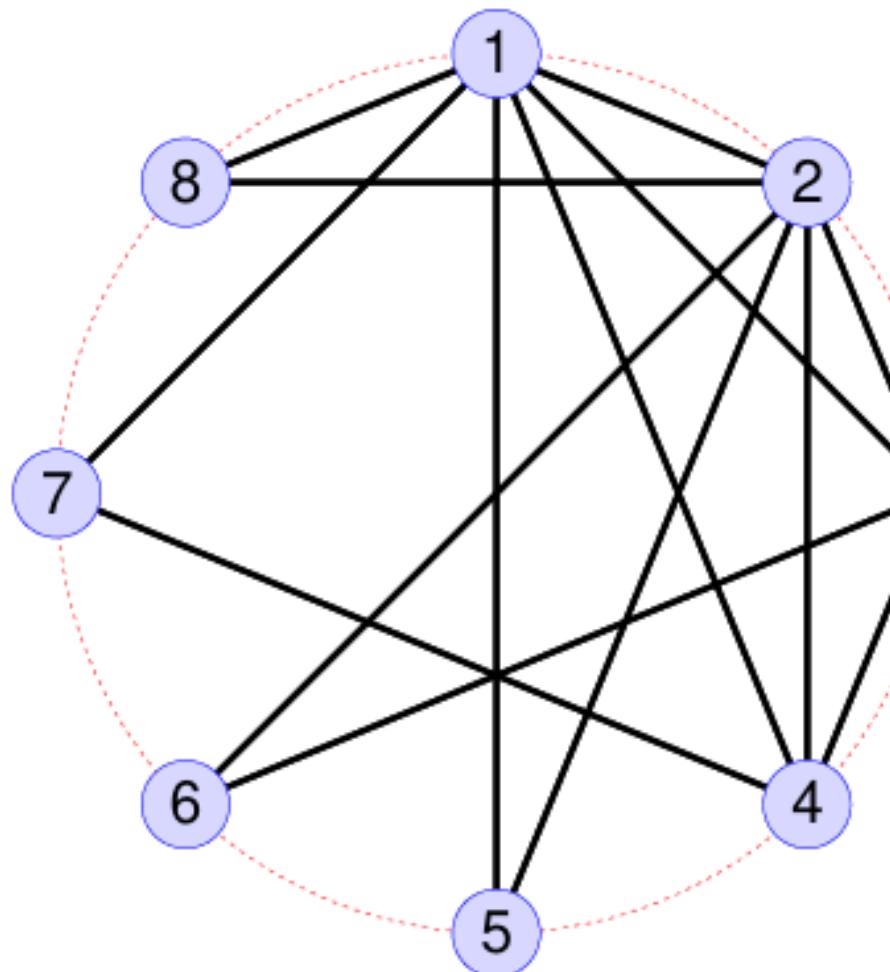
*Używając powyższych programów, napisać program do tworzenia losowego grafu eulerowskiego i znajdowania na nim cyklu Eulera*

#### Ad. 4: Graf eulerowski, poszukiwanie cyklu Eulera

**Dane wejściowe:** liczba wierzchołków  $n$  lub nic (jeśli  $n$  będzie również losowane).

**Wyjście programu:** cykl Eulera w dowolnej formie (np. wypisanie poszczególnych wierzchołków).

**Przykład:** dla  $n = 8 \Rightarrow$  wylosowano stopnie  $4 \ 2 \ 6 \ 2 \ 6 \ 2 \ 4 \ 2$ , dla których wygenerowano losowy graf wejściowy z rys. 2. Znaleziony cykl Eulera: listing 2.

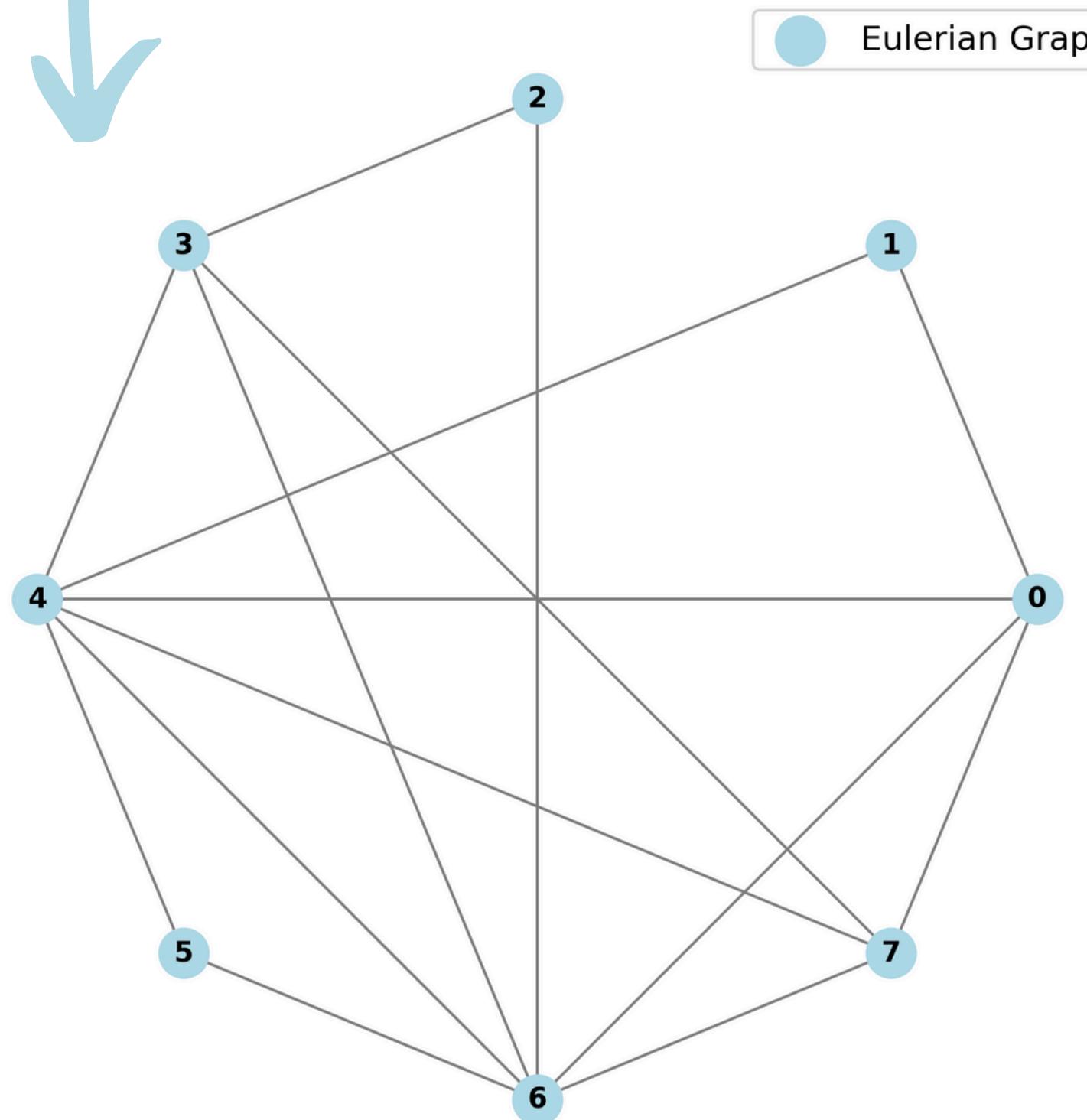


Rysunek 2: Wygenerowany przykładowy graf eulerowski (po stu randomizacjach)

```
1 [1 - 2 - 3 - 1 - 4 - 2 - 5 - 1 - 7 - 4 - 3 - 6 - 2 - 8 - 1]
```

Listing 2: Dane **wyjściowe** – wypisano kolejne wierzchołki cyklu Eulera

```
n = 8
euler_graph = generate_eulerian_graph(n)
print("Generated Eulerian graph (degrees):", [len(nei) for nei in euler_graph.data])
euler_cycle = find_eulerian_cycle(euler_graph)
print("Eulerian cycle:", euler_cycle)
Draw(euler_graph, "eulerian_graph.png", "Eulerian Graph")
```



Generated Eulerian graph (degrees): [4, 2, 2, 4, 6, 2, 6, 4]  
Eulerian cycle: [0, 1, 4, 5, 6, 2, 3, 7, 0, 4, 3, 6, 7, 4, 6, 0]  
Graf zapisany jako eulerian\_graph.png

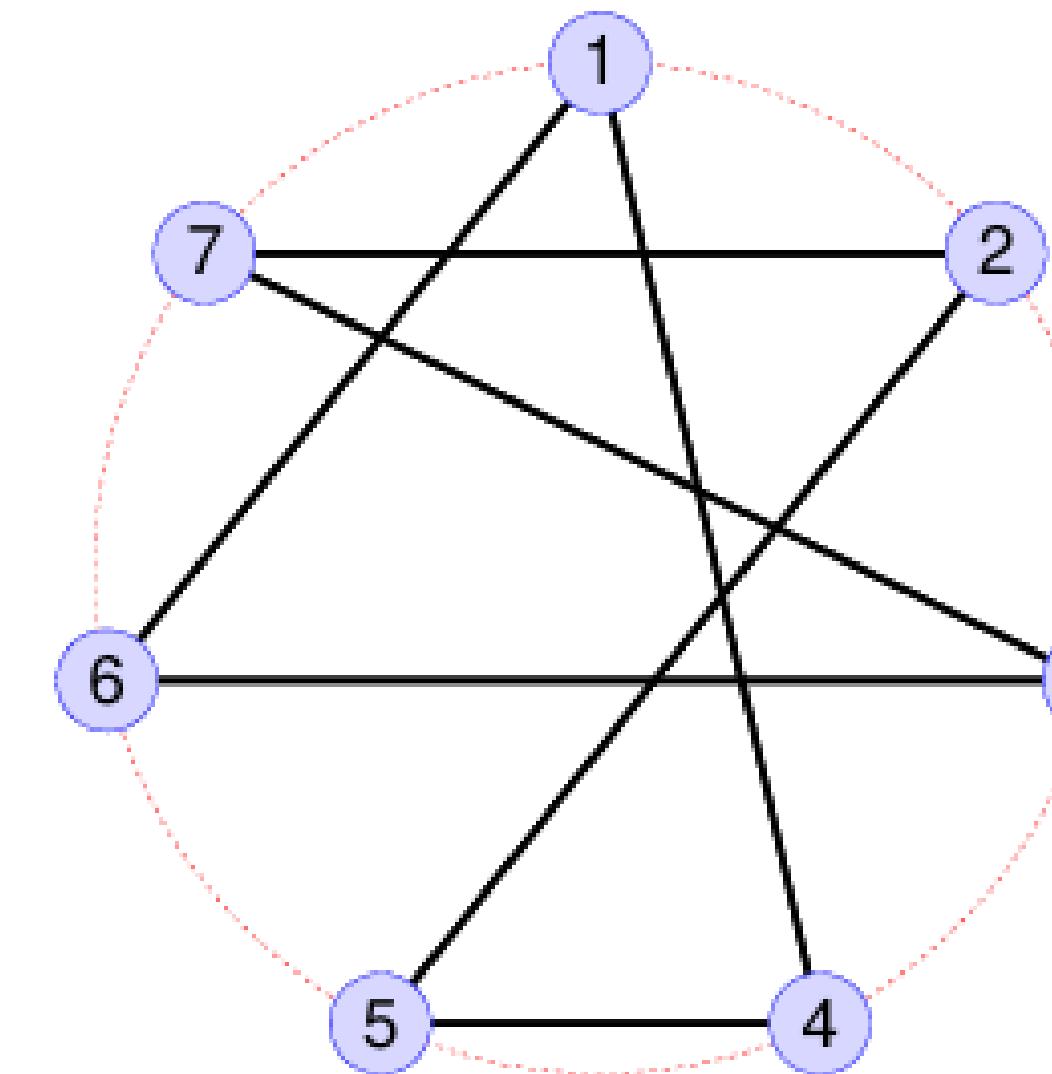
## **zadanie 5**

***Napisać program do generowania losowych grafów  
k-regularnych.***

## Ad. 5: Grafy $k$ -regularne

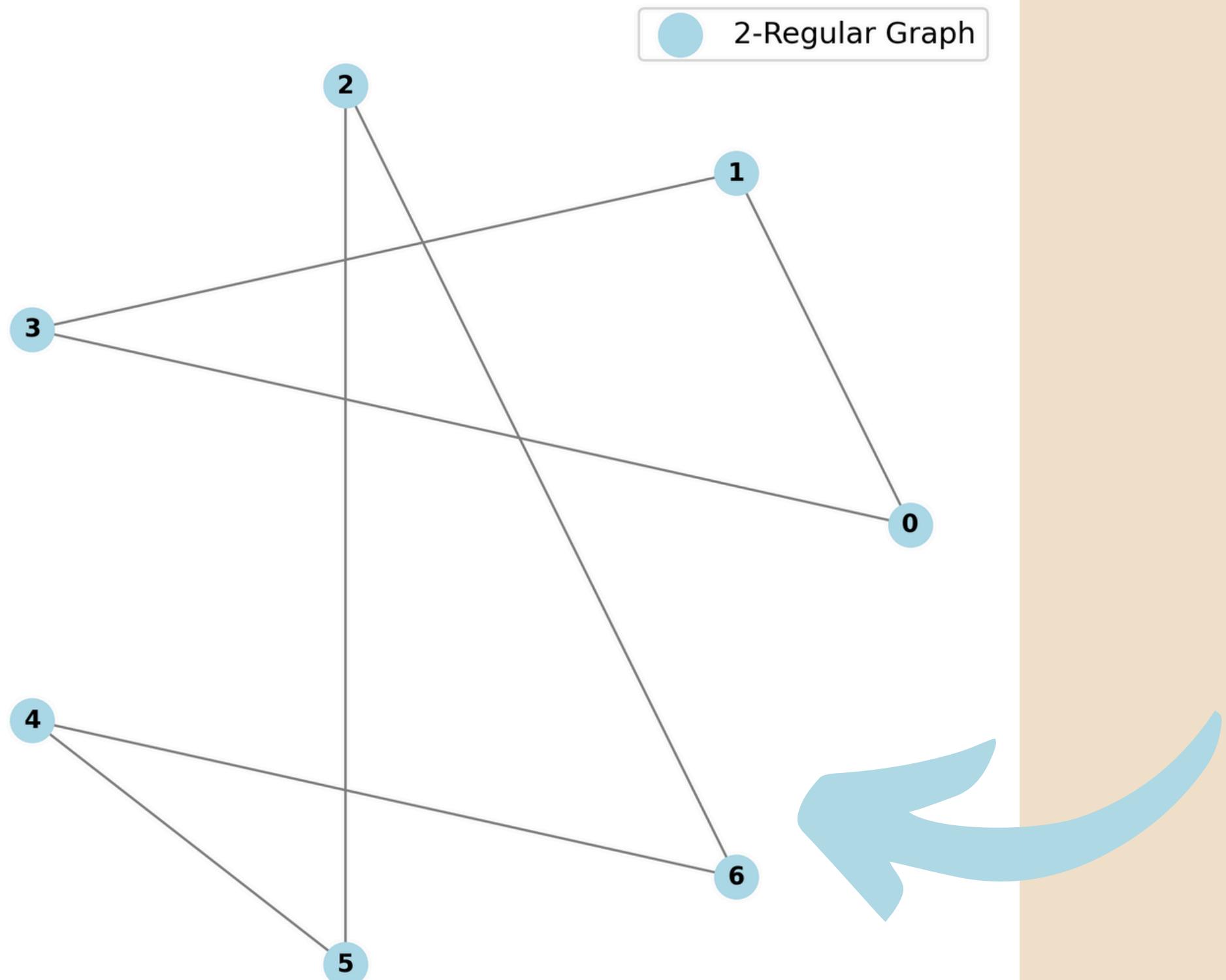
Dane wejściowe: liczba wierzchołków  $n$  oraz stopień wierzchołków  $k$ .

Wyjście programu: wynikowy graf w dowolnej reprezentacji, np. wizualizacja (jak w przykładzie z rys. 3.).



Rysunek 3: Wygenerowany losowy graf  $k$ -regularny dla parametrów wejściowych:  $n = 7$ ,  $k = 2$  (po stu randomizacjach)

```
n, k = 7, 2
k_reg_graph = generate_k_regular_graph(n, k)
print(f"Generated {k}-regular graph:")
print(k_reg_graph)
Draw(k_reg_graph, "k_regular_graph.png", f"{k}-Regular Graph")
```



**Generated 2-regular graph:**

N: 7 L: 7

0: [1, 3]

1: [0, 3]

2: [6, 5]

3: [1, 0]

4: [6, 5]

5: [4, 2]

6: [2, 4]

Graf zapisany jako k\_regular\_graph.png

## **zadanie 6**

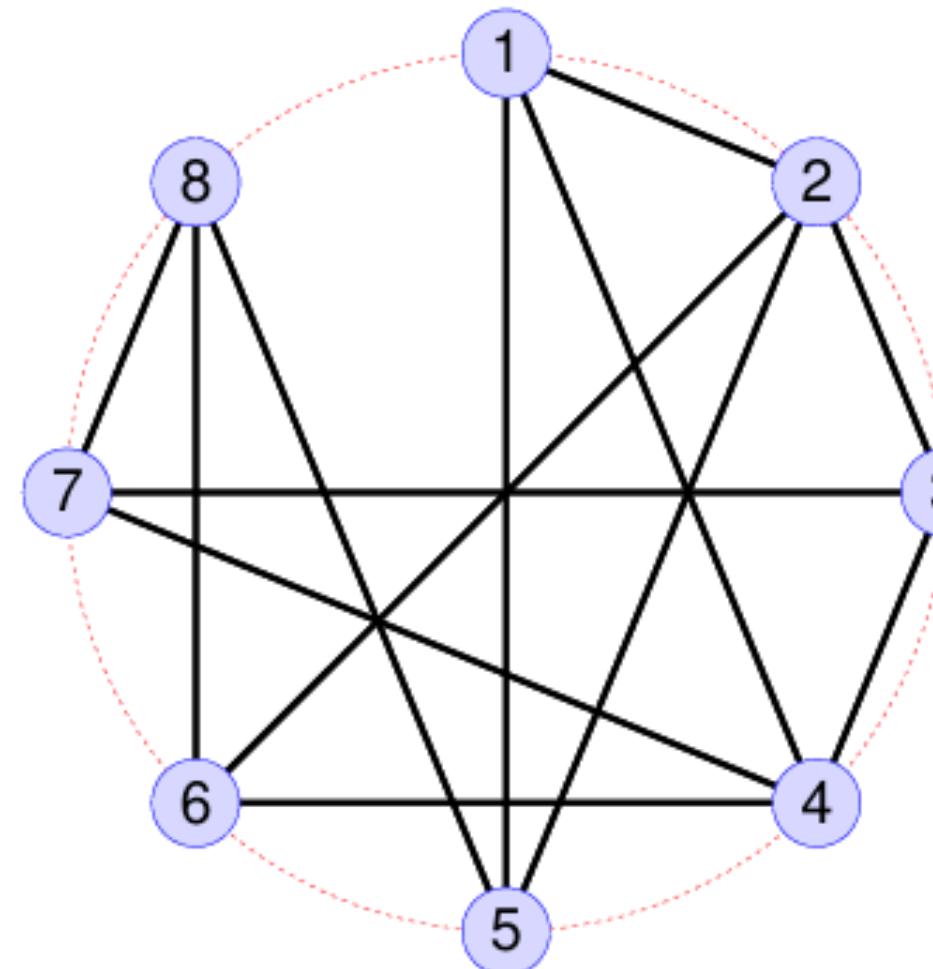
***Napisać program do sprawdzania (dla małych grafów), czy graf jest hamiltonowski.***

## Ad. 6: Graf hamiltonowski, poszukiwanie cyklu Hamiltona

**Dane wejściowe:** dowolne (graf może być zadany przy pomocy wejściowej reprezentacji lub losowany).

**Wyjście programu:** informacja, czy graf jest hamiltonowski; jeśli tak: cykl Hamiltona w dowolnej formie (np. wypisanie poszczególnych wierzchołków).

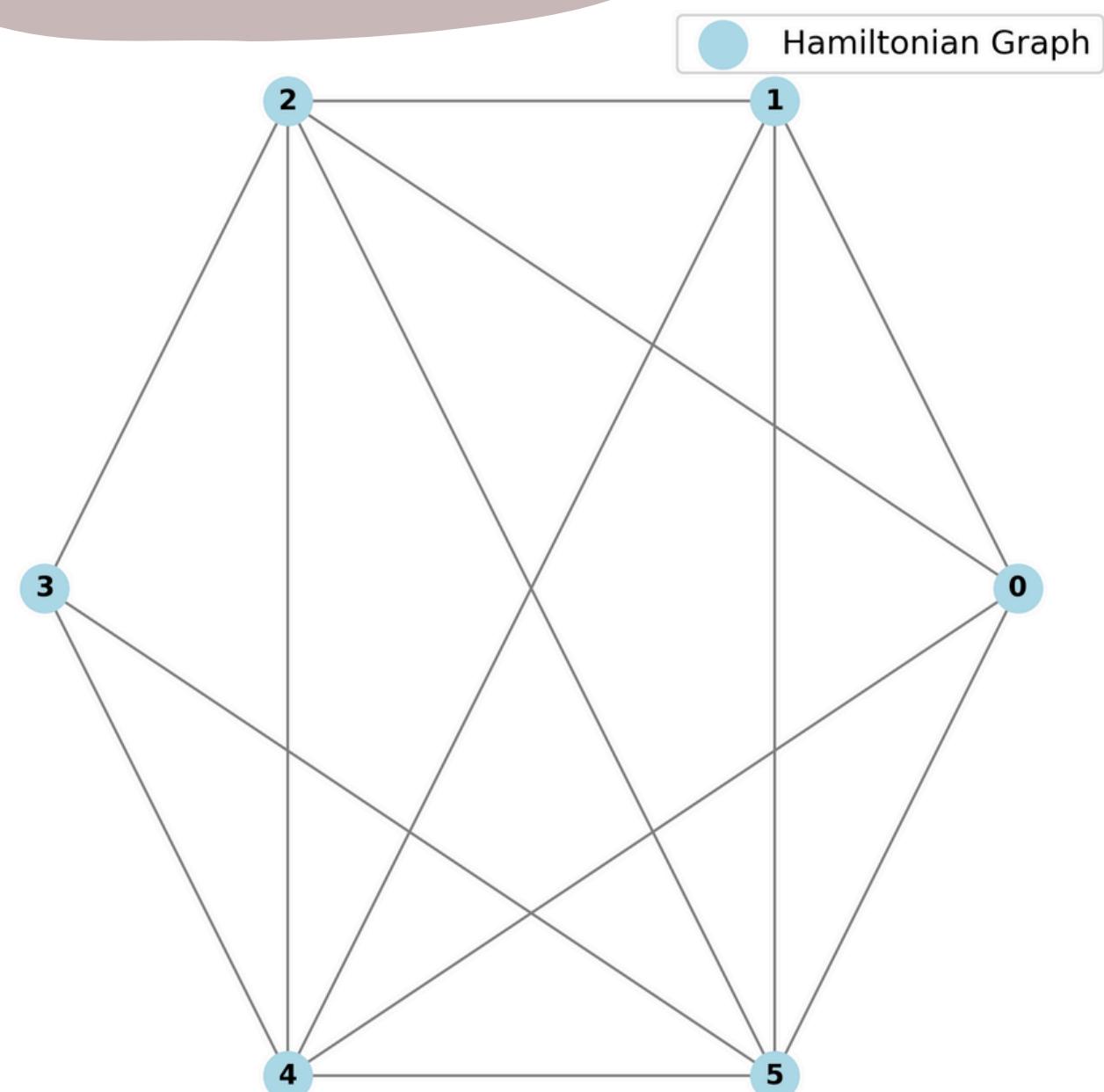
**Przykład:** dla grafu wejściowego z rys. 4. znaleziono cykl Hamiltona z listingu 3.



Rysunek 4: Przykładowy wejściowy graf

```
1 [1 - 2 - 3 - 7 - 4 - 6 - 8 - 5 - 1]
```

Listing 3: Dane wyjściowe – wypisano kolejne wierzchołki cyklu Hamiltona



```
small_graph = random_graph_by_probability(6, 0.6)
print("Small graph for Hamiltonian check:")
print(small_graph)
ham_cycle = find_hamiltonian_cycle(small_graph)
if ham_cycle:
    print("Hamiltonian cycle found:", ham_cycle)
    Draw(small_graph, "hamiltonian_graph.png", "Hamiltonian Graph")
else:
    print("No Hamiltonian cycle exists for this graph.")
```

## ***Small graph for Hamiltonian check:***

**N: 6 L: 13**

**0: [1, 2, 4, 5]**

**1: [0, 2, 4, 5]**

**2: [0, 1, 3, 4, 5]**

**3: [2, 4, 5]**

**4: [0, 1, 2, 3, 5]**

**5: [0, 1, 2, 3, 4]**

**Hamiltonian cycle found: [0, 1, 2, 3, 4, 5, 0]**

**Graf zapisany jako hamiltonian\_graph.png**

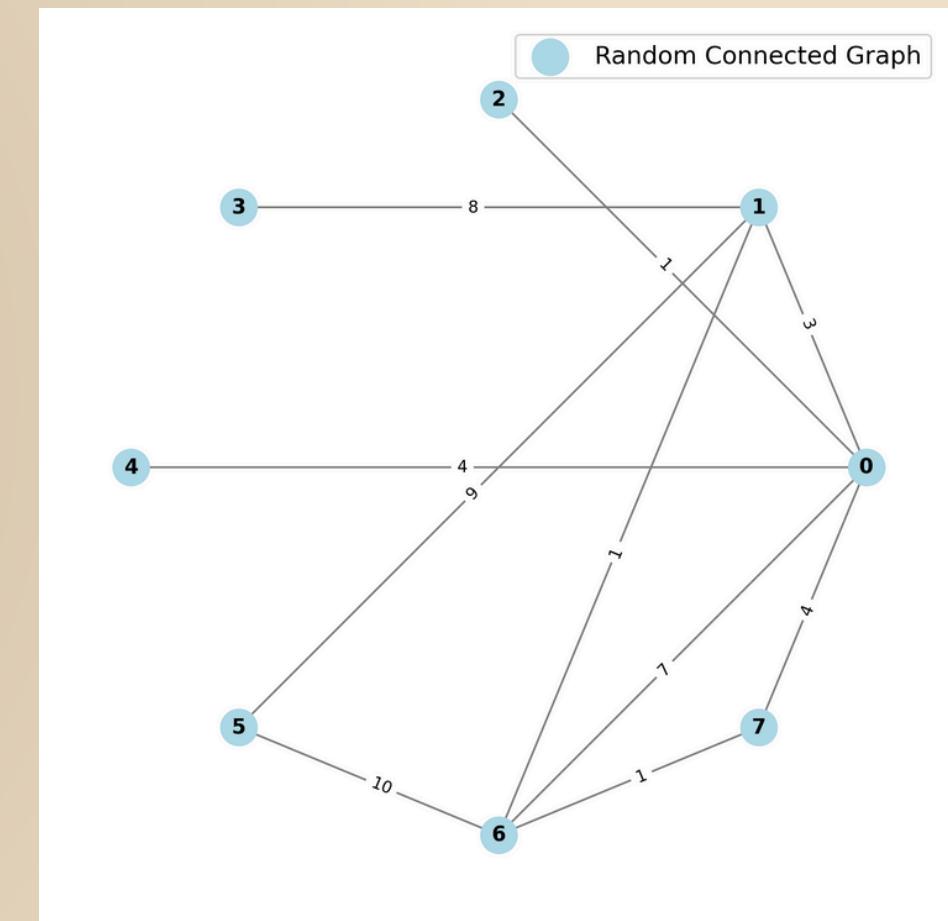
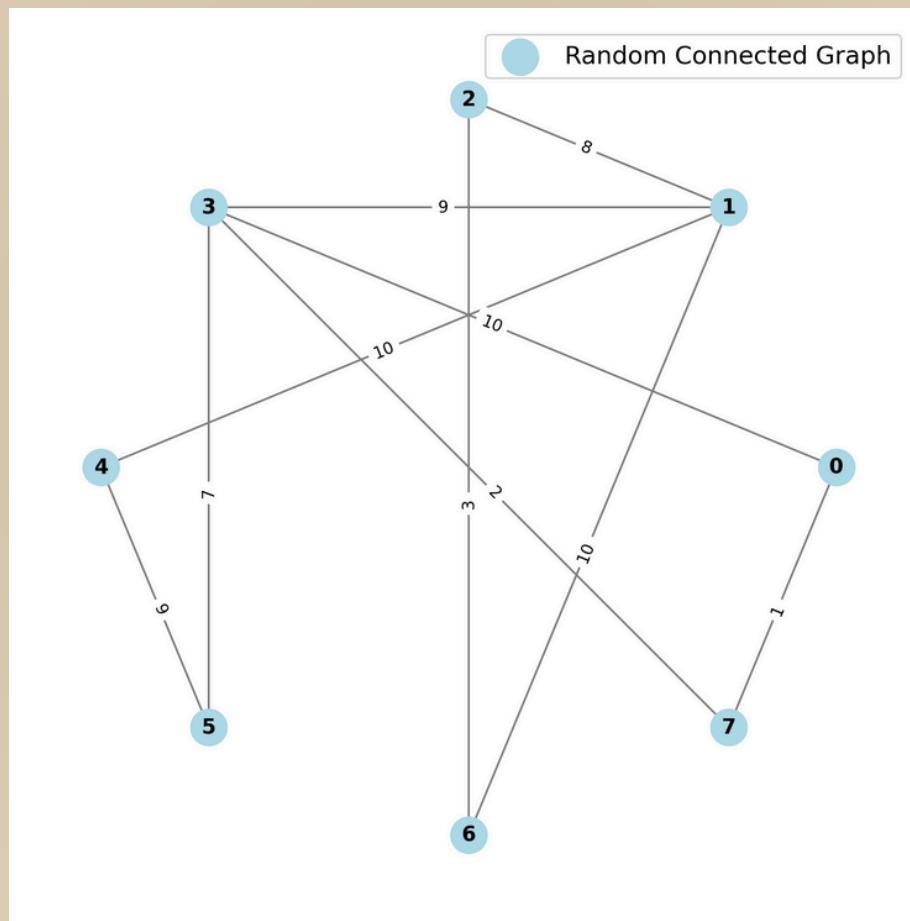
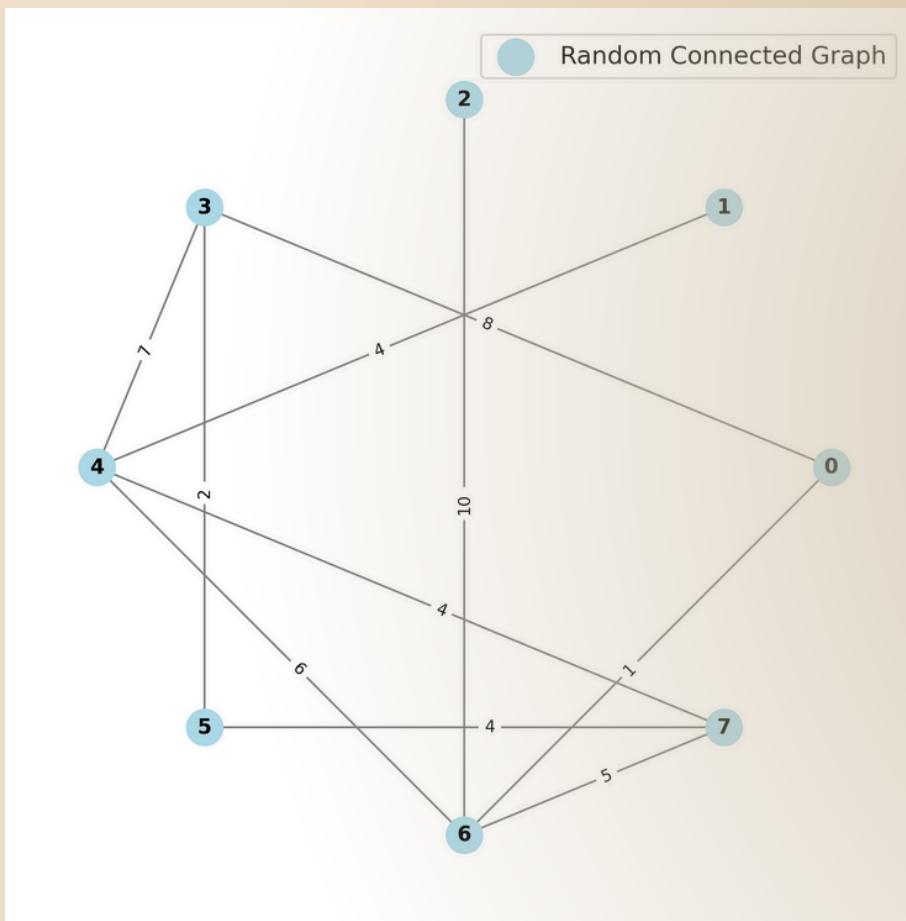
# **Zestaw 3**

# **Joanna Hełdak**



# zadanie 1

Korzystając z programów z poprzednich zestawów, wygenerować spójny graf losowy. Przypisać każdej krawędzi tego grafu losową wagę będącą liczbą naturalną z zakresu od 1 do 10.



Wygenerowałem losowy graf spójny:

- Użyłem już zaimplementowanej funkcji do generowania grafu losowego na podstawie liczby wierzchołków oraz liczby krawędzi
- Skorzystałem z już zaimplementowanej funkcji zwracającej największą spójną składową grafu

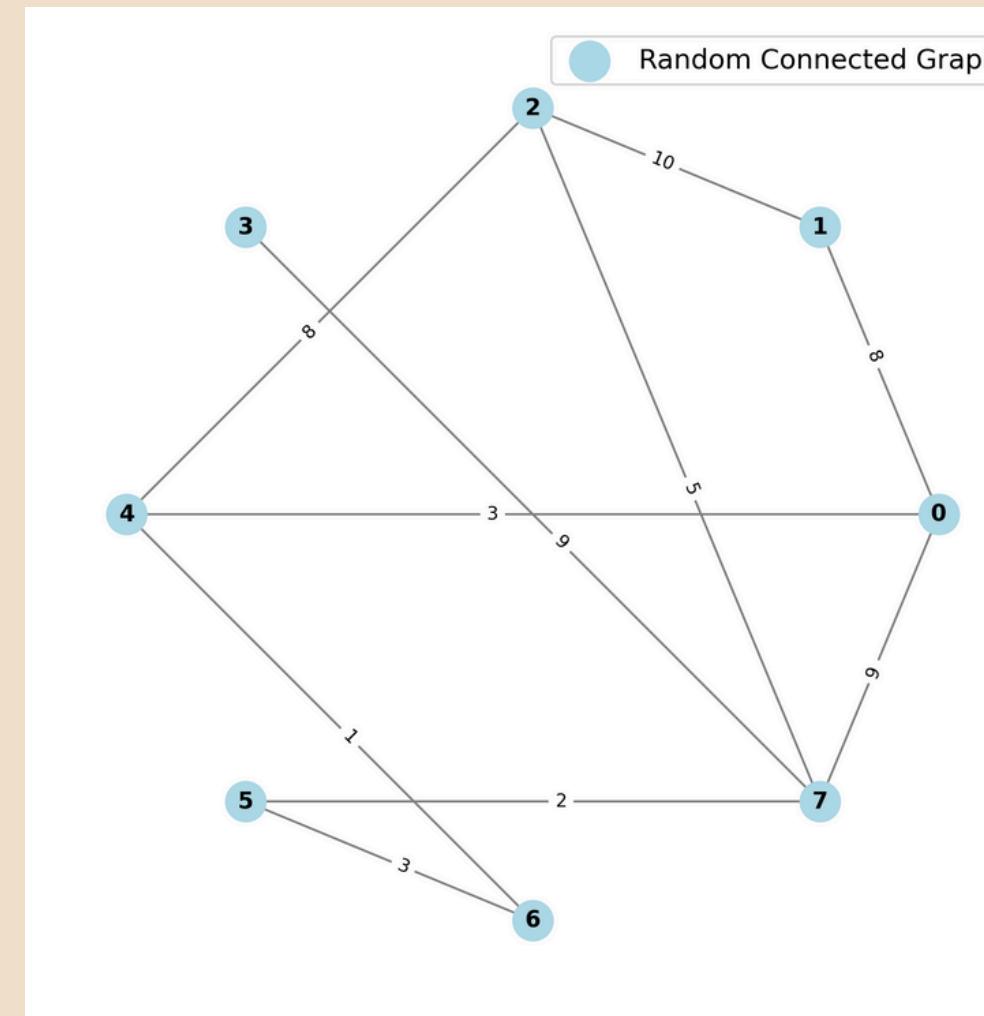
Wagi:

- Zmodyfikowałem naszą klasę Graph: dodałem słownik wag i metodę zapisującą wagę według referencji do funkcji generującej wartości wag
- Żeby zwizualizować utworzony graf z wagami, zmodyfikowałem naszą funkcję Draw w pliku DrawGraph.py: linijka 47



# zadanie 2

Zaimplementować algorytm Dijkstry do znajdowania najkrótszych ścieżek od zadanego wierzchołka do pozostałych wierzchołków i zastosować go do grafu z zadania pierwszego, w którym wagi krawędzi interpretowane są jako odległości wierzchołków. Wypisać wszystkie najkrótsze ścieżki od danego wierzchołka i ich długości.

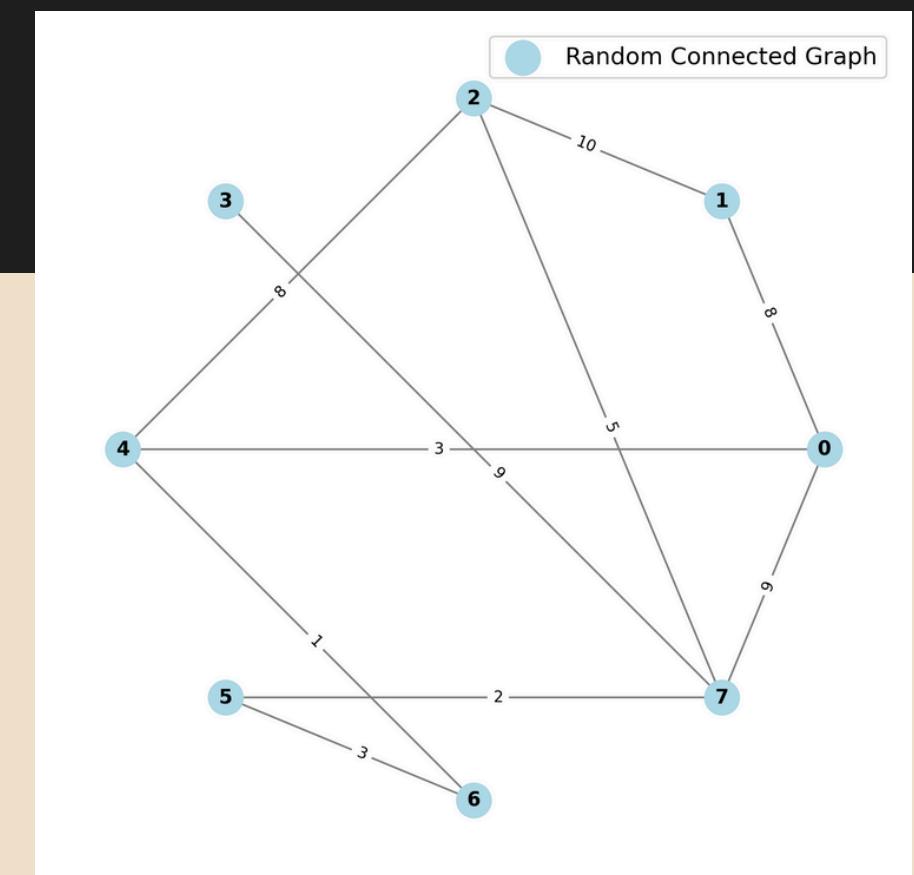


```
distancess, paths = dijkstra(g, 0)  
distancess, paths = dijkstra(g, 4)
```

```
distances, paths = dijkstra(g, 0)
```

Output po 1. iteracji do końca:

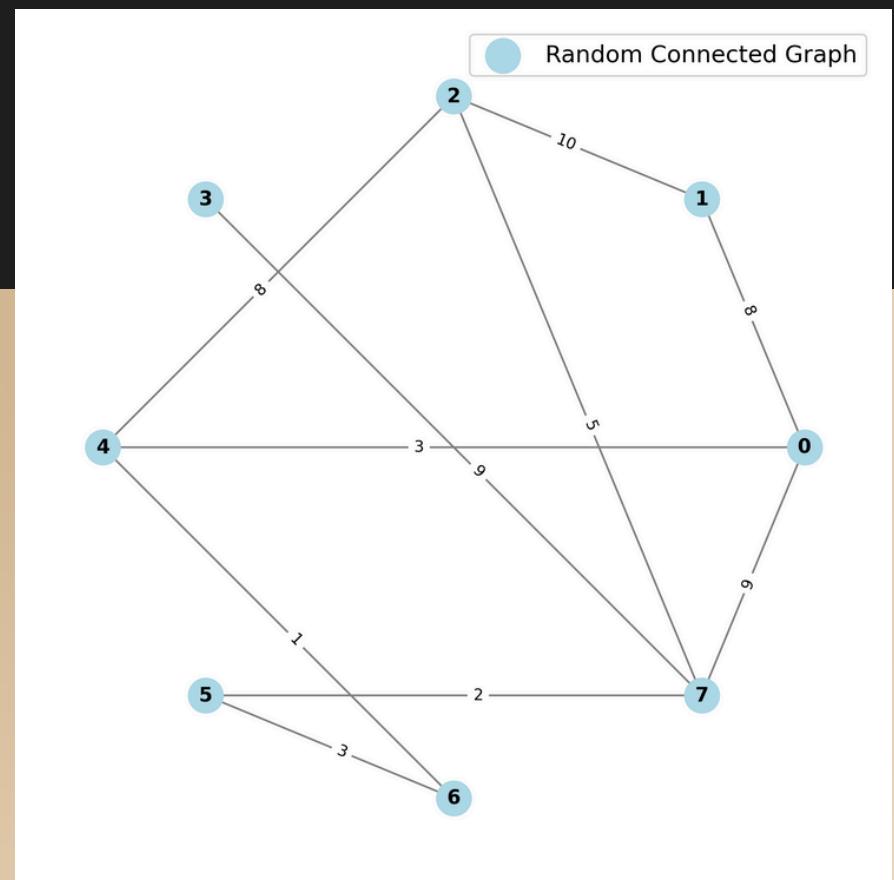
```
Visited: {0}, Distances: {0: 0, 1: 8, 2: 1000000000.0, 3: 10000000000.0, 4: 3, 5: 10000000000.0, 6: 10000000000.0, 7: 9}, Predecessors: {0: None, 1: 0, 2: None, 3: None, 4: 0, 5: None, 6: None, 7: 0}
Visited: {0, 4}, Distances: {0: 0, 1: 8, 2: 11, 3: 10000000000.0, 4: 3, 5: 10000000000.0, 6: 4, 7: 9}, Predecessors: {0: None, 1: 0, 2: 4, 3: None, 4: 0, 5: None, 6: 4, 7: 0}
Visited: {0, 4, 6}, Distances: {0: 0, 1: 8, 2: 11, 3: 10000000000.0, 4: 3, 5: 7, 6: 4, 7: 9}, Predecessors: {0: None, 1: 0, 2: 4, 3: None, 4: 0, 5: 6, 6: 4, 7: 0}
Visited: {0, 4, 5, 6}, Distances: {0: 0, 1: 8, 2: 11, 3: 10000000000.0, 4: 3, 5: 7, 6: 4, 7: 9}, Predecessors: {0: None, 1: 0, 2: 4, 3: None, 4: 0, 5: 6, 6: 4, 7: 0}
Visited: {0, 1, 4, 5, 6}, Distances: {0: 0, 1: 8, 2: 11, 3: 10000000000.0, 4: 3, 5: 7, 6: 4, 7: 9}, Predecessors: {0: None, 1: 0, 2: 4, 3: None, 4: 0, 5: 6, 6: 4, 7: 0}
Visited: {0, 1, 4, 5, 6, 7}, Distances: {0: 0, 1: 8, 2: 11, 3: 18, 4: 3, 5: 7, 6: 4, 7: 9}, Predecessors: {0: None, 1: 0, 2: 4, 3: 7, 4: 0, 5: 6, 6: 4, 7: 0}
Visited: {0, 1, 2, 4, 5, 6, 7}, Distances: {0: 0, 1: 8, 2: 11, 3: 18, 4: 3, 5: 7, 6: 4, 7: 9}, Predecessors: {0: None, 1: 0, 2: 4, 3: 7, 4: 0, 5: 6, 6: 4, 7: 0}
Visited: {0, 1, 2, 3, 4, 5, 6, 7}, Distances: {0: 0, 1: 8, 2: 11, 3: 18, 4: 3, 5: 7, 6: 4, 7: 9}, Predecessors: {0: None, 1: 0, 2: 4, 3: 7, 4: 0, 5: 6, 6: 4, 7: 0}
Final Distances: {0: 0, 1: 8, 2: 11, 3: 18, 4: 3, 5: 7, 6: 4, 7: 9}, Predecessors: {0: None, 1: 0, 2: 4, 3: 7, 4: 0, 5: 6, 6: 4, 7: 0}
Paths = {0: [0], 1: [0, 1], 2: [0, 4, 2], 3: [0, 7, 3], 4: [0, 4], 5: [0, 4, 6, 5], 6: [0, 4, 6], 7: [0, 7]}
START: s = 0
d(0) = 0 ==> [0]
d(1) = 8 ==> [0 - 1]
d(2) = 11 ==> [0 - 4 - 2]
d(3) = 18 ==> [0 - 7 - 3]
d(4) = 3 ==> [0 - 4]
d(5) = 7 ==> [0 - 4 - 6 - 5]
d(6) = 4 ==> [0 - 4 - 6]
d(7) = 9 ==> [0 - 7]
```



```
distances, paths = dijkstra(g, 4)
```

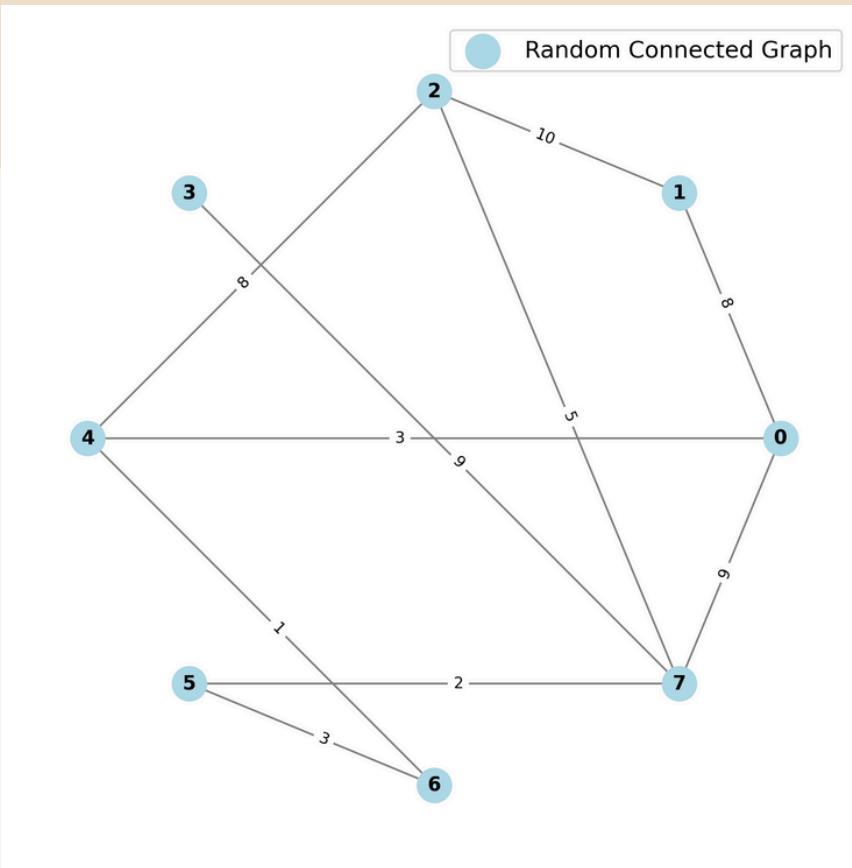
Output po 1. iteracji do końca:

```
Visited: {4}, Distances: {0: 3, 1: 10000000000.0, 2: 8, 3: 10000000000.0, 4: 0, 5: 10000000000.0, 6: 1, 7: 10000000000.0}, Predecessors: {0: 4, 1: None, 2: 4, 3: None, 4: None, 5: None, 6: 4, 7: None}
Visited: {4, 6}, Distances: {0: 3, 1: 10000000000.0, 2: 8, 3: 10000000000.0, 4: 0, 5: 4, 6: 1, 7: 10000000000.0}, Predecessors: {0: 4, 1: None, 2: 4, 3: None, 4: None, 5: 6, 6: 4, 7: None}
Visited: {0, 4, 6}, Distances: {0: 3, 1: 11, 2: 8, 3: 10000000000.0, 4: 0, 5: 4, 6: 1, 7: 12}, Predecessors: {0: 4, 1: 0, 2: 4, 3: None, 4: None, 5: 6, 6: 4, 7: 0}
Visited: {0, 4, 5, 6}, Distances: {0: 3, 1: 11, 2: 8, 3: 10000000000.0, 4: 0, 5: 4, 6: 1, 7: 6}, Predecessors: {0: 4, 1: 0, 2: 4, 3: None, 4: None, 5: 6, 6: 4, 7: 5}
Visited: {0, 4, 5, 6, 7}, Distances: {0: 3, 1: 11, 2: 8, 3: 15, 4: 0, 5: 4, 6: 1, 7: 6}, Predecessors: {0: 4, 1: 0, 2: 4, 3: 7, 4: None, 5: 6, 6: 4, 7: 5}
Visited: {0, 2, 4, 5, 6, 7}, Distances: {0: 3, 1: 11, 2: 8, 3: 15, 4: 0, 5: 4, 6: 1, 7: 6}, Predecessors: {0: 4, 1: 0, 2: 4, 3: 7, 4: None, 5: 6, 6: 4, 7: 5}
Visited: {0, 1, 2, 4, 5, 6, 7}, Distances: {0: 3, 1: 11, 2: 8, 3: 15, 4: 0, 5: 4, 6: 1, 7: 6}, Predecessors: {0: 4, 1: 0, 2: 4, 3: 7, 4: None, 5: 6, 6: 4, 7: 5}
Visited: {0, 1, 2, 3, 4, 5, 6, 7}, Distances: {0: 3, 1: 11, 2: 8, 3: 15, 4: 0, 5: 4, 6: 1, 7: 6}, Predecessors: {0: 4, 1: 0, 2: 4, 3: 7, 4: None, 5: 6, 6: 4, 7: 5}
Final Distances: {0: 3, 1: 11, 2: 8, 3: 15, 4: 0, 5: 4, 6: 1, 7: 6}, Predecessors: {0: 4, 1: 0, 2: 4, 3: 7, 4: None, 5: 6, 6: 4, 7: 5}
Paths = {0: [4, 0], 1: [4, 0, 1], 2: [4, 2], 3: [4, 6, 5, 7, 3], 4: [4], 5: [4, 6, 5], 6: [4, 6], 7: [4, 6, 5, 7]}
START: s = 4
d(0) = 3 => [4 - 0]
d(1) = 11 => [4 - 0 - 1]
d(2) = 8 => [4 - 2]
d(3) = 15 => [4 - 6 - 5 - 7 - 3]
d(4) = 0 => [4]
d(5) = 4 => [4 - 6 - 5]
d(6) = 1 => [4 - 6]
d(7) = 6 => [4 - 6 - 5 - 7]
```



# zadanie 3

Wyznaczyć macierz odległości między wszystkimi parami wierzchołków na tym grafie.



Pierwszy wiersz

```
START: s = 0
d(0) = 0 => [0]
d(1) = 8 => [0 - 1]
d(2) = 11 => [0 - 4 - 2]
d(3) = 18 => [0 - 7 - 3]
d(4) = 3 => [0 - 4]
d(5) = 7 => [0 - 4 - 6 - 5]
d(6) = 4 => [0 - 4 - 6]
d(7) = 9 => [0 - 7]
```

```
START: s = 3
d(0) = 18 => [3 - 7 - 0]
d(1) = 24 => [3 - 7 - 2 - 1]
d(2) = 14 => [3 - 7 - 2]
d(3) = 0 => [3]
d(4) = 15 => [3 - 7 - 5 - 6 - 4]
d(5) = 11 => [3 - 7 - 5]
d(6) = 14 => [3 - 7 - 5 - 6]
d(7) = 9 => [3 - 7]
```

```
START: s = 6
d(0) = 4 => [6 - 4 - 0]
d(1) = 12 => [6 - 4 - 0 - 1]
d(2) = 9 => [6 - 4 - 2]
d(3) = 14 => [6 - 5 - 7 - 3]
d(4) = 1 => [6 - 4]
d(5) = 3 => [6 - 5]
d(6) = 0 => [6]
d(7) = 5 => [6 - 5 - 7]
```

Drugi wiersz

```
START: s = 1
d(0) = 8 => [1 - 0]
d(1) = 0 => [1]
d(2) = 10 => [1 - 2]
d(3) = 24 => [1 - 2 - 7 - 3]
d(4) = 11 => [1 - 0 - 4]
d(5) = 15 => [1 - 0 - 4 - 6 - 5]
d(6) = 12 => [1 - 0 - 4 - 6]
d(7) = 15 => [1 - 2 - 7]
```

```
START: s = 4
d(0) = 3 => [4 - 0]
d(1) = 11 => [4 - 0 - 1]
d(2) = 8 => [4 - 2]
d(3) = 15 => [4 - 6 - 5 - 7 - 3]
d(4) = 0 => [4]
d(5) = 4 => [4 - 6 - 5]
d(6) = 1 => [4 - 6]
d(7) = 6 => [4 - 6 - 5 - 7]
```

```
START: s = 7
d(0) = 9 => [7 - 0]
d(1) = 15 => [7 - 2 - 1]
d(2) = 5 => [7 - 2]
d(3) = 9 => [7 - 3]
d(4) = 6 => [7 - 5 - 6 - 4]
d(5) = 2 => [7 - 5]
d(6) = 5 => [7 - 5 - 6]
d(7) = 0 => [7]
```

itd.

```
START: s = 2
d(0) = 11 => [2 - 4 - 0]
d(1) = 10 => [2 - 1]
d(2) = 0 => [2]
d(3) = 14 => [2 - 7 - 3]
d(4) = 8 => [2 - 4]
d(5) = 7 => [2 - 7 - 5]
d(6) = 9 => [2 - 4 - 6]
d(7) = 5 => [2 - 7]
```

```
START: s = 5
d(0) = 7 => [5 - 6 - 4 - 0]
d(1) = 15 => [5 - 6 - 4 - 0 - 1]
d(2) = 7 => [5 - 7 - 2]
d(3) = 11 => [5 - 7 - 3]
d(4) = 4 => [5 - 6 - 4]
d(5) = 0 => [5]
d(6) = 3 => [5 - 6]
d(7) = 2 => [5 - 7]
```

Distance matrix:

	0	1	2	3	4	5	6	7
0	0	8	11	18	3	7	4	9
1	8	0	10	24	11	15	12	15
2	11	10	0	14	8	7	9	5
3	18	24	14	0	15	11	14	9
4	3	11	8	15	0	4	1	6
5	7	15	7	11	4	0	3	2
6	4	12	9	14	1	3	0	5
7	9	15	5	9	6	2	5	0

# zadanie 4

Wyznaczyć centrum grafu, to znaczy wierzchołek, którego suma odległości do pozostałych wierzchołków jest minimalna. Wyznaczyć centrum minimax, to znaczy wierzchołek, którego odległość do najdalszego wierzchołka jest minimalna.

Distance matrix:

	0	1	2	3	4	5	6	7	
0	0	8	11	18	3	7	4	9	60
1	8	0	10	24	11	15	12	15	95
2	11	10	0	14	8	7	9	5	64
3	18	24	14	0	15	11	14	9	105
4	3	11	8	15	0	4	1	6	48
5	7	15	7	11	4	0	3	2	49
6	4	12	9	14	1	3	0	5	48
7	9	15	5	9	6	2	5	0	51

Centrum = 4 (suma odleglosci: 48)

Centrum minimax = 2 (odleglosc od najdalszego: 14)



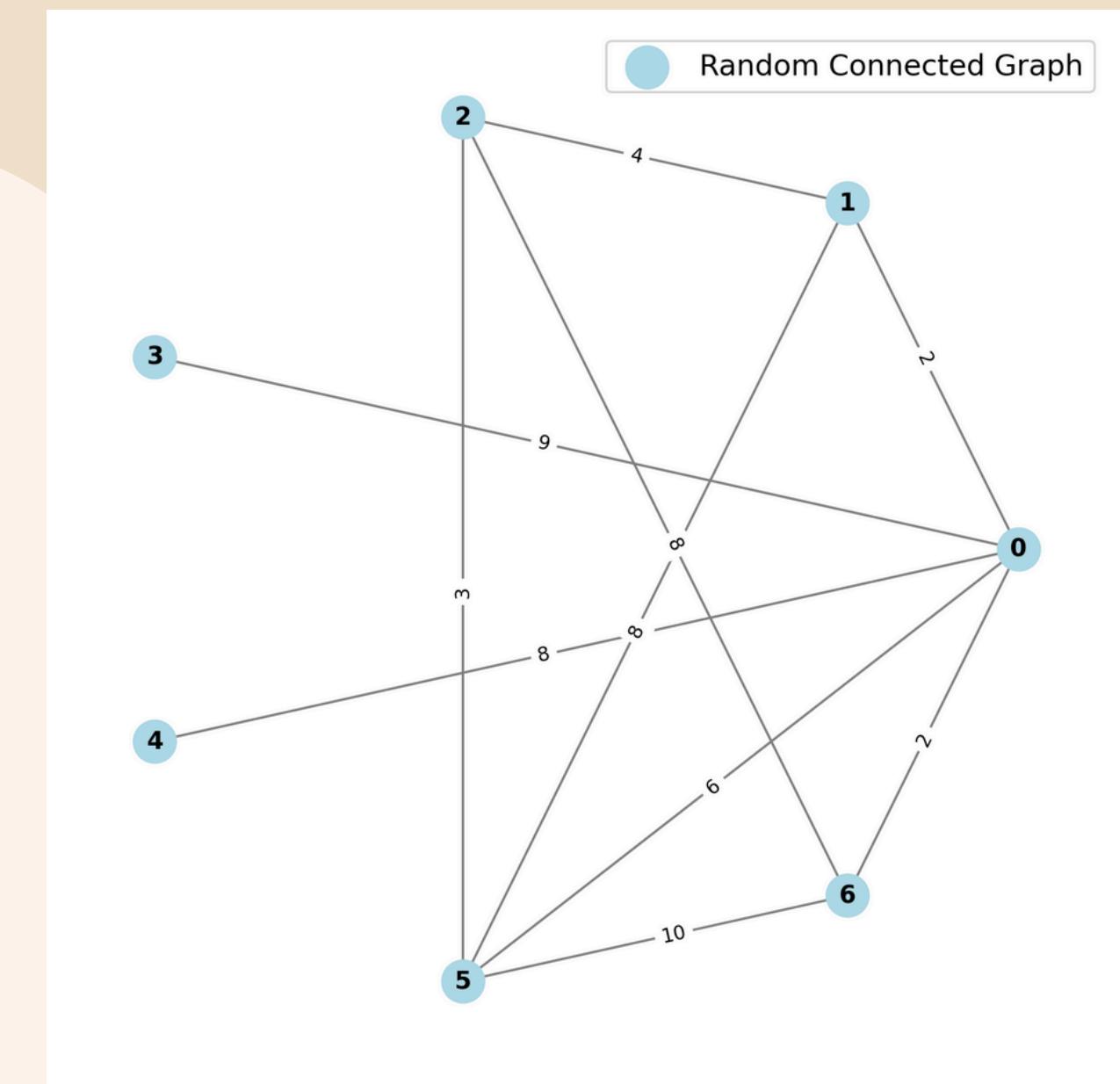
# zadanie 5

**Wyznaczyć minimalne drzewo rozpinające (algorytm Prima lub Kruskala).**

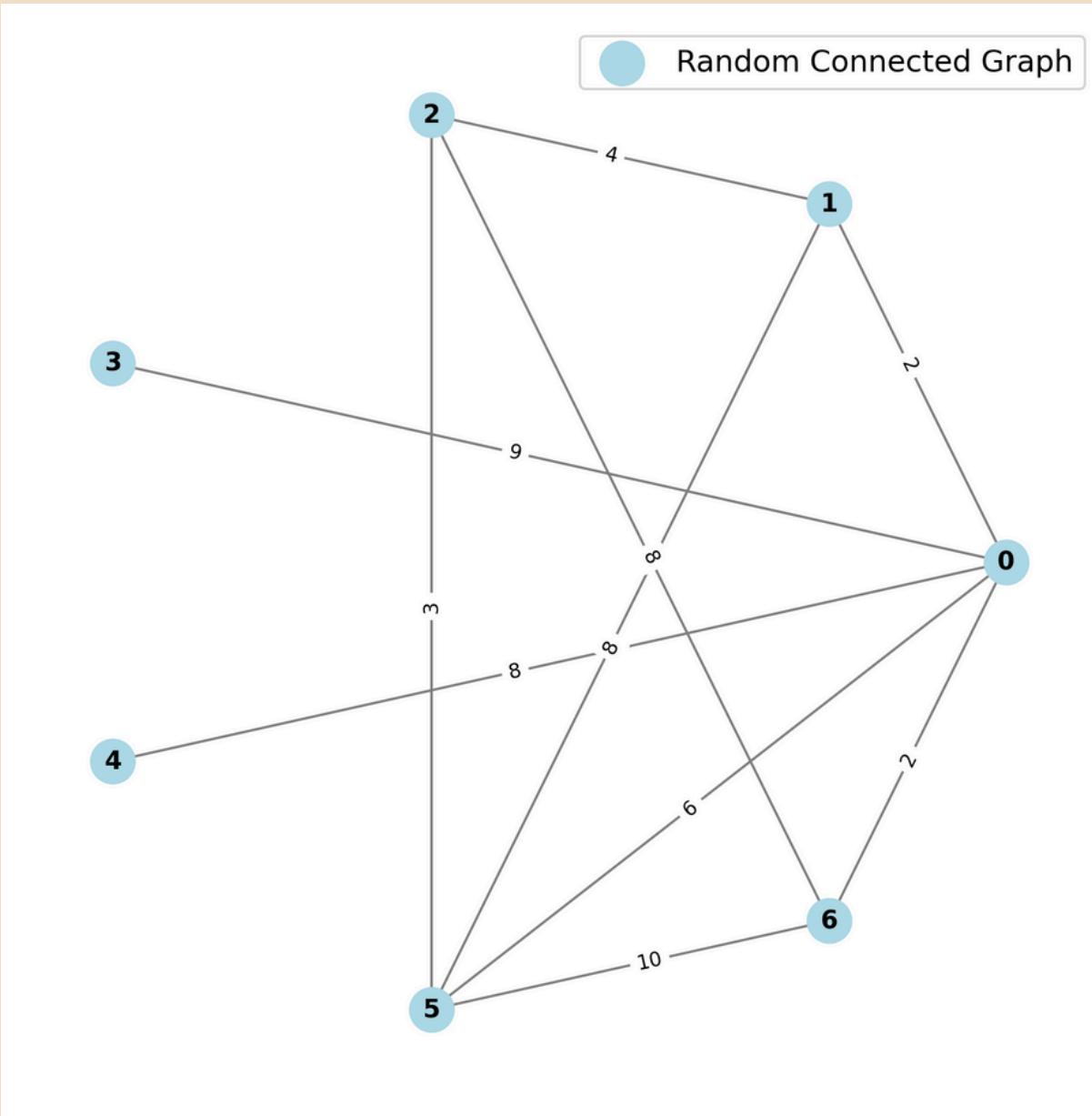
**Drzewo rozpinające grafu** = złożone ze wszystkich wierzchołków i niektórych krawędzi – tak, by było acykliczne i spójne.

**Minimalne drzewo rozpinające grafu** = o minimalnej sumie wag.

inaczej: **MST (Minimal Spinning Tree)**



# Prim



```
T = [0]
W = [1, 2, 3, 4, 5, 6]
edges = [(2, 0, 1), (2, 0, 6), (8, 0, 4), (9, 0, 3), (6, 0, 5)]
T = [0, 1]
W = [2, 3, 4, 5, 6]
edges = [(2, 0, 6), (6, 0, 5), (4, 1, 2), (9, 0, 3), (8, 1, 5), (8, 0, 4)]
T = [0, 1, 6]
W = [2, 3, 4, 5]
edges = [(4, 1, 2), (6, 0, 5), (8, 0, 4), (9, 0, 3), (8, 1, 5), (8, 6, 2), (10, 6, 5)]
T = [0, 1, 6, 2]
W = [3, 4, 5]
edges = [(3, 2, 5), (8, 1, 5), (6, 0, 5), (9, 0, 3), (10, 6, 5), (8, 6, 2), (8, 0, 4)]
T = [0, 1, 6, 2, 5]
W = [3, 4]
edges = [(6, 0, 5), (8, 1, 5), (8, 0, 4), (9, 0, 3), (10, 6, 5), (8, 6, 2)]
T = [0, 1, 6, 2, 5]
W = [3, 4]
edges = [(8, 0, 4), (8, 1, 5), (8, 6, 2), (9, 0, 3), (10, 6, 5)]
T = [0, 1, 6, 2, 5, 4]
W = [3]
edges = [(8, 1, 5), (9, 0, 3), (8, 6, 2), (10, 6, 5)]
T = [0, 1, 6, 2, 5, 4]
W = [3]
edges = [(8, 6, 2), (9, 0, 3), (10, 6, 5)]
T = [0, 1, 6, 2, 5, 4]
W = [3]
edges = [(9, 0, 3), (10, 6, 5)]
Prim:
T=[0, 1, 6, 2, 5, 4, 3]
W=[]
mst_prim=[(0, 1, 2), (0, 6, 2), (1, 2, 4), (2, 5, 3), (0, 4, 8), (0, 3, 9)]
```

Pokazać kroki na wizualizacji!

# Dziękujemy za uwagę!

