Лабораторна робота 1

Дискретна математика

Студенти: Студент Едвард, Багатир-Захарченко Вероніка.

Обов’язки: Вероніка – алгоритм Крускала та Прима. Складання звіту та робота з репозиторієм, порівняння роботи графів.

Едвард – алгоритм Белмана-Форда та алгоритм Флойда-Воршала, складання звіту та робота з репозиторієм

Нам необхідно було дослідити роботу і використати алгоритми Крускала, Прима, Флойда-Воршала та Белмана – Форда. Час виконання наших алгоритмів необхідно було порівняти із часом роботи вбудованих алгоритмів.

На ввід нам дається граф. Ми можемо змінити кількість його вершин (4), наповнюваність (1), орієнтований чи неорієнтований (False), вивід графіку (True).

```
G = gnp_random_connected_graph(4, 1, False, True)
```

Якщо ймовірність заповнювання дорівнює 1 то ми можемо побудувати повний граф. Якщо менше 1, то є ймовірність побудувати неповний граф.

Під час лабораторної роботи ми мали виконати наступне:

1) Дослідити роботу алгоритмів Крускала і Прима. Порівняти їх роботу із вбудованими.
2) Дослідити роботу алгоритмів Флойда-Воршала і Белмана-Форда. Порівняти їх роботу із вбудованими.

При порівнянні нам необхідно було зробити перевірку на 10, 20, 50, 100, 200 чи 500 вершин.

При цьому варто було змінювати вище зазначені 4 параметри.

Для Белмана – Форда та Флойда – Воршала варто було зробити перевірку на від’ємні ваги та цикли. Наприклад, Белмана – Форда не працював би при негативних циклах.

Код алгоритма Краскала та Прима:

Краскал:

```python
def kruskal(graph):
    inputka = graph.edges(data = True)
    new_inputka=[]
    karkas=[]
    edges=set()
    for i in inputka:
        for n in i[2].values():
            new_inputka.append([i[0], i[1], n])
    for i in inputka:
        edges.add(i[0])
        edges.add(i[1])
    end=max(edges)
    setik=set()
    tupok=None
    i=0
    while i<end:
        mini=20000
        for tup in new_inputka:
            if tup[0] and tup[1] in setik:
                new_inputka.remove(tup)
            elif tup[2]<mini and (tup[0] not in setik or tup[1] not in setik):
                mini=tup[2]
                tupok=tup
        setik.add(tupok[0])
        setik.add(tupok[1])
        karkas.append(tupok)
        i+=1
    if len(karkas)!=end:
        return 'Граф не зв’язний'
    suma=0
    rez=[]
    for i in karkas:
        suma+=i[2]
        rez.append((i[0], i[1]))
    return (rez, suma)
# print(kruskal(G))
```

Цей код працює за рахунок пошуку найменшого ребра графа, обидві вершини якого не можуть бути в множині використаних вершин

Алгоритм Прима:

```
def prim(graph):
    inputka = graph.edges(data = True)
    new_inputka=[]
    karkas=[]
    setik=set()
    minimum=2000
    nodes = list(G.nodes())
    end=max(nodes)
    k=0
    for i in inputka:
        for n in i[2].values():
            new_inputka.append([i[0], i[1], n])
            if n < minimum:
                tup = [i[0], i[1], n]
                minimum=n
    karkas.append((tup))
    new_inputka.remove(tup)
    setik.add(tup[0])
    setik.add(tup[1])
    while k<end-1:
        minimum=2000
        for t in new_inputka:
            if t[0] and t[1] in setik:
                new_inputka.remove(t)
            elif (t[0] in setik or t[1] in setik) and t[2]<minimum:
                tup=t
                minimum=t[2]
        setik.add(tup[0])
        setik.add(tup[1])
        karkas.append((tup))
        new_inputka.remove(tup)
        k+=1
        rez=[]
    suma=0
    print
    for i in karkas:
        suma+=i[2]
        rez.append((i[0], i[1]))
    return (rez, suma)
# print(prim(G))
```

Головна суть алгоритму обійти всі вершини і зупинитись на вершині з найменшою вагою, яка якось прив’язана до вже наявних вершин, які знаходяться в сеті

Алгоритм Флойда-Воршалла та алгоритм Беллмана – Форда:

Флойда – Воршала:

Ці алгоритми повинні шукати найкоротший шлях від певної початкової вершини до будь-якої іншої.

При цьому варто зауважити, що граф може мати ребра з від’ємними вагами і, відповідно, від’ємні цикли. Це пов’язано з тим, що код Белмана-Форда та Флойда-Воршала не працює коректно при від’ємних циклах.

Алгоритм Флойда – Воршалла:

алгоритм для знаходження найкоротших відстаней між усіма вершинами зваженого графа без циклів з від'ємними вагами:

```python
def floyd_warshall(graph_):
    '''
    Floyd-Warshall algorithm
    '''
    graph = graph_.edges (data = True)
    num_vertices = max(*([elem[0] for elem in graph]), *([elem[1] for elem in graph]
    dist = []
    for _ in range(num_vertices):
        dis=[float('inf')] * num_vertices
        dist.append(dis)
    # кожному ребру надаємо вагу
    for u, v, data in graph:
        dist[u][v] = data['weight']
    # діагональні елементи рівні нулю
    for i in range(num_vertices):
        dist[i][i] = 0
    # власне алгоритм Флойда-Воршала
    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    text=''
    for source in range(num_vertices):
        text+=f"Distances with {source} source: "
        distances_output = {}
        for v in range(num_vertices):
            if dist[source][v] != float('inf'):
                distances_output[v] = dist[source][v]
            else:
                distances_output[v] = 'inf'
        text+=f'{distances_output}\n'
    return text.strip()
# print(floyd_warshall(G))
```

Наша функція на ввід отримує: graph_ - власне граф. Далі за допомогою `graph = graph_.edges (data = True)` ми отримує список з

ребрами. Кожне ребро в цьому списку представлене у вигляді кортежу (u, v, data), де u та v - вершини, які з'єднує це ребро, а data - додаткова інформація про ребро, наприклад, його вага.

```
num_vertices = max(*([elem[0] for elem in graph]), *([elem[1] for elem in graph])
dist = []
for _ in range(num_vertices):
    dis=[float('inf')] * num_vertices
    dist.append(dis)
```

Далі ми ініціалізуємо матрицю, в якій на початку усі вершини отримують значення нескінченності. Початкову матрицю зберігаємо у список dist.

Наступним кроком є встановлення ваги для кожного ребра графа:

```
# кожному ребру надаємо вагу
for u, v, data in graph:
    dist[u][v] = data['weight']
```

Ми знаємо, що значення діагональних елементів у Флойді-Воршалі дорівнює 0, тому ми прописуємо це у нашій функції

```
for i in range(num_vertices):
    dist[i][i] = 0
```

Далі уже маємо власне основну частину алгоритму Флойда-Воршалла:

```
# власне алгоритм Флойда-Воршала
for k in range(num_vertices):
    for i in range(num_vertices):
        for j in range(num_vertices):
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] = dist[i][k] + dist[k][j]
text=''
for source in range(num_vertices):
    text+=f"Distances with {source} source: "
    distances_output = {}
    for v in range(num_vertices):
        if dist[source][v] != float('inf'):
            distances_output[v] = dist[source][v]
        else:
            distances_output[v] = 'inf'
    text+=f'{distances_output}\n'
return text.strip()
```

Для кожної вершини k, два вкладені цикли перебирають усі можливі пари вершин i та j і перевіряють, чи можна скоротити шлях з вершини i до вершини j, пройшовши через вершину k. Якщо так, то відстань з i до j оновлюється, встановлюючи її рівним сумі відстаней від i до k та від k до j.

Після завершення алгоритму матриця dist містить усі можливі найкоротші відстані між вершинами графа.

Щоб подати наші результати у гарному вигляді ми використовуємо такий шаблон:

```
text=''
for source in range(num_vertices):
    text+=f"Distances with {source} source: "
    distances_output = {}
    for v in range(num_vertices):
        if dist[source][v] != float('inf'):
            distances_output[v] = dist[source][v]
        else:
            distances_output[v] = 'inf'
    text+=f'{distances_output}\n'
return text.strip()
```

Ось порівняння швидкості нашого і вбудованого алгоритмів Флойда-Воршалла. Як бачимо, результати майже ідентичні, однак наші результати дещо повільніші. На фрагментах коду видно умови за яких ми перевіряли обидва алгоритми (кількість ітерацій, кількість вершин, ймовірність наповнення графа, орієнтованість):

```
NUM_OF_ITERATIONS = 200
```

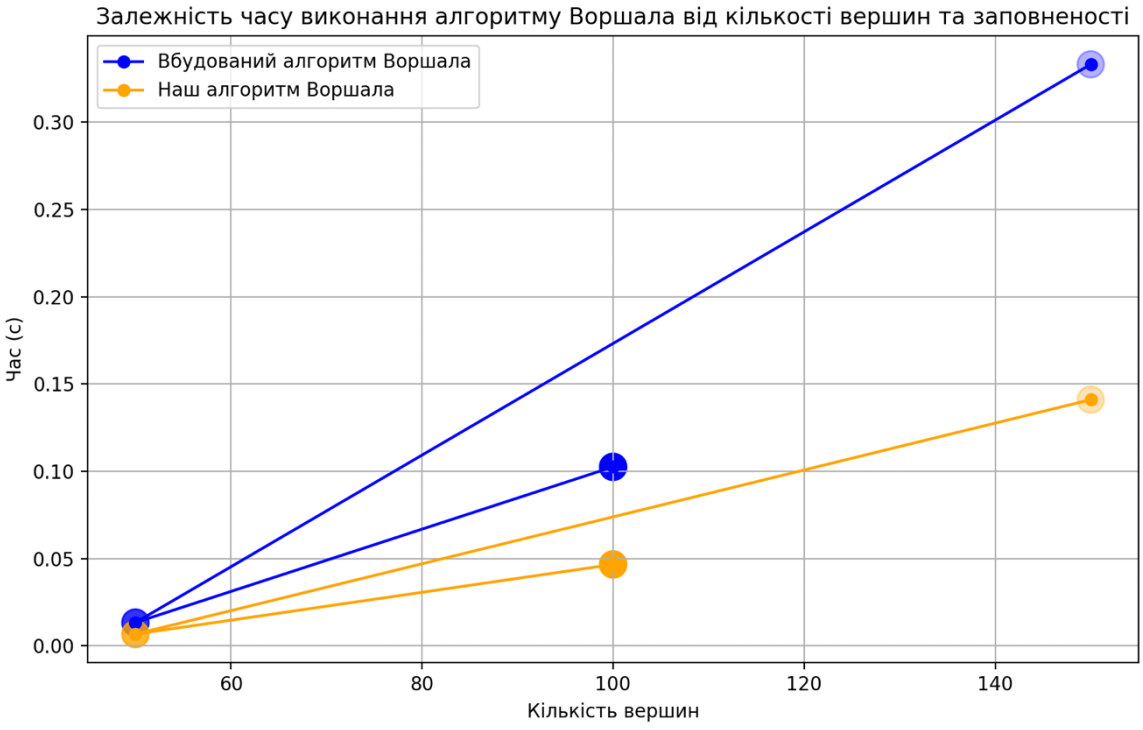
```
G = gnp_random_connected_graph(50, 0.5, True)
```

Прикріпляємо порівняння часу роботи вбудованого алгоритму і нашого для неорієнтованого графа:

```python
# tree.minimum_spanning_tree(G, algorithm="kruskal")


NUM_OF_ITERATIONS = 100
time_taken = 0
time_taken1=0
for i in tqdm(range(NUM_OF_ITERATIONS)):

    # note that we should not measure time of graph creation
    G = gnp_random_connected_graph(150, 0.2, False)

    start = time.time()
    floyd_warshall_predecessor_and_distance(G)
    end = time.time()
    start1 = time.time()
    floyd_warshall(G)
    end1 = time.time()

    time_taken += end - start
    time_taken1  += end1-start1

print(time_taken / NUM_OF_ITERATIONS)
print(time_taken1 / NUM_OF_ITERATIONS)
```

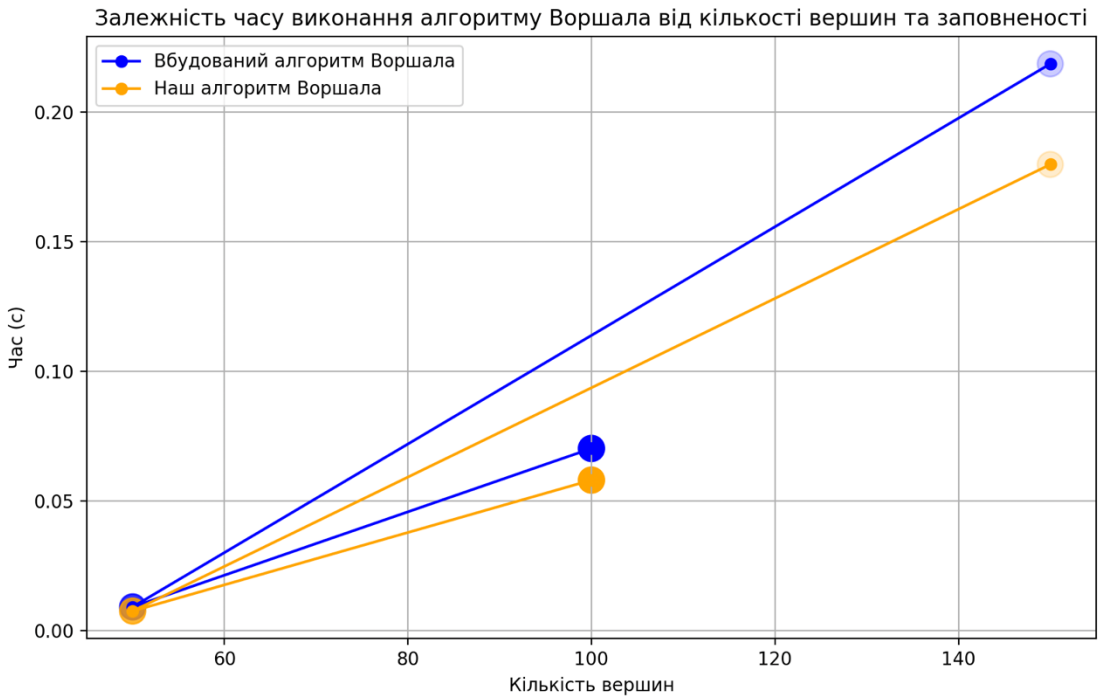
✓ 47.8s

```
100%|          | 100/100 [00:47<00:00,  2.09it/s]
0.3331905436515808
0.14104789972305298
```



Прикріпляємо порівняння часу роботи вбудованого алгоритму і нашого для орієнтованого графа:

```
    # tree.minimum_spanning_tree(G, algorithm="kruskal")


    NUM_OF_ITERATIONS = 100
    time_taken = 0
    time_taken1=0
    for i in tqdm(range(NUM_OF_ITERATIONS)):

        # note that we should not measure time of graph creation
        G = gnp_random_connected_graph(100, 1, True)

        start = time.time()
        floyd_warshall_predecessor_and_distance(G)
        end = time.time()
        start1 = time.time()
        floyd_warshall(G)
        end1 = time.time()

        time_taken += end - start
        time_taken1  += end1-start1

    print(time_taken / NUM_OF_ITERATIONS)
    print(time_taken1 / NUM_OF_ITERATIONS)
97]  ✓ 13.3s                                                    Pytho
..  100%|██████████| 100/100 [00:13<00:00,  7.50it/s]
    0.07025751113891601
    0.058015990257263186
```



Видно, що, наприклад, при однакових значеннях повноти наш алгоритм працює швидше.

Алгоритм Белмана-Форда:

Він виконує так ж функцію, як Флойда-Воршалла: шукає найкоротші шляхи між вершинами графа. На ввід наша функція отримує граф та стартову точку. Далі ми «витягуємо» ребра з графа і їхні ваги.

```
def bellman_ford(graph, start):

    inputka = graph.edges(data = True)
```

Для кожної вершини спочатку задаємо нескінченність. Для початкової точки – 0.

```
n = len(inputka)
distances = {node: float('inf') for node in range(n)}
distances[start] = 0
```

Далі за, власне, алгоритмом Беллмана-Форда ми оновлюємо значення дистанції до вершин:

```
for _ in range(n - 1):
    for u, v, data in inputka:
        if distances[u] + data['weight'] < distances[v]:
            distances[v] = distances[u] + data['weight']
```

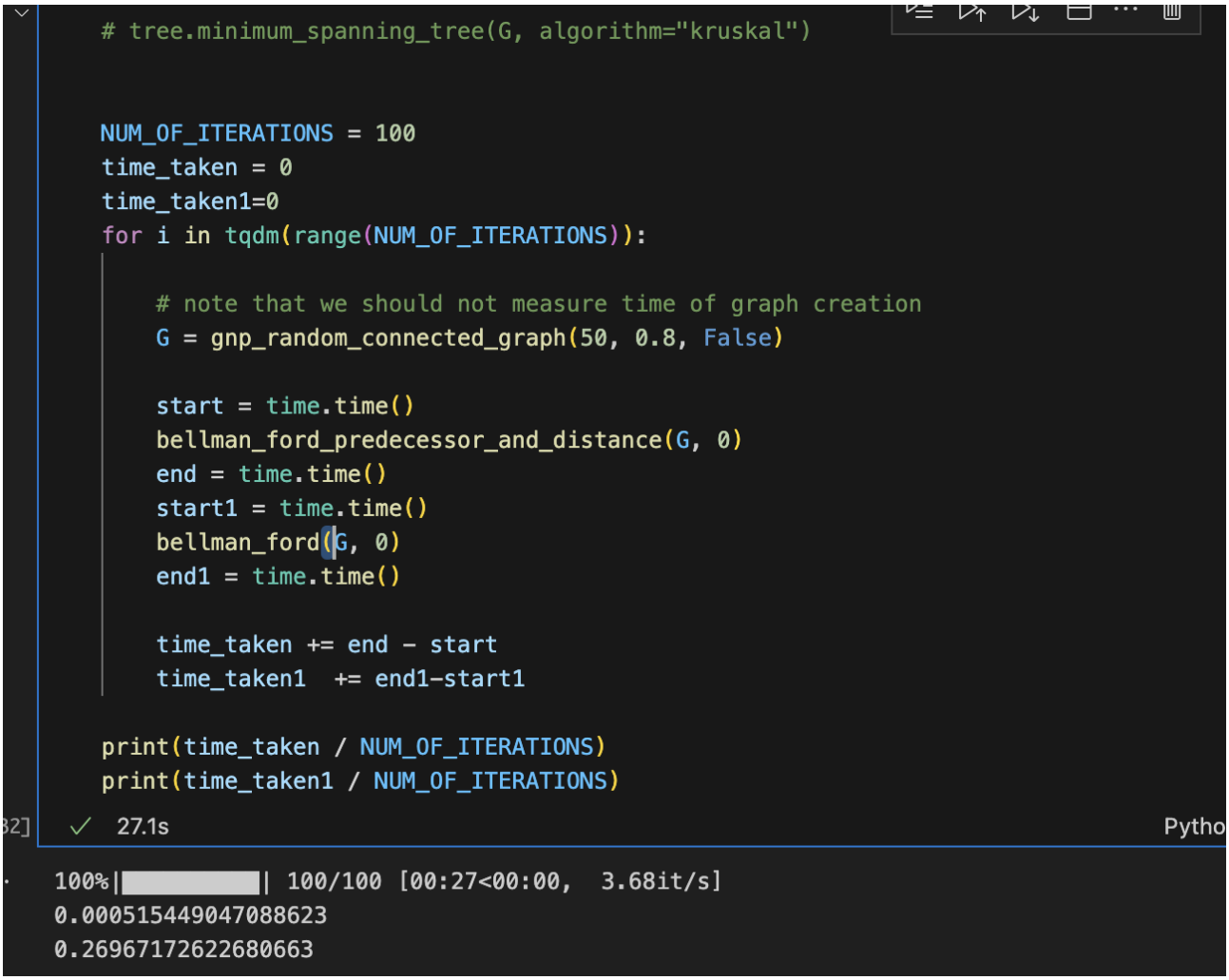
Якщо у графі є від’ємні цикли, то алгоритм припиняє роботу і видає відповідне повідомлення.

```
    for u, v, data in inputka:
        if distances[u] + data['weight'] < distances[v]:
            return 'Negative cycle detected'
```

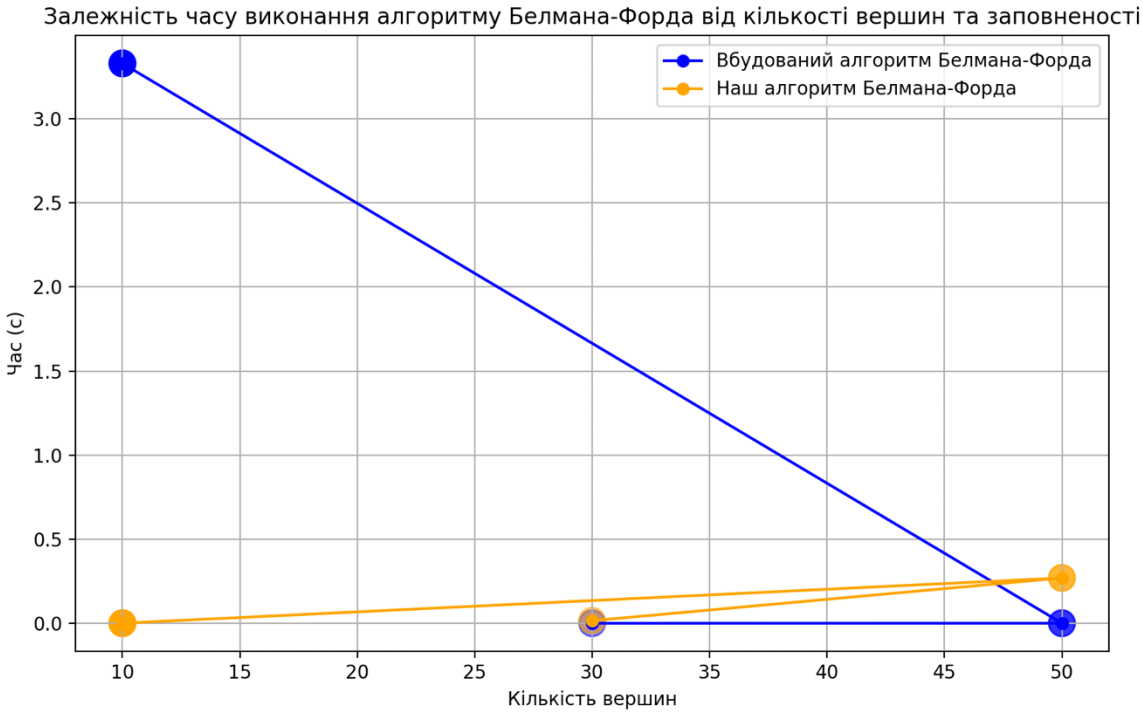
Зберігаємо вивід функції у відповідному форматі:

```
res = ''
for vertex, distance in distances.items():
    if distance != float('inf'):
        res += f"Distance to {vertex}: {distance}\n"
return res
```

Порівняємо швидкість нашого алгоритму та вбудованого для неорієнтованого графа:



```
    # tree.minimum_spanning_tree(G, algorithm="kruskal")


    NUM_OF_ITERATIONS = 100
    time_taken = 0
    time_taken1=0
    for i in tqdm(range(NUM_OF_ITERATIONS)):

        # note that we should not measure time of graph creation
        G = gnp_random_connected_graph(50, 0.8, False)

        start = time.time()
        bellman_ford_predecessor_and_distance(G, 0)
        end = time.time()
        start1 = time.time()
        bellman_ford(G, 0)
        end1 = time.time()

        time_taken += end - start
        time_taken1  += end1-start1

    print(time_taken / NUM_OF_ITERATIONS)
    print(time_taken1 / NUM_OF_ITERATIONS)
32]  ✓ 27.1s                                                    Pytho
100%|██████████| 100/100 [00:27<00:00,  3.68it/s]
0.000515449047088623
0.26967172622680663
```

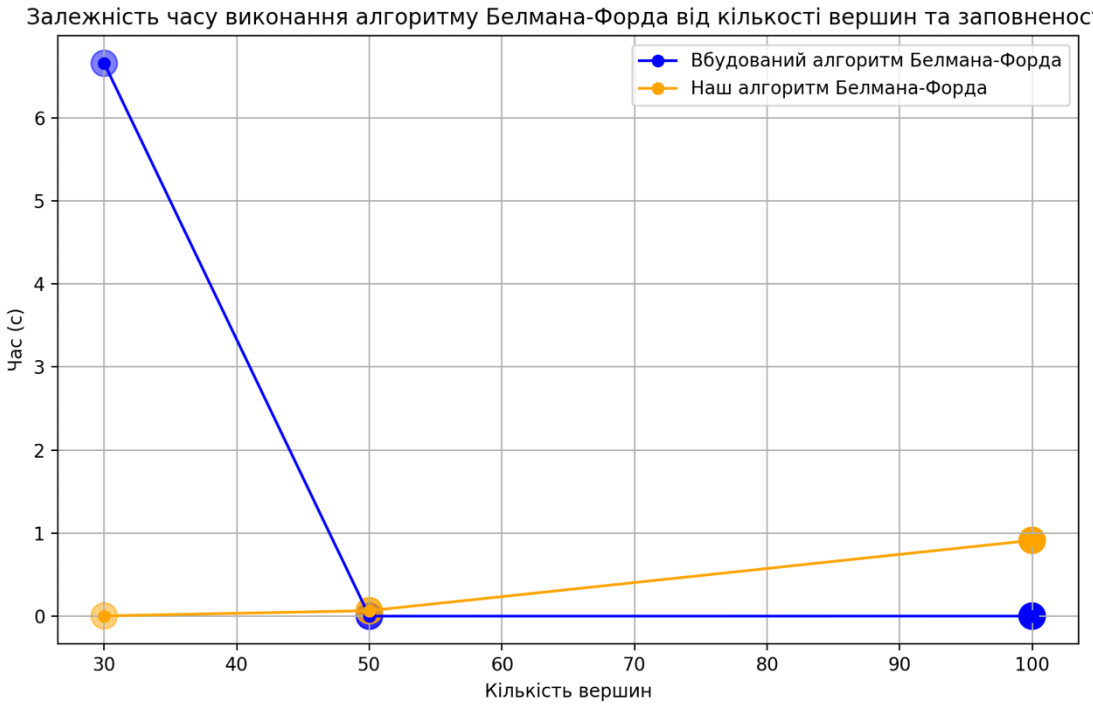


Порівняємо швидкість нашого алгоритму та вбудованого для орієнтованого графа:

```
    # tree.minimum_spanning_tree(G, algorithm="kruskal")


    NUM_OF_ITERATIONS = 100
    time_taken = 0
    time_taken1=0
    for i in tqdm(range(NUM_OF_ITERATIONS)):

        # note that we should not measure time of graph creation
        G = gnp_random_connected_graph(30, 0.5, True)

        start = time.time()
        bellman_ford_predecessor_and_distance(G, 0)
        end = time.time()
        start1 = time.time()
        bellman_ford(G
        end1 = time.ti
                                (variable) end: float
                                end

        time_taken += end - start
        time_taken1  += end1-start1

    print(time_taken / NUM_OF_ITERATIONS)
    print(time_taken1 / NUM_OF_ITERATIONS)
```

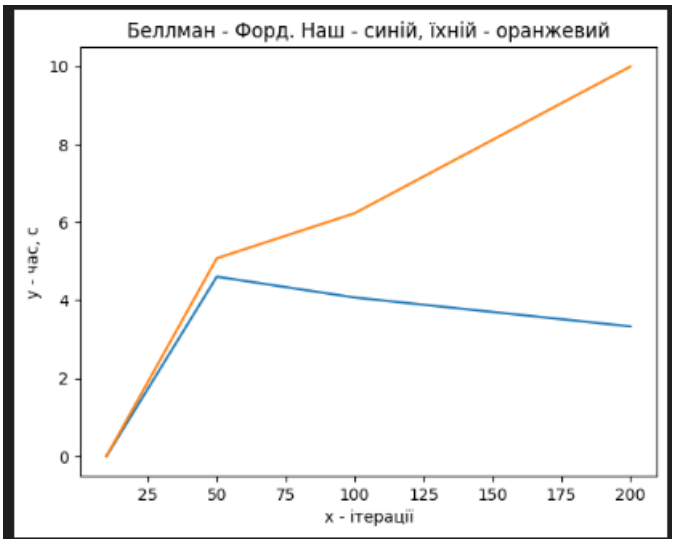
29] ✓ 0.3s

```
100%|██████████| 100/100 [00:00<00:00, 270.60it/s]
6.65736198425293e-05
0.003383157253265381
```



Можемо зробити висновок, що наш алгоритм буде краще працювати для маленьких графів, а вбудований для великих

Також можна порівняти наші алгоритмі в ітерацій до часу:



Як бачимо. В залежності від кількості ітерацій зроблений нами алгоритм працює повільніше.

Порівняння швидкостей алгоритмів Прима та Краскала

Розмір крапочки відповідає за заповненість(більша крапочка – більша заповненість

```
# tree.minimum_spanning_tree(G, algorithm="kruskal")


NUM_OF_ITERATIONS = 100
time_taken = 0
time_taken1=0
for i in tqdm(range(NUM_OF_ITERATIONS)):

    # note that we should not measure time of graph creation
    G = gnp_random_connected_graph(50, 0.4, False)

    start = time.time()
    tree.minimum_spanning_tree(G, algorithm="kruskal")
    end = time.time()
    start1 = time.time()
    kruskal(G)
    end1 = time.time()

    time_taken += end - start
    time_taken1  += end1-start1

print(time_taken / NUM_OF_ITERATIONS)
print(time_taken1 / NUM_OF_ITERATIONS)
```

✓ 0.1s Pytho

```
100%|          | 100/100 [00:00<00:00, 599.66it/s]
0.0005108261108398437
0.000684046745300293
```

```python
    # tree.minimum_spanning_tree(G, algorithm="kruskal")


    NUM_OF_ITERATIONS = 100
    time_taken = 0
    time_taken1=0
    for i in tqdm(range(NUM_OF_ITERATIONS)):

        # note that we should not measure time of graph creation
        G = gnp_random_connected_graph(100, 0.8, False)

        start = time.time()
        tree.minimum_spanning_tree(G, algorithm="prim")
        end = time.time()
        start1 = time.time()
        prim(G)
        end1 = time.time()

        time_taken += end - start
        time_taken1  += end1-start1

    print(time_taken / NUM_OF_ITERATIONS)
    print(time_taken1 / NUM_OF_ITERATIONS)
```

2] ✓ 1.9s

```
100%|          | 100/100 [00:01<00:00, 51.15it/s]
0.001706874370574951
0.014112930297851562
```

```python
# tree.minimum_spanning_tree(G, algorithm="kruskal")


NUM_OF_ITERATIONS = 100
time_taken = 0
time_taken1=0
for i in tqdm(range(NUM_OF_ITERATIONS)):

    # note that we should not measure time of graph creation
    G = gnp_random_connected_graph(150, 1, False)

    start = time.time()
    tree.minimum_spanning_tree(G, algorithm="prim")
    end = time.time()
    start1 = time.time()
    prim(G)
    end1 = time.time()

    time_taken += end - start
    time_taken1  += end1-start1

print(time_taken / NUM_OF_ITERATIONS)
print(time_taken1 / NUM_OF_ITERATIONS)
```

✓ 7.4s Python

```
100%|          | 100/100 [00:07<00:00, 13.35it/s]
0.005383629798889161
0.06153529405593872
```

```python
NUM_OF_ITERATIONS = 100
time_taken = 0
time_taken1=0
for i in tqdm(range(NUM_OF_ITERATIONS)):

    # note that we should not measure time of graph creation
    G = gnp_random_connected_graph(100, 0.8, False)

    start = time.time()
    tree.minimum_spanning_tree(G, algorithm="kruskal")
    end = time.time()
    start1 = time.time()
    kruskal(G)
    end1 = time.time()

    time_taken += end - start
    time_taken1  += end1-start1

print(time_taken / NUM_OF_ITERATIONS)
print(time_taken1 / NUM_OF_ITERATIONS)
```

✓ 1.5s

```
100%|██████████| 100/100 [00:01<00:00, 64.67it/s]
0.0030546045303344728
0.009496562480926514
```

Можемо спостерігати, що наш алгоритм працює повільніше, але не менш точно

Отже, ми провели відповідні дослідження наших алгоритмів і зрозуміли за яких умов наш алгоритм кращий чи гірший за вбудований. Наш алгоритм Флойда-Воршала працює краще за вбудований, якщо порівнювати за параметрами кількості вершин та заповненості. Наш алгоритм Белмана-Форда працює довше: це видно в залежності часу до кількості ітерацій. Загалом наші алгоритми Крускала і Прима працюють трішки повільніше, аніж вбудовані, що видно на графіках.

Загалом ці 4 алгоритми чудово підходять для розв’язання задач про найкоротший шлях.