Звіт

Команда: Сампара Софія і Шапошніков Михайло Артемій

Задача: реалізувати алгоритми (Флойда-Воршала і Белмена-Форда) для знаходження шляхів мінімальної ваги між вершинами орієнтованих графів.

1. Алгоритм Флойда-Воршала

```python
def floyd_warshall_algorytm(graph: 'nx.Graph'):
    num_of_nodes = len(graph.nodes())
    list_of_edges = list(graph.edges())
    matrix_of_distances = {i: {j: graph.edges[i, j]['weight'] \
            if (i, j) in list_of_edges else 0 if i == j else inf for j in range(num_of_nodes)} \
            for i in range(num_of_nodes)}
    matrix_of_predecessors = {i: {j: i if i!=j else 0 for j in range(num_of_nodes)} \
            for i in range(num_of_nodes)}
    for k in range(0, num_of_nodes):
        matrix_k = deepcopy(matrix_of_distances)
        for i in range(num_of_nodes):
            for j in range(num_of_nodes):
                matrix_k[i][j] = min(matrix_of_distances[i][j], \
                            matrix_of_distances[i][k] + matrix_of_distances[k][j])
                if matrix_k[i][j] != matrix_of_distances[i][j]:
                    matrix_of_predecessors[i][j] = matrix_of_predecessors[k][j]
        matrix_of_distances = matrix_k
    if any(matrix_of_distances[n][n] < 0 for n in range(len(matrix_of_distances))):
        raise ValueError('Negative cycle detected')
    return {outer_key: {key: value for key, value in matrix_of_predecessors[outer_key].items() \
                if key != outer_key} for outer_key in range(len(matrix_of_predecessors))}, matrix_of_distances
```

Наша реалізація досить велика, адже ми працювали з матрицями відстаней і матрицями попередників. Наш алгоритм повертає кортеж із двох елементів: словник з матриці та словник з матриці відстаней (ключ – початкова вершина, значення – найменша вага шляху між цією вершиною і вершиною під номером індексу). Цей алгоритм досить затратний в часі, але він дозволяє знайти відстані між будь-якою парою вершин у графі, на відміну від іншого алгоритму.

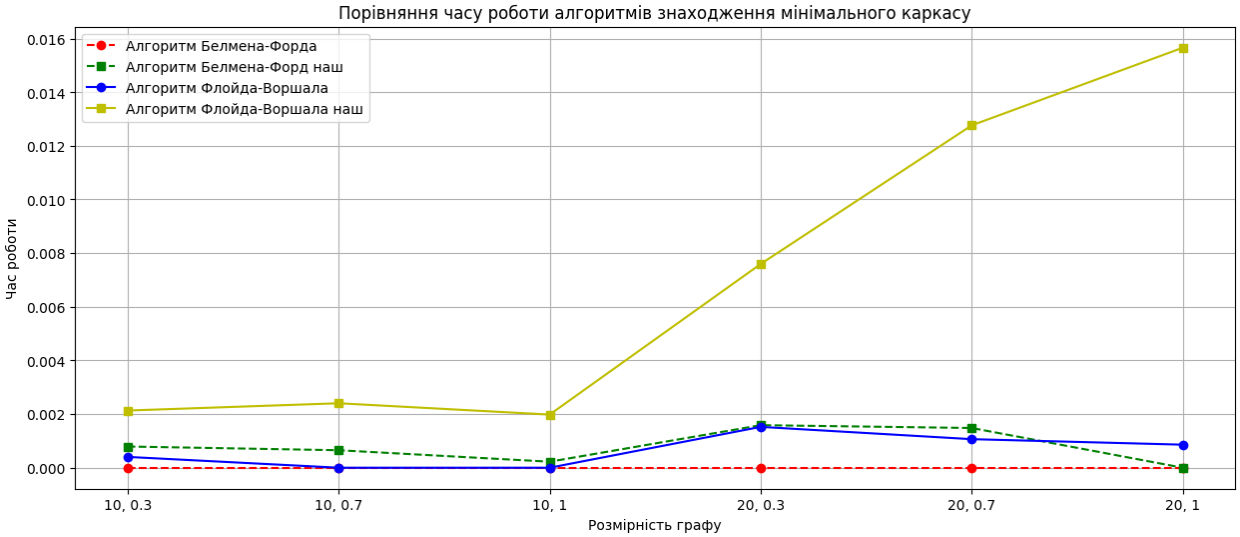
2. Алгоритм Белмена-Форда

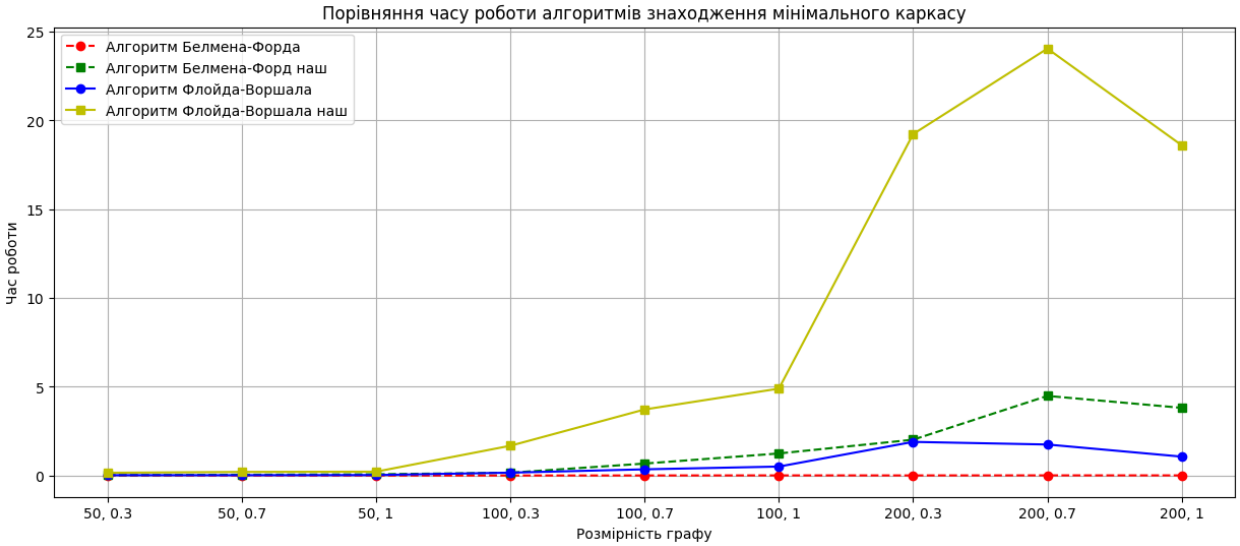
```python
def Bellman_Ford_algorithm(graph: nx.DiGraph, source):
    list_of_ver = list(graph.nodes())
    list_of_edges = list(graph.edges())
    distance = {ver: float('inf') if ver != source else 0 for ver in list(graph.nodes())}
    predecessor = {ver: None for ver in list_of_ver}
    for i in range(1, len(list_of_ver)):
        for u, v in list_of_edges:
            new_value = distance[u] + graph.edges()[u, v]['weight']
            if distance[u] + graph.edges()[u, v]['weight'] < distance[v]:
                distance[v] = new_value
                predecessor[v] = u
    for u, v in list_of_edges:
        if distance[u] + graph.edges()[u, v]['weight'] < distance[v]:
            raise ValueError("Negative cycle detected")
    return predecessor, distance
```

Цей алгоритм ми написали по псевдо-коду, тому його реалізація вийшла досить компактною. Алгоритм Белмена-Форда шукає шляхи найкоротшої

ваги від початкової заданої вершини (source) до всіх інших вершин графа. Повертає наш алгоритм кортеж з двох значень: словник, що представляє матрицю попередників, та словник відстаней, в якому ключем є вершина, а значенням найменша вага шляху між початковою вершиною та цією.

Порівняння результатів





Наш алгоритм Белмена-Форда працює досить добре (на рівні з вбудованим), а Флойда-Воршала працює набагато довше. Це пов’язано з тим, що наш алгоритм працює з матрицями.

Обрахунок результатів

```
NUM_OF_ITERATIONS = 10
time_bellman_ford = []
time_bellman_ford_our = []
time_floyd_warshal = []
time_floyd_warshal_our = []
for nodes in [10, 20, 50, 100, 200]:
    for completeness in [0.3, 0.7, 1]:
        time_taken1 = 0
        time_taken2 = 0
        time_taken3 = 0
        time_taken4 = 0
        for i in tqdm(range(NUM_OF_ITERATIONS)):

# note that we should not measure time of graph creation
            G = gnp_random_connected_graph(nodes, completeness, True)

            start = time.time()
            bellman_ford_predecessor_and_distance(G, 0)
            end = time.time()
            time_taken1 += end - start

            start = time.time()
            Bellman_Ford_algorithm(G, 0)
            end = time.time()
            time_taken2 += end - start

            start = time.time()
            floyd_warshall_predecessor_and_distance(G)
            end = time.time()
            time_taken3 += end - start

            start = time.time()
            floyd_warshall_algorytm(G)
            end = time.time()
            time_taken4 += end - start

        time_bellman_ford.append(time_taken1 / NUM_OF_ITERATIONS)
        time_bellman_ford_our.append(time_taken2 / NUM_OF_ITERATIONS)
        time_floyd_warshal.append(time_taken3 / NUM_OF_ITERATIONS)
        time_floyd_warshal_our.append(time_taken4 / NUM_OF_ITERATIONS)

print(time_bellman_ford)
print(time_bellman_ford_our)
print(time_floyd_warshal)
print(time_floyd_warshal_our)
```

Висновок: наші імплементації алгоритмів працюють повільніше, ніж вбудовані, хоча алгоритм Белмена-Форда не набагато відстає. Алгоритм Флойда-Воршала працює повільніше на алгоритм Белмена-Форда (як наш, так і вбудований), адже він визначає відстані між усіма вершинами графа, на відміну від другого.