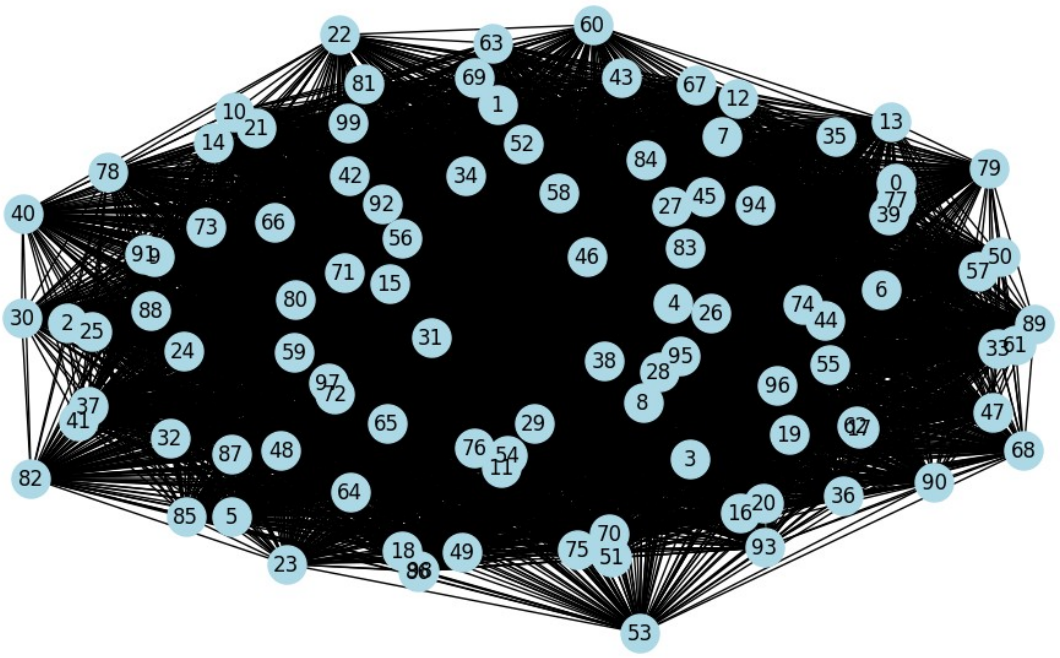
```
# !pip install networkx
# !pip install matplotlib
# !pip install tqdm

import random
import networkx as nx
import matplotlib.pyplot as plt
from itertools import combinations, groupby
```

Generating graph

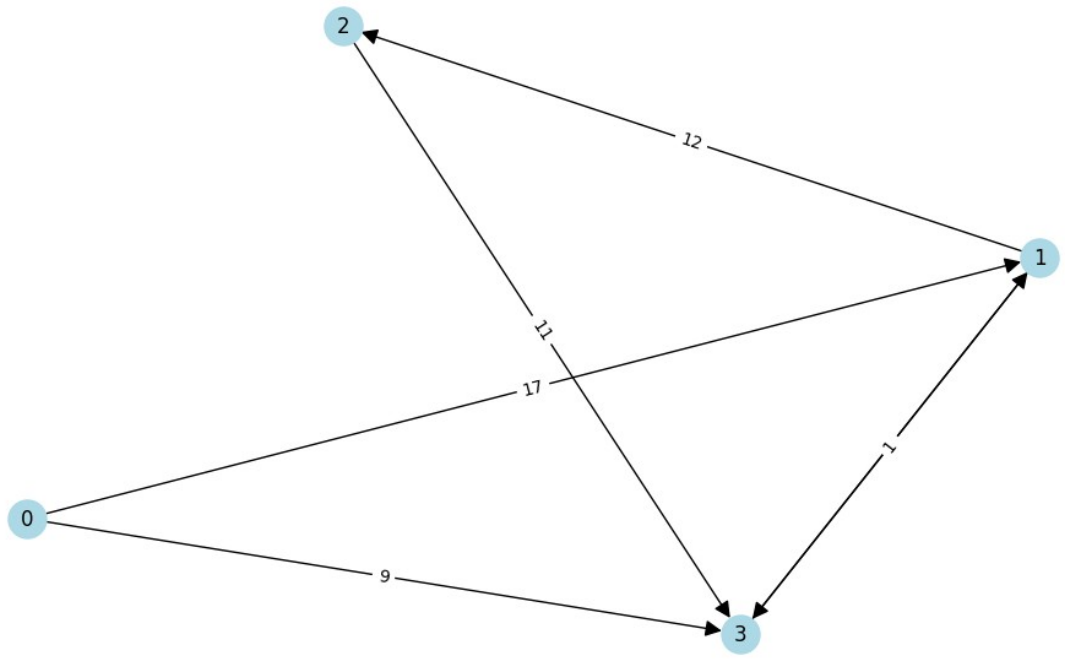
```
# You can use this function to generate a random graph with
'num_of_nodes' nodes
# and 'completeness' probability of an edge between any two nodes
# If 'directed' is True, the graph will be directed
# If 'draw' is True, the graph will be drawn
def gnp_random_connected_graph(num_of_nodes: int,
                               completeness: int,
                               directed: bool = False,
                               draw: bool = False):
    """
    Generates a random graph, similarly to an Erdős-Rényi
    graph, but enforcing that the resulting graph is conneted (in case
of undirected graphs)
    """


    if directed:
        G = nx.DiGraph()
    else:
        G = nx.Graph()
    edges = combinations(range(num_of_nodes), 2)
    G.add_nodes_from(range(num_of_nodes))

    for _, node_edges in groupby(edges, key = lambda x: x[0]):
        node_edges = list(node_edges)
        random_edge = random.choice(node_edges)
        if random.random() < 0.5:
            random_edge = random_edge[::-1]
        G.add_edge(*random_edge)
        for e in node_edges:
            if random.random() < completeness:
                G.add_edge(*e)

    for (u,v,w) in G.edges(data=True):
        w['weight'] = random.randint(-5, 20)

    if draw:
        plt.figure(figsize=(10,6))
```

```
        if directed:
            # draw with edge weights
            pos = nx.arf_layout(G)
            nx.draw(G,pos, node_color='lightblue',
                    with_labels=True,
                    node_size=500,
                    arrowsize=20,
                    arrows=True)
            labels = nx.get_edge_attributes(G,'weight')
            nx.draw_networkx_edge_labels(G, pos,edge_labels=labels)

        else:
            nx.draw(G, node_color='lightblue',
                with_labels=True,
                node_size=500)
    return G

G = gnp_random_connected_graph(100, 1, False, True)
```



For Task 2

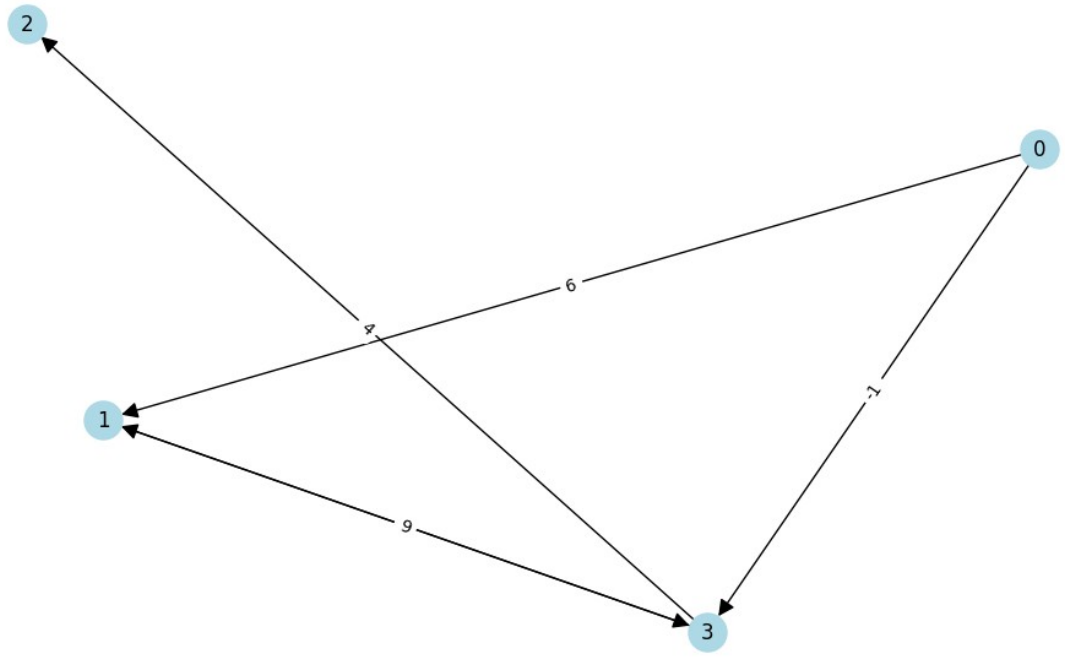
```
G = gnp_random_connected_graph(4, 0.5, True, True)
```



Bellman-Ford algorithm

```
from networkx.algorithms import bellman_ford_predecessor_and_distance


G = gnp_random_connected_graph(4, 0.5, True, True)
# pred is a dictionary of predecessors, dist is a dictionary of
distances
try:
    pred, dist = bellman_ford_predecessor_and_distance(G, 0)
    for k, v in dist.items():
        print(f"Distance to {k}:", v)
except:
    print("Negative cycle detected")
```

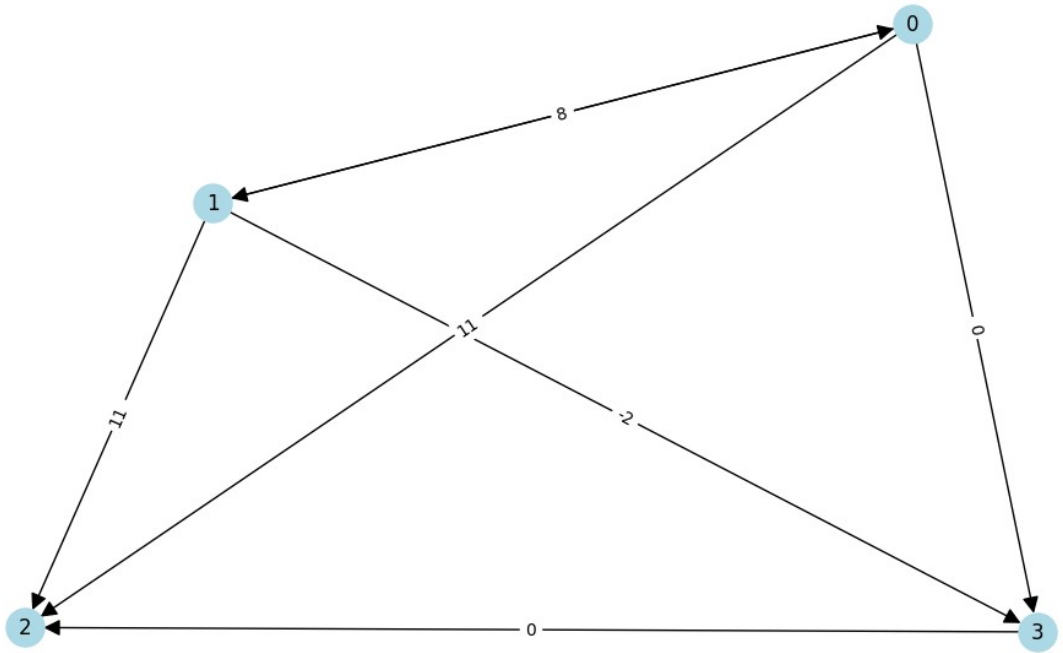
```
Distance to 0: 0
Distance to 3: -1
Distance to 1: 6
Distance to 2: 3
```




```
def bellman_ford(G,start=0):
    distances=[float('inf') for _ in range(len(G.nodes))]
    distances[start]=0
    pred=[[] for _ in range(len(G.nodes))]
    for _ in range(len(G.nodes)-1):
        for u,v,w in G.edges(data=True):
            if distances[u]+w['weight']<distances[v]:
                distances[v]=distances[u]+w['weight']
                pred[v]=[u]
            elif distances[u]+w['weight']==distances[v]:
                if u not in pred[v]:
                    pred[v].append(u)
    for u,v,w in G.edges(data=True):
        if distances[u]+w['weight']<distances[v]:
            return 'Negative cycle detected', None
    return {i:v for i,v in enumerate(pred)},{i:v for i,v in
enumerate(distances)}

G = gnp_random_connected_graph(4, 0.5, True, True)
print(bellman_ford_predecessor_and_distance(G,0))
bellman_ford(G,0)
```

```
({0: [], 1: [0], 2: [3], 3: [1]}, {0: 0, 1: 0, 2: -2, 3: -2})
```

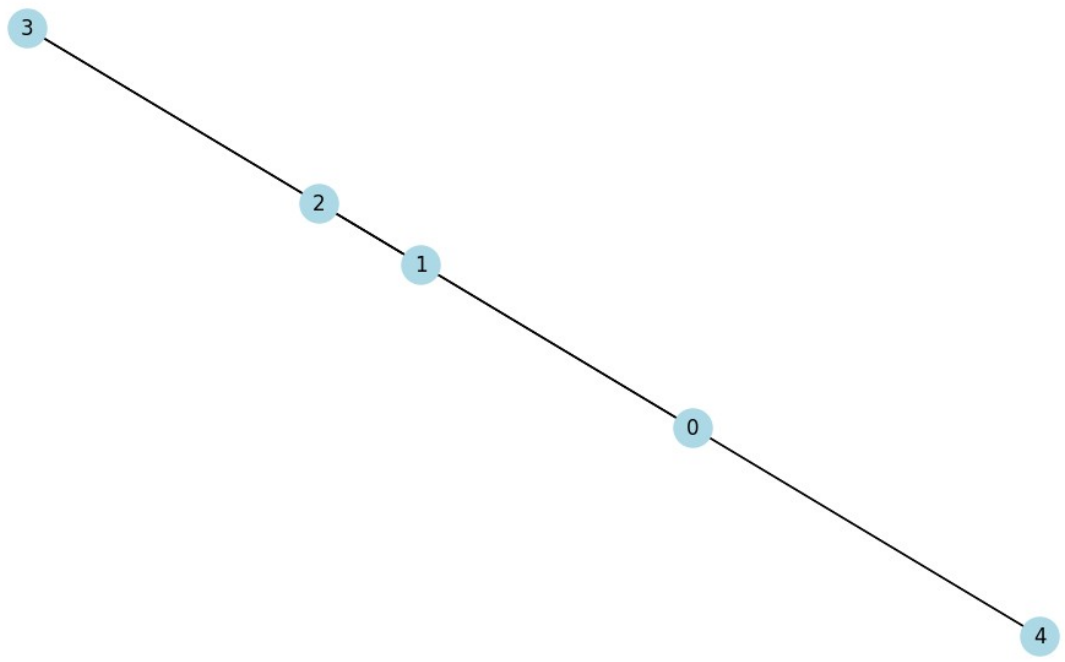
```
({0: [], 1: [0], 2: [3], 3: [1]}, {0: 0, 1: 0, 2: -2, 3: -2})
```



Floyd-Warshall algorithm

```
from networkx.algorithms import
floyd_warshall_predecessor_and_distance

G = gnp_random_connected_graph(5, 0.5, False, True)
for e, v, w in G.edges(data=True):
    print(e, v, w['weight'])
print(G)
```

```
0 4 17
0 1 12
0 2 9
1 3 6
1 2 8
2 3 14
3 4 -2
Graph with 5 nodes and 7 edges
```




```
# pred is a dictionary of predecessors, dist is a dictionary of
distances dictionaries
try:
    pred, dist = floyd_warshall_predecessor_and_distance(G)
    for k, v in dist.items():
        print(f"Distances with {k} source:", dict(v))
except:
    print("Negative cycle detected")
```

```
Distances with 0 source: {0: 0, 4: 12, 1: 12, 2: 9, 3: 14}
Distances with 1 source: {1: 0, 0: 12, 3: 2, 2: 8, 4: 0}
Distances with 2 source: {2: 0, 0: 9, 1: 8, 3: 10, 4: 8}
Distances with 3 source: {3: -4, 1: 2, 2: 10, 4: -6, 0: 14}
Distances with 4 source: {4: -8, 0: 12, 3: -6, 1: 0, 2: 8}
```

```
def create_matrix(graph):
    is_directed = isinstance(graph, nx.DiGraph)
    num_nodes = len(graph.nodes)
    matrix = [[float('inf')] * num_nodes for _ in range(num_nodes)]

    for u, v, w in graph.edges(data=True):
        matrix[u][v] = w['weight']
        if not is_directed:
            matrix[v][u] = w['weight']

    for i in range(num_nodes):
        matrix[i][i] = 0
```

```
    return matrix

def floyd_warshall(graph):
    matrix = create_matrix(graph)
    num_nodes = len(matrix)

    for k in range(num_nodes):
        for i in range(num_nodes):
            for j in range(num_nodes):
                if matrix[i][k] + matrix[k][j] < matrix[i][j]:
                    matrix[i][j] = matrix[i][k] + matrix[k][j]

    for i in range(num_nodes):
        if matrix[i][i] < 0:
            return "Negative cycle occurred"
    res_matrix={}
    for i in range(num_nodes):
        res_matrix[i]={}
        for k in range(num_nodes):
            res_matrix[i][k]=matrix[i][k]
    return res_matrix

floyd_warshall(G)
```

```
'Negative cycle occurred'
```

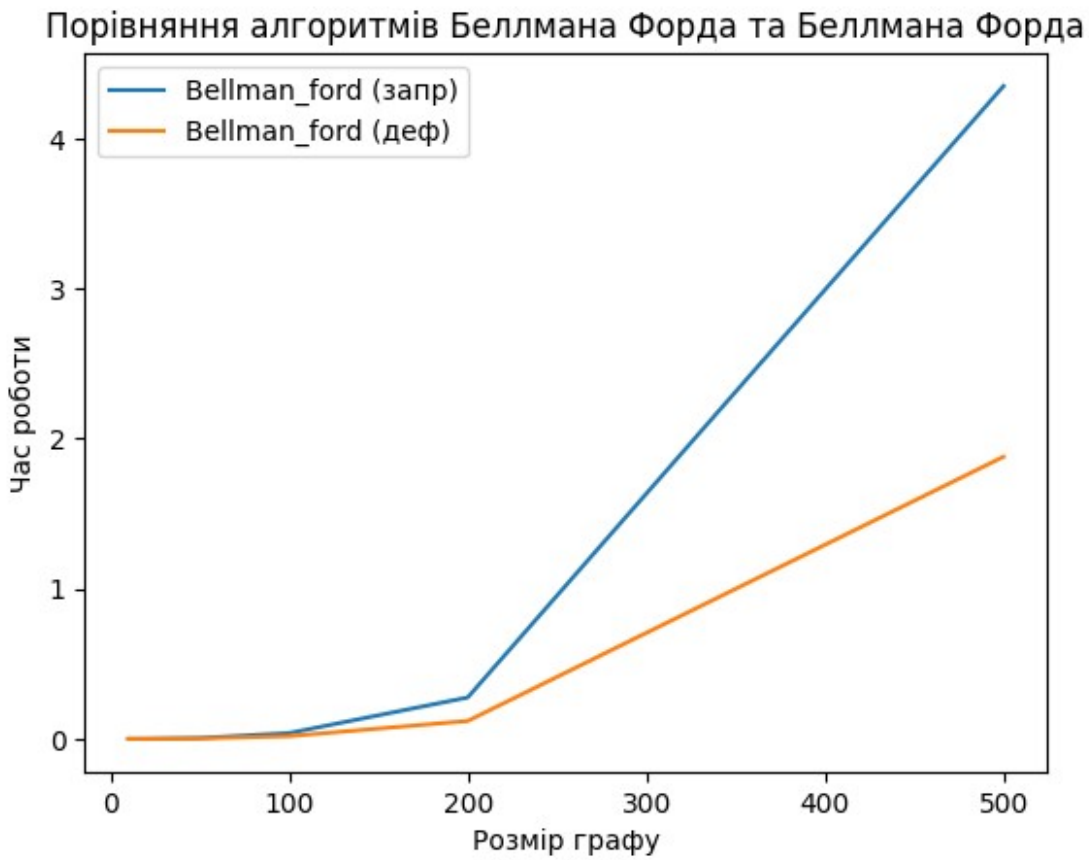
```
import time
from tqdm import tqdm

NUM_OF_ITERATIONS=100
def count_time(function, size):
    time_taken = 0
    for _ in tqdm(range(NUM_OF_ITERATIONS)):

        # note that we should not measure time of graph creation
        G = gnp_random_connected_graph(size, 0.4, False)

        start = time.time()
        try:
            function(G, 0)
        except:
            pass
        end = time.time()

        time_taken += end - start

    return time_taken / NUM_OF_ITERATIONS
```

Bellman Ford

Спочатку ініціалізовуємо список ваг, де початковій вершині присвоюємо значення 0, а решті нескінченність. Ініціалізовуємо список індексів. Далі відбувається "релаксація" ребер, порівнюючи попередньо записані довжини шляху між вершинами, при знаходженні коротшого шляху, записуємо вдповідні значення в список ваг та список індексів. Опісля проводимо перевірку на наявність від'ємного циклу.

```
graph_sizes = [10, 20, 50, 100, 200, 500]
bellman_ford_mine = [count_time(bellman_ford, size) for size in
graph_sizes]  # Приклад часу роботи для алгоритму Крускала
(запрограмованого)
bellman_ford_def = [count_time(bellman_ford_predecessor_and_distance,
size) for size in graph_sizes]  # Приклад часу роботи для алгоритму
Крускала (дефолтного)

plt.plot(graph_sizes, bellman_ford_mine, label='Bellman_ford (запр)')
plt.plot(graph_sizes, bellman_ford_def, label='Bellman_ford (деф)')
plt.xlabel('Розмір графу')
plt.ylabel('Час роботи')
plt.title('Порівняння алгоритмів Беллмана Форда та Беллмана Форда')
plt.legend()
plt.show()
```

```
100%|██████████| 100/100 [00:00<00:00, 2696.23it/s]
100%|██████████| 100/100 [00:00<00:00, 1895.16it/s]
100%|██████████| 100/100 [00:00<00:00, 170.32it/s]
100%|██████████| 100/100 [00:04<00:00, 24.10it/s]
100%|██████████| 100/100 [00:28<00:00,  3.51it/s]
100%|██████████| 100/100 [07:21<00:00,  4.42s/it]
100%|██████████| 100/100 [00:00<00:00, 8638.61it/s]
100%|██████████| 100/100 [00:00<00:00, 2258.10it/s]
100%|██████████| 100/100 [00:00<00:00, 276.33it/s]
100%|██████████| 100/100 [00:02<00:00, 49.85it/s]
100%|██████████| 100/100 [00:12<00:00,  7.86it/s]
100%|██████████| 100/100 [03:14<00:00,  1.94s/it]
```



Алгоритм працює чудово у всіх випадках окрім надто великих, але тоді вже мариці надто великі і важко справитись навіть вбудованому алгоритмові. Погіршення можуть виникати через не повністю оптимізований код, але ми не знаємо як його покращити.

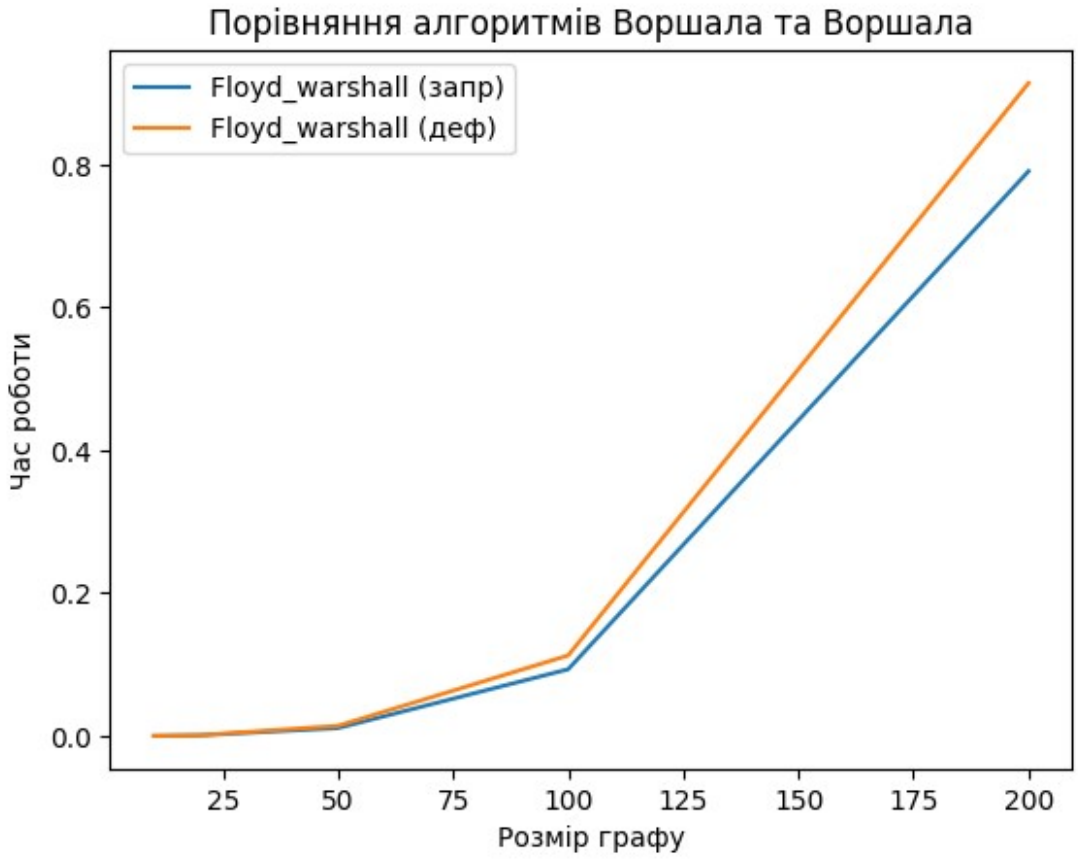
Floyd Warshall

Зробили допоміжні функції, які створюють матрицю ваг. Потім код брав кожен рядок і відповідний стовпець і шукав нові значення за допомогою додавання значень на цих рядках і стовпцях, так робив доки не завершувалась матриця, потім перетворював в словник і повертав. Якщо якась вершина має до себе шлях менше 1, то код повертає "Negative cycle occurred".

```
graph_sizes = [10, 20, 50, 100, 200]
warshall_mine = [count_time(floyd_warshall, size) for size in
graph_sizes]  # Приклад часу роботи для алгоритму Крускала
(запрограмованого)
warshall_def = [count_time(floyd_warshall_predecessor_and_distance,
size) for size in graph_sizes]  # Приклад часу роботи для алгоритму
Крускала (дефолтного)

plt.plot(graph_sizes, warshall_mine, label='Floyd_warshall (запр)')
plt.plot(graph_sizes, warshall_def, label='Floyd_warshall (деф)')
plt.xlabel('Розмір графу')
```

```
plt.ylabel('Час роботи')
plt.title('Порівняння алгоритмів Воршала та Воршала')
plt.legend()
plt.show()
```

```
100%|██████████| 100/100 [00:00<00:00, 7378.49it/s]

100%|██████████| 100/100 [00:00<00:00, 988.53it/s]
100%|██████████| 100/100 [00:01<00:00, 86.15it/s]
100%|██████████| 100/100 [00:09<00:00, 10.46it/s]
100%|██████████| 100/100 [01:19<00:00,  1.25it/s]
100%|██████████| 100/100 [00:00<00:00, 6138.31it/s]
100%|██████████| 100/100 [00:00<00:00, 930.15it/s]
100%|██████████| 100/100 [00:01<00:00, 67.95it/s]
100%|██████████| 100/100 [00:11<00:00,  8.69it/s]
100%|██████████| 100/100 [01:32<00:00,  1.08it/s]
```



З кожним разом алгоритм все сповільнювався і сповільнувався через розмір матриці, тож ми мусили зменшити максимальний розмір до 200, адже потім на виконання одного алгоритму йшло 15-20 секунд. Це може бути пов’язано з створенням матриці, та доволі примітивним обходом її, але наші алгоритми доволі схожі, похибка мінімальна