Матвій Шумилович і Богдан Мілян

In [1]:

```
import random
import time
from itertools import combinations, groupby
import networkx as nx
import matplotlib.pyplot as plt
from tqdm import tqdm
from networkx.algorithms import floyd_warshall_predecessor_and_distance
from networkx.algorithms import bellman_ford_predecessor_and_distance
```

In [2]:

```
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 grap
hs)
    """
    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:
            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
```

In [3]:

```
def bellman_ford(graph:nx.classes.graph.Graph, sourse:int)->dict | str:
    '''
    Bellman-Ford algorithm which allows to know all the passes from the sourse
    returns dictinary {node: distanse:int}

    If negative cycle is detected returns a message:
    'Negative cycle detected!'
```

```
    '''
    lenght = len(graph.nodes()) - 1
    shortest_path = {i:float('inf') for i in range(len(graph.nodes))}
    shortest_path[sourse] = 0

    ###Calculate distances
    data = list(graph.edges(data=True))

    for i in range(lenght):
        cache_path = shortest_path.copy()
        for sour, dest, weight in data:
            if shortest_path[dest] > shortest_path[sour] + weight['weight']:
                shortest_path[dest] = weight['weight'] + shortest_path[sour]
        if cache_path == shortest_path:
            # for _ in range(lenght - i):
            for sour, dest, weight in data:
                if shortest_path[dest] > shortest_path[sour] + weight['weight']:
                    return 'Negative cycle detected!'
            return shortest_path
    ###Check for a negative cycle

    for sour, dest, weight in data:
        if shortest_path[dest] > shortest_path[sour] + weight['weight']:
            return 'Negative cycle detected!'
    return shortest_path
```

In [4]:

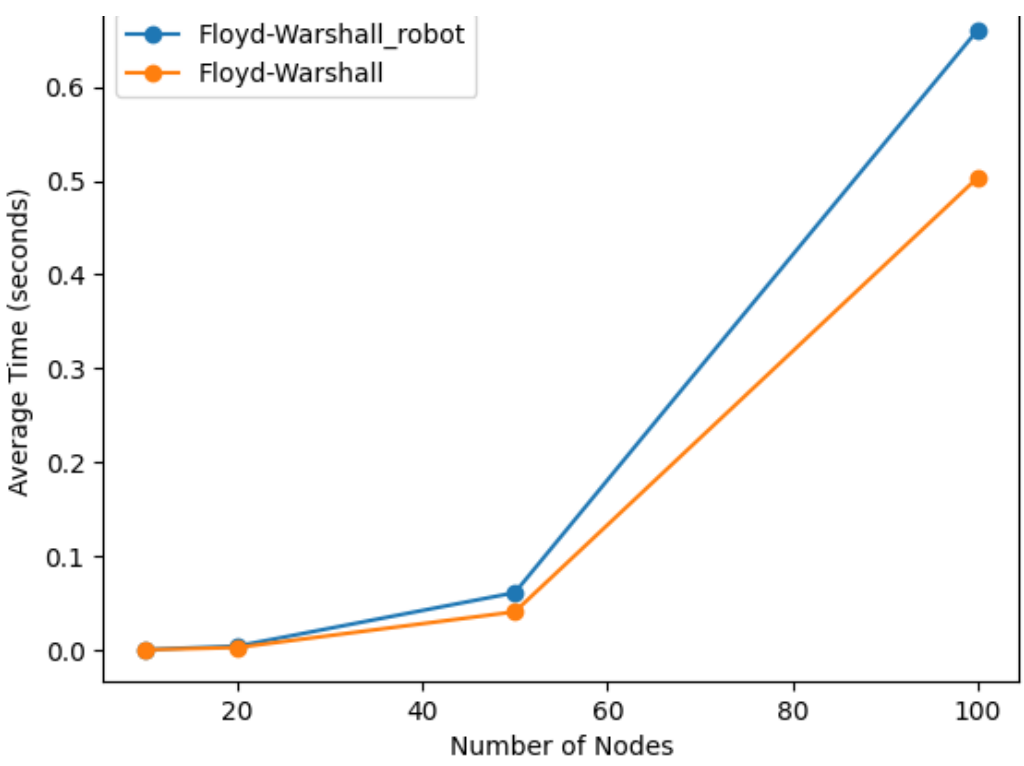
```
"""Floyd Worshal"""
def floyd_warshall(graph):
    edges = list(graph.edges(data=True))
    num_nodes = max(max(u, v) for u, v, _ in edges) + 1
    matrix = [[float('inf') for _ in range(num_nodes)] for _ in range(num_nodes)]
    for edge in edges:
        u, v, w = edge[0], edge[1], edge[2]['weight']
        matrix[u][v] = w
    n = len(matrix)
    dist = [row[:] for row in matrix]
    pred = [[None if i != j and matrix[i][j] != float('inf') \
             else -1 for j in range(n)] for i in range(n)]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] != float('inf') and dist[k][j] != float('inf'):
                    if dist[i][j] > dist[i][k] + dist[k][j]:
                        dist[i][j] = dist[i][k] + dist[k][j]
                        pred[i][j] = k
    pred_dict = {v: {u: pred[u][v] for u in range(n) if pred[u][v] is not None} for v in
range(n)}
    return pred_dict, {v: dict(enumerate(dist[v])) for v in range(n)}
```

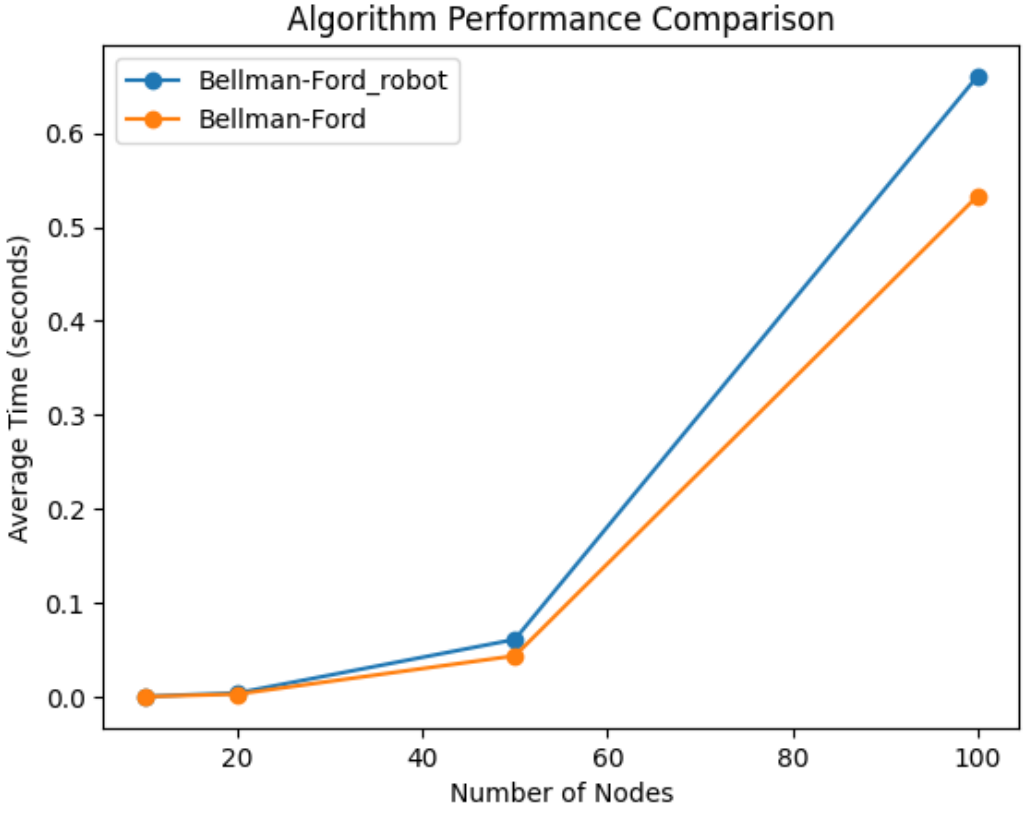
In [5]:

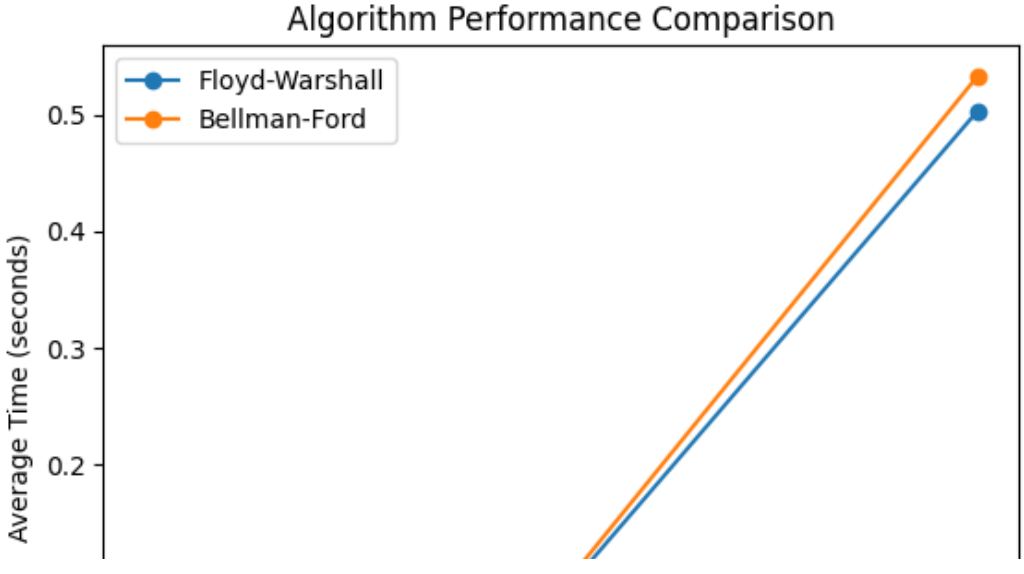
```
#For floyd worshal and Belman
NUM_OF_ITERATIONS = 100
average_times_w = []
average_times_b=[]
average_times_rb=[]
average_times_rw=[]
time_taken = 0
nums=[10,20,50,100]
for i in nums:
    G = gnp_random_connected_graph(i, 0.5, True, False)

    for _ in tqdm(range(NUM_OF_ITERATIONS)):
        start = time.time()
        minimum_spanning_tree = floyd_warshall(G)
        end = time.time()
        time_taken += end - start
    average_time_w = time_taken / NUM_OF_ITERATIONS
```

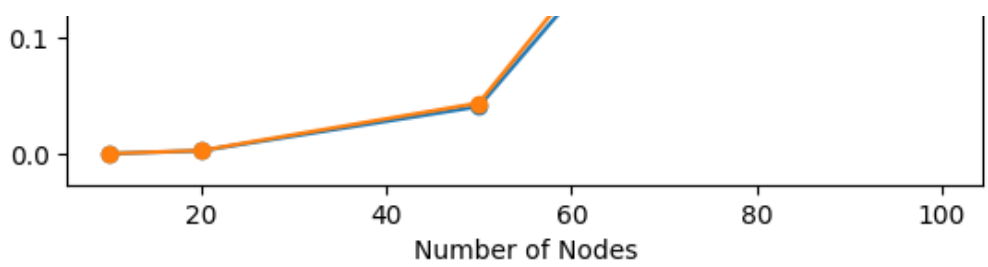
```
    average_times_w.append(average_time_w)

    for _ in tqdm(range(NUM_OF_ITERATIONS)):
        start = time.time()
        minimum_spanning_tree = bellman_ford(G,0)
        end = time.time()
        time_taken += end - start
    average_time_b = time_taken / NUM_OF_ITERATIONS
    average_times_b.append(average_time_b)

    for _ in tqdm(range(NUM_OF_ITERATIONS)):
        start = time.time()
        try:
            pred, dist = floyd_warshall_predecessor_and_distance(G)
        except:
            continue
        end = time.time()
        time_taken += end - start
    average_time_rw = time_taken / NUM_OF_ITERATIONS
    average_times_rw.append(average_time_rw)

    for _ in tqdm(range(NUM_OF_ITERATIONS)):
        start = time.time()
        try:
            pred, dist = bellman_ford_predecessor_and_distance(G, 0)
        except:
            continue
        end = time.time()
        time_taken += end - start
    average_time_rb = time_taken / NUM_OF_ITERATIONS
    average_times_rb.append(average_time_rb)
plt.plot(nums, average_times_rw, marker='o', label='Floyd-Warshall_robot')
plt.plot(nums, average_times_w, marker='o', label='Floyd-Warshall')
plt.title('Algorithm Performance Comparison')
plt.xlabel('Number of Nodes')
plt.ylabel('Average Time (seconds)')
plt.legend()
plt.show()
plt.plot(nums, average_times_rb, marker='o', label='Bellman-Ford_robot')
plt.plot(nums, average_times_b, marker='o', label='Bellman-Ford')
plt.title('Algorithm Performance Comparison')
plt.xlabel('Number of Nodes')
plt.ylabel('Average Time (seconds)')
plt.legend()
plt.show()
plt.plot(nums, average_times_w, marker='o', label='Floyd-Warshall')
plt.plot(nums, average_times_b, marker='o', label='Bellman-Ford')
plt.title('Algorithm Performance Comparison')
plt.xlabel('Number of Nodes')
plt.ylabel('Average Time (seconds)')
plt.legend()
plt.show()
```

```
100%|██████████| 100/100 [00:00<00:00, 2777.76it/s]
100%|██████████| 100/100 [00:00<00:00, 33314.57it/s]
100%|██████████| 100/100 [00:00<00:00, 5263.34it/s]
100%|██████████| 100/100 [00:00<00:00, 25000.32it/s]
100%|██████████| 100/100 [00:00<00:00, 451.03it/s]
100%|██████████| 100/100 [00:00<00:00, 4166.89it/s]
100%|██████████| 100/100 [00:00<00:00, 825.91it/s]
100%|██████████| 100/100 [00:00<00:00, 3030.37it/s]
100%|██████████| 100/100 [00:03<00:00, 27.17it/s]
100%|██████████| 100/100 [00:00<00:00, 330.63it/s]
100%|██████████| 100/100 [00:01<00:00, 57.07it/s]
100%|██████████| 100/100 [00:00<00:00, 167.12it/s]
100%|██████████| 100/100 [00:44<00:00,  2.26it/s]
100%|██████████| 100/100 [00:03<00:00, 33.16it/s]
100%|██████████| 100/100 [00:12<00:00,  7.80it/s]
100%|██████████| 100/100 [00:03<00:00, 28.36it/s]
```


Floyd-Warshall_robot
Floyd-Warshall
Average Time (seconds)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
20
40
60
80
100
Number of Nodes


Algorithm Performance Comparison
Bellman-Ford_robot
Bellman-Ford
Average Time (seconds)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
20
40
60
80
100
Number of Nodes


Algorithm Performance Comparison
Floyd-Warshall
Bellman-Ford
Average Time (seconds)
0.2
0.3
0.4
0.5



Видно, що алгоритм Белмана-Форда встроєний в пайтон довший за Богдановий так само з Флойда-Воршала, взагалом помітно,що Воршала трохи швидший за Белмана. Імплементація Белмана-Форда не ідеальна, тому Воршала виходить швидше.

Загальний підсумок: Вбудовані алгоритми працюють гірше за імплементовані нами. Перевагою алгоритму Прима є його складність, яка є кращою за алгоритм Крускала. Тому алгоритм Прима корисний для роботи зі щільними графами з великою кількістю ребер. Однак, алгоритм Прима не дозволяє нам багато контролювати вибрані ребра, коли зустрічається декілька ребер з однаковою вагою. Але імпліментація Матвія виявилась ефективнішою, тому Крускал працює швидше

Флойда-Уоршалла: Може обробляти графи з від'ємною вагою ребер, але не може обробляти графи з від'ємною вагою циклів, оскільки не завершується у таких випадках. Беллмана-Форда: Може виявити та повідомити про наявність циклів з від'ємною вагою у графі.

Флойда-Уоршалла: Часова складність дорівнює **O(V^3)**. Беллмана-Форда: Часова складність дорівнює **O(V * E)**. Беллман-Форд менш ефективним для щільних графів.