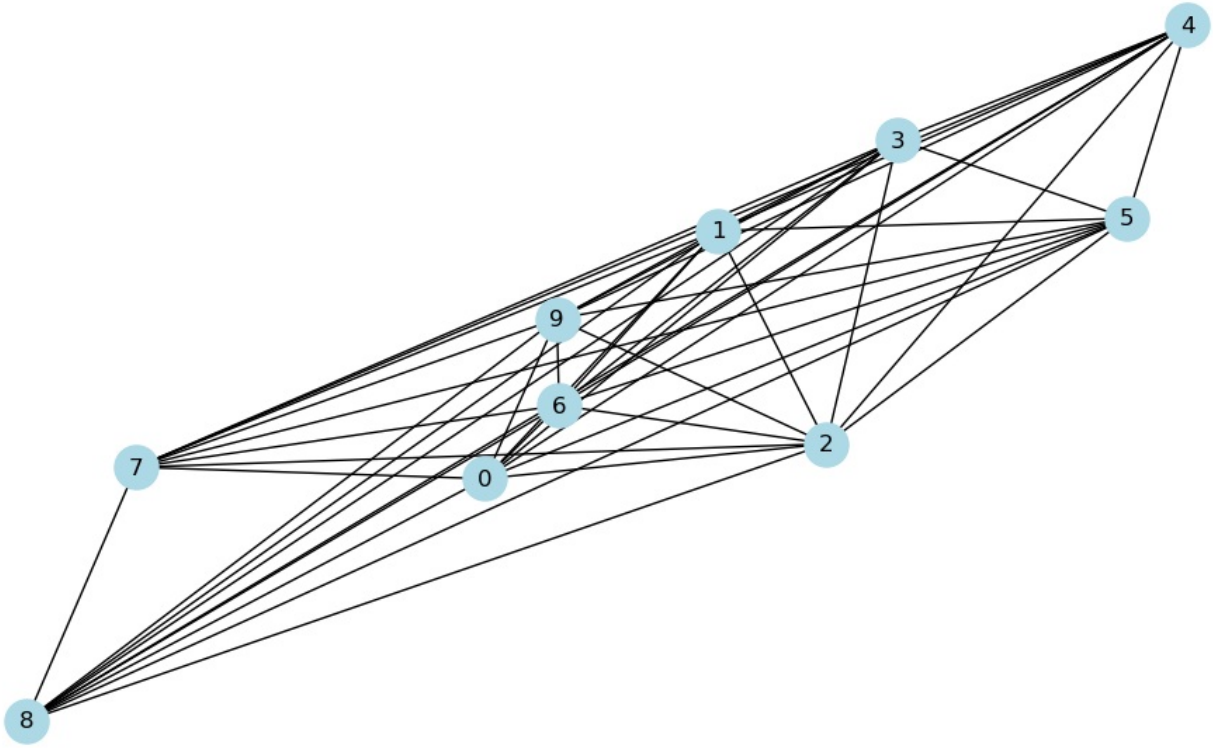
In [1]:

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

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)
        # w = 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(10, 1, False, True)
```



Павлосюк Роман Коваль Вікторія Мета: реалізувати алгоритми, вивчені на лекціях, та порівняти їх з вдудованими Нижче написані алгоритми Флойда Воршалла та Беллмана Форда

In [3]:

```
# pred is a dictionary of predecessors, dist is a dictionary of distances
start=time.time()
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")
end=time.time()
taken_time=end-start
print('вбудований',taken_time)
```

```
Negative cycle detected
вбудований 0.0011115074157714844
```

У функції **Bellman_Ford_algorithm** ми реалізували алгоритм Белмана-Форда. Спочатку ми знайшли загальну кількість вершин, а потім створили список з відстанями від кожної вершини до початкової вершини, спочатку це нескінченність. Потім ми задали відстань до початкової вершини **0(**по дефолту першою вершиною ми обрали **0)** Перший цикл **for** здійснює **V-1(V -** кількість вершин у графі**)** ітерацій. У цьому циклі для кожного ребра перевіряється, чи можливо зменшити відстань до кінцевої вершини, оновлюючи значення відстані, якщо відстань до початкової вершини плюс вага ребра менше поточної відстані до кінцевої вершини. Наступний цикл відповідає за перевірку на наявність від'ємного циклу. Для цього знову перевіряємо кожне ребро, і якщо для будь-якого ребра можна зменшити відстань, то повертається повідомлення, що знайдено від'ємний цикл. Останнім циклом ми просто виводимо результат виконаного алгоритму

In [4]:

```
start=time.time()
def Bellman_Ford_algorithm(edges=list(G.edges(data=True)), first_node=0):
    num_nodes = max(max(edge[0], edge[1]) for edge in edges) + 1
    distance = [float('inf')] * num_nodes
    distance[first_node] = 0
    summary=''
    for _ in range(num_nodes - 1):
```

```
        for source, dest, edge_data in edges:
            weight = edge_data['weight']
            if distance[dest] > distance[source] + weight:
                distance[dest] = distance[source] + weight
    for source, dest, edge_data in edges:
        weight = edge_data['weight']
        if distance[dest] > distance[source] + weight:
            return "Negative cycle detected"
    for k, v in enumerate(distance):
        summary += f"Distance to {k}: {v}\n"
    return summary
print(Bellman_Ford_algorithm())
end=time.time()
taken_time=end-start
print('мій',taken_time)
```

```
Distance to 0: 0
Distance to 1: 19
Distance to 2: 17
Distance to 3: 1
Distance to 4: -5
Distance to 5: -2
Distance to 6: -2
Distance to 7: -8
Distance to 8: -9
Distance to 9: -4

мій 0.0004475116729736328
```

In [5]:

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

```
Distances with 0 source: {0: -106792, 2: -106794, 1: -106793, 3: -106795, 4: -106803, 5:
-106855, 6: -107163, 7: -109017, 8: -120142, 9: -186888}
Distances with 1 source: {1: -106794, 0: -106793, 5: -106856, 2: -106795, 3: -106796, 4:
-106804, 6: -107164, 7: -109018, 8: -120143, 9: -186889}
Distances with 2 source: {2: -106796, 0: -106794, 1: -106795, 3: -106797, 4: -106805, 5:
-106857, 6: -107165, 7: -109019, 8: -120144, 9: -186890}
Distances with 3 source: {3: -106798, 0: -106795, 1: -106796, 2: -106797, 5: -106858, 4:
-106806, 6: -107166, 7: -109020, 8: -120145, 9: -186891}
Distances with 4 source: {4: -106814, 0: -106803, 1: -106804, 2: -106805, 3: -106806, 6:
-107174, 5: -106866, 7: -109028, 8: -120153, 9: -186899}
Distances with 5 source: {5: -106918, 0: -106855, 1: -106856, 2: -106857, 3: -106858, 4:
-106866, 6: -107226, 7: -109080, 8: -120205, 9: -186951}
Distances with 6 source: {6: -107534, 0: -107163, 1: -107164, 2: -107165, 3: -107166, 4:
-107174, 5: -107226, 9: -187259, 7: -109388, 8: -120513}
Distances with 7 source: {7: -111242, 0: -109017, 1: -109018, 2: -109019, 3: -109020, 4:
-109028, 5: -109080, 6: -109388, 9: -189113, 8: -122367}
Distances with 8 source: {8: -133492, 0: -120142, 1: -120143, 2: -120144, 3: -120145, 4:
-120153, 5: -120205, 6: -120513, 7: -122367, 9: -200238}
Distances with 9 source: {9: -266984, 0: -186888, 1: -186889, 2: -186890, 3: -186891, 4:
-186899, 5: -186951, 6: -187259, 7: -189113, 8: -200238}
вбудований 0.0005838871002197266
```

У функції **Floyd_Warshall_algorithm** ми реалізували алгоритм Флойда-Воршала. Спочатку ми знайшли загальну кількість вершин, а потім створили словник **distances_with_source,** який буде зберігати відстані для кожної вершини. Далі ми заходимо в зовнішній цикл, в якому ми визначаємо вершину яку обробляємо(ми проходитемося по абсолютно всіх вершинах) і записуємо відстань від цієї вершини до себе самої як **0.** Дальше нашим внутрішнім циклом ми встановлюємо відстань від нашої вершини до суміжних, а потім ifкою

перевіряємо чи нашу поточну відстань не можна скоротити, якщо так, то перезаписуємо нашу відстань, по завершенню циклу для кожної вихідної вершини зберігається словник **distance**, який містить відстані до всіх вершин від цієї вихідної вершини. Далі перевіряємо на негативний цикл по тому ж принципу, що і в попередньому алгоритмі і так само виводимо результат.

In [6]:

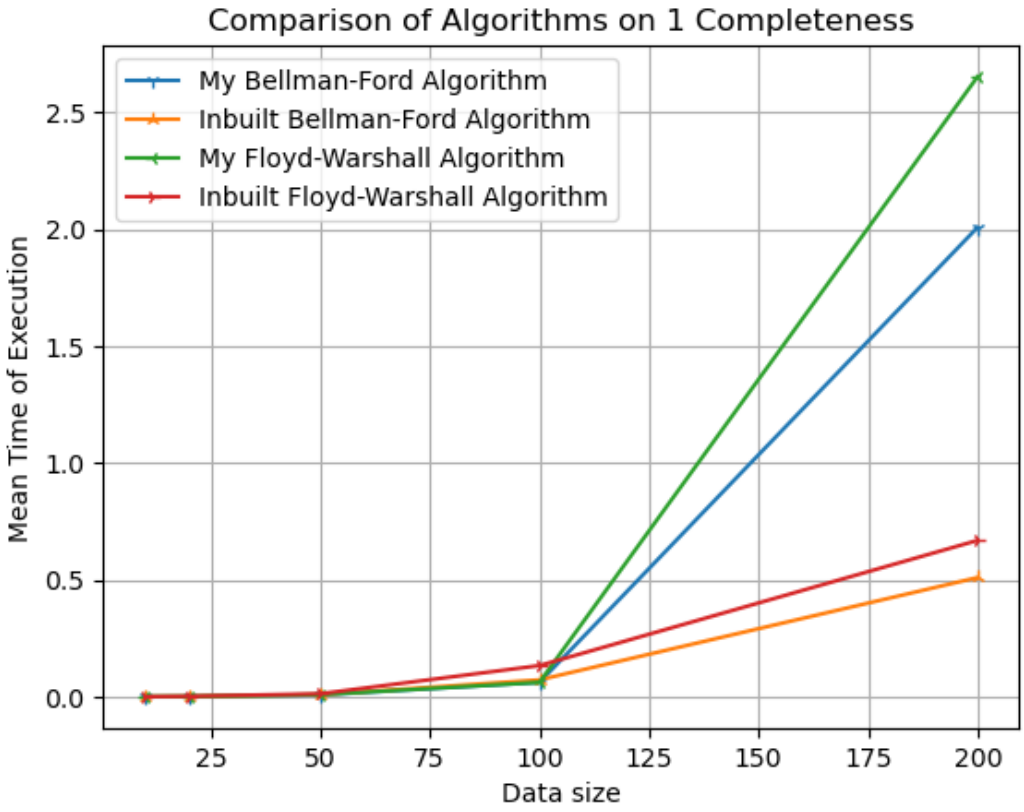
```
start=time.time()
def Floyd_Warshall_algorithm(edges=list(G.edges(data=True))):
    num_nodes = max(max(edge[0], edge[1]) for edge in edges) + 1
    distances_with_source = {}
    summary=''
    for source in range(num_nodes):
        distance = {node: float('inf') for node in range(num_nodes)}
        distance[source] = 0
        for _ in range(num_nodes - 1):
            for source_node, dest_node, edge_data in edges:
                weight = edge_data['weight']
                if distance[dest_node] > distance[source_node] + weight:
                    distance[dest_node] = distance[source_node] + weight
        distances_with_source[source] = distance
        for source_node, dest_node, edge_data in edges:
            weight = edge_data['weight']
            if distance[dest_node] > distance[source_node] + weight:
                return "Negative cycle detected"
    for source, dist in distances_with_source.items():
        dist_str = ', '.join([f"{node}: {dist}" for node, dist in dist.items()])
        summary+=f"Distances with {source} source: {{{dist_str}}}\n"
    return summary
print(Floyd_Warshall_algorithm())
end=time.time()
taken_time=end-start
print('мій',taken_time)
```

```
Distances with 0 source: {0: 0, 1: 19, 2: 17, 3: 1, 4: -5, 5: -2, 6: -2, 7: -8, 8: -9, 9:
-4}
Distances with 1 source: {0: inf, 1: 0, 2: -1, 3: 2, 4: 5, 5: -4, 6: 2, 7: -9, 8: -7, 9:
-6}
Distances with 2 source: {0: inf, 1: inf, 2: 0, 3: 3, 4: 6, 5: -3, 6: 9, 7: -8, 8: -6, 9:
-5}
Distances with 3 source: {0: inf, 1: inf, 2: inf, 3: 0, 4: 14, 5: 15, 6: 9, 7: 5, 8: -5, 9
: 4}
Distances with 4 source: {0: inf, 1: inf, 2: inf, 3: inf, 4: 0, 5: 3, 6: 3, 7: -3, 8: -4,
9: 1}
Distances with 5 source: {0: inf, 1: inf, 2: inf, 3: inf, 4: inf, 5: 0, 6: 17, 7: -5, 8:
-3, 9: -2}
Distances with 6 source: {0: inf, 1: inf, 2: inf, 3: inf, 4: inf, 5: inf, 6: 0, 7: 12, 8:
18, 9: 10}
Distances with 7 source: {0: inf, 1: inf, 2: inf, 3: inf, 4: inf, 5: inf, 6: inf, 7: 0, 8:
12, 9: 13}
Distances with 8 source: {0: inf, 1: inf, 2: inf, 3: inf, 4: inf, 5: inf, 6: inf, 7: inf,
8: 0, 9: 9}
Distances with 9 source: {0: inf, 1: inf, 2: inf, 3: inf, 4: inf, 5: inf, 6: inf, 7: inf,
8: inf, 9: 0}

мій 0.0020971298217773438
```

In [7]:

```
list_num_nodes = [10, 20, 50, 100, 200]
# 1 completeness:
my_bellman_ford = [0.0004022121429443359,0.0008933544158935547,0.007898330688476562,0.06
148552894592285,2.008122444152832]
my_floyd_warshall= [0.0009691715240478516,0.0012805461883544922,0.008960723876953125,0.06
141209602355957,2.6526590347290039]
notmy_bellman_ford = [0.0008509159088134766,0.0016875267028808594,0.008820295333862305,0.
0722653865814209,0.5100231170654297]
notmy_floyd_warshall = [0.0004200935363769531,0.0018966197967529297,0.012864112854003906,
0.13337397575378418,0.6689374446868896]
```

```
plt.plot(list_num_nodes, my_bellman_ford , marker='1', label="My Bellman-Ford Algorithm")
plt.plot(list_num_nodes, notmy_bellman_ford , marker='2', label="Inbuilt Bellman-Ford Alg
orithm")
plt.plot(list_num_nodes, my_floyd_warshall, marker='3', label="My Floyd-Warshall Algorith
m")
plt.plot(list_num_nodes, notmy_floyd_warshall, marker='4', label="Inbuilt Floyd-Warshall
Algorithm")

plt.xlabel('Data size')
plt.ylabel('Mean Time of Execution')
plt.title('Comparison of Algorithms on 1 Completeness')
plt.legend()

plt.grid(True)
plt.show()
```



Мої алгоритм Белмана-Форда і Флойда-Воршала починає значно відставати в часі на великих графах, це може бути зумовлено великою складністю мого алгоритму, так-як він містить вкладені цикли та можливою оптимізацією вбудованого алгоритму, якої у мене немає.

Отже, краще працює, алгоритм Белмана-Форда, так як він містить в собі меншу кількість вкладених циклів, тому його складність коду трохи менша, ніж у Флойда-воршала, не зовсім розумію чому мої алгоритми так сильно відстають на великих графах, але думаю, що у вбудованах все ж застосовується якась оптимізація коду, яка дає змогу не так силбно втрачати час з такою великою кількістю вершин.