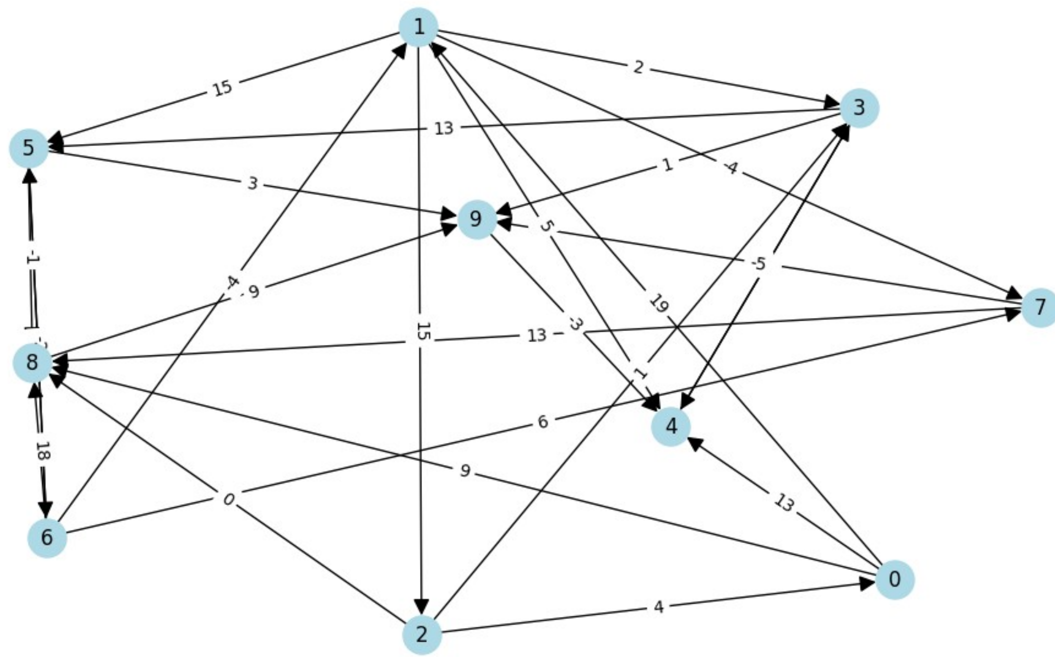
```python
# !pip install networkx
# !pip install matplotlib
# !pip install tqdm

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

Generating graph

```python
# 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:
```

```python
        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, 0.4, True, True)
dict(G.adjacency())
```

```
{0: {1: {'weight': 19}, 4: {'weight': 13}, 8: {'weight': 9}},
 1: {2: {'weight': 15},
  3: {'weight': 2},
  4: {'weight': 5},
  5: {'weight': 15},
  7: {'weight': -4}},
 2: {0: {'weight': 4}, 8: {'weight': 0}, 3: {'weight': 1}},
 3: {4: {'weight': 2}, 5: {'weight': 13}, 9: {'weight': 1}},
 4: {3: {'weight': 4}},
 5: {6: {'weight': 6}, 8: {'weight': -1}, 9: {'weight': 3}},
 6: {1: {'weight': -4},
  5: {'weight': -2},
  8: {'weight': 18},
  7: {'weight': 6}},
 7: {9: {'weight': -5}, 8: {'weight': 13}},
 8: {9: {'weight': 9}},
 9: {4: {'weight': -3}}}
```



Bellman-Ford algorithm

```python
from networkx.algorithms import bellman_ford_predecessor_and_distance

def bellman_ford(graph, source):
    distances = {node: float('inf') for node in graph.nodes()}
    predecessors = {node: None for node in graph.nodes()}
    distances[source] = 0

    for _ in range(len(graph.nodes()) - 1):
        for u, v in graph.edges():
            weight = graph[u][v]['weight']
            if distances[u] + weight < distances[v]:
                distances[v] = distances[u] + weight
                predecessors[v] = u

    for u, v in graph.edges():
        weight = graph[u][v]['weight']
        if distances[u] + weight < distances[v]:
            raise ValueError('Negative cycle detected')

    return predecessors, distances

# biuld-in bellman-ford algorithm
try:
    pred, dist = bellman_ford_predecessor_and_distance(G, 0)
    print(bellman_ford_predecessor_and_distance(G, 0))
```

```
    for k, v in dist.items():
        print(f'Distance to {k}:', v)
except:
    print('Negative cycle detected')
```

```
({0: [], 1: [0], 4: [9], 8: [0], 2: [1], 3: [4], 5: [3], 7: [1], 9:
[7], 6: [5]}, {0: 0, 1: 19, 4: 7, 8: 9, 2: 34, 3: 11, 5: 24, 7: 15, 9:
10, 6: 30})
Distance to 0: 0
Distance to 1: 19
Distance to 4: 7
Distance to 8: 9
Distance to 2: 34
Distance to 3: 11
Distance to 5: 24
Distance to 7: 15
Distance to 9: 10
Distance to 6: 30
```

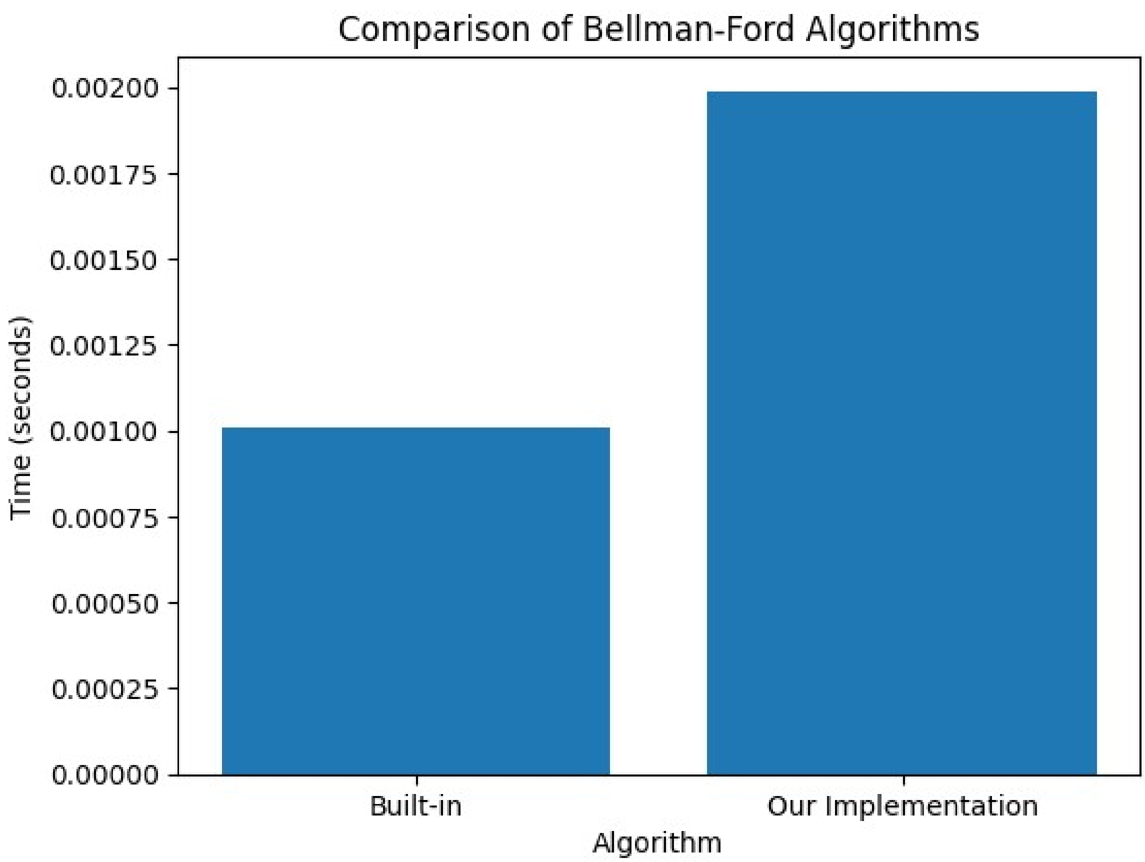
```
# custom bellman-ford algorithm
pred, dist = bellman_ford(G, 0)
print(bellman_ford(G, 0))
for k, v in dist.items():
    print(f'Distance to {k}:', v)
```

```
({0: None, 1: 0, 2: 1, 3: 4, 4: 9, 5: 3, 6: 5, 7: 1, 8: 0, 9: 7}, {0:
0, 1: 19, 2: 34, 3: 11, 4: 7, 5: 24, 6: 30, 7: 15, 8: 9, 9: 10})
Distance to 0: 0
Distance to 1: 19
Distance to 2: 34
Distance to 3: 11
Distance to 4: 7
Distance to 5: 24
Distance to 6: 30
Distance to 7: 15
Distance to 8: 9
Distance to 9: 10
```

```
start1 = time.time()
bellman_ford_predecessor_and_distance(G, 0)
end1 = time.time()

start2 = time.time()
bellman_ford(G, 0)
end2 = time.time()

time_taken1 = end1 - start1
time_taken2 = end2 - start2

print(f'Built-in Bellman-Ford took {time_taken1} seconds')
```

```python
print(f'Our Bellman-Ford took {time_taken2} seconds')
try:
    if time_taken1 > time_taken2:
        print(f'Our implementation is {time_taken1 / time_taken2}
times faster than the built-in one')
    else:
        print(f'The built-in implementation is {time_taken2 /
time_taken1} times faster than ours')
except:
    print('Division by zero')

plt.bar(['Built-in', 'Our Implementation'], [time_taken1,
time_taken2])
plt.xlabel('Algorithm')
plt.ylabel('Time (seconds)')
plt.title('Comparison of Bellman-Ford Algorithms')
plt.show()
```

```
Built-in Bellman-Ford took 0.0010106563568115234 seconds
Our Bellman-Ford took 0.001987934112548828 seconds
The built-in implementation is 1.966973342769521 times faster than
ours
```



Floyd-Warshall algorithm

```
from networkx.algorithms import
floyd_warshall_predecessor_and_distance

def floyd_warshall(graph):
    num_nodes = graph.number_of_nodes()
    distances = [[float('inf')] * num_nodes for _ in range(num_nodes)]

    for i in range(num_nodes):
        if distances[i][i] < 0:
            return 'Negative cycle detected!'

    for u, v, w in graph.edges(data=True):
        distances[u][v] = w['weight']

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

    for k in range(num_nodes):
        for i in range(num_nodes):
            for j in range(num_nodes):
                distances[i][j] = min(distances[i][j], distances[i][k]
+ distances[k][j])

    dist_dict = {}
    for i in range(num_nodes):
        dist_dict[i] = {}
        for j in range(num_nodes):
            dist_dict[i][j] = distances[i][j]

    return dist_dict

# built-in floyd-warshall algorithm
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, 1: 19, 4: 7, 8: 9, 2: 34, 3: 11, 5:
24, 6: 30, 7: 15, 9: 10}
Distances with 1 source: {1: 0, 2: 15, 3: -8, 4: -12, 5: 5, 7: -4, 0:
19, 6: 11, 8: 4, 9: -9}
Distances with 2 source: {2: 0, 0: 4, 8: 0, 3: 1, 1: 16, 4: -1, 5: 14,
6: 20, 7: 12, 9: 2}
Distances with 3 source: {3: 0, 4: -2, 5: 13, 9: 1, 0: 34, 1: 15, 2:
30, 6: 19, 7: 11, 8: 12}
Distances with 4 source: {4: 0, 3: 4, 0: 38, 1: 19, 2: 34, 5: 17, 6:
23, 7: 15, 8: 16, 9: 5}
```

```
Distances with 5 source: {5: 0, 6: 6, 8: -1, 9: -7, 0: 21, 1: 2, 2: 17, 3: -6, 4: -10, 7: -2}
Distances with 6 source: {6: 0, 1: -4, 5: -2, 8: -3, 7: -8, 0: 15, 2: 11, 3: -12, 4: -16, 9: -13}
Distances with 7 source: {7: 0, 9: -5, 8: 8, 0: 30, 1: 11, 2: 26, 3: -4, 4: -8, 5: 9, 6: 15}
Distances with 8 source: {8: 0, 9: 9, 0: 44, 1: 25, 2: 40, 3: 10, 4: 6, 5: 23, 6: 29, 7: 21}
Distances with 9 source: {9: 0, 4: -3, 0: 35, 1: 16, 2: 31, 3: 1, 5: 14, 6: 20, 7: 12, 8: 13}
```

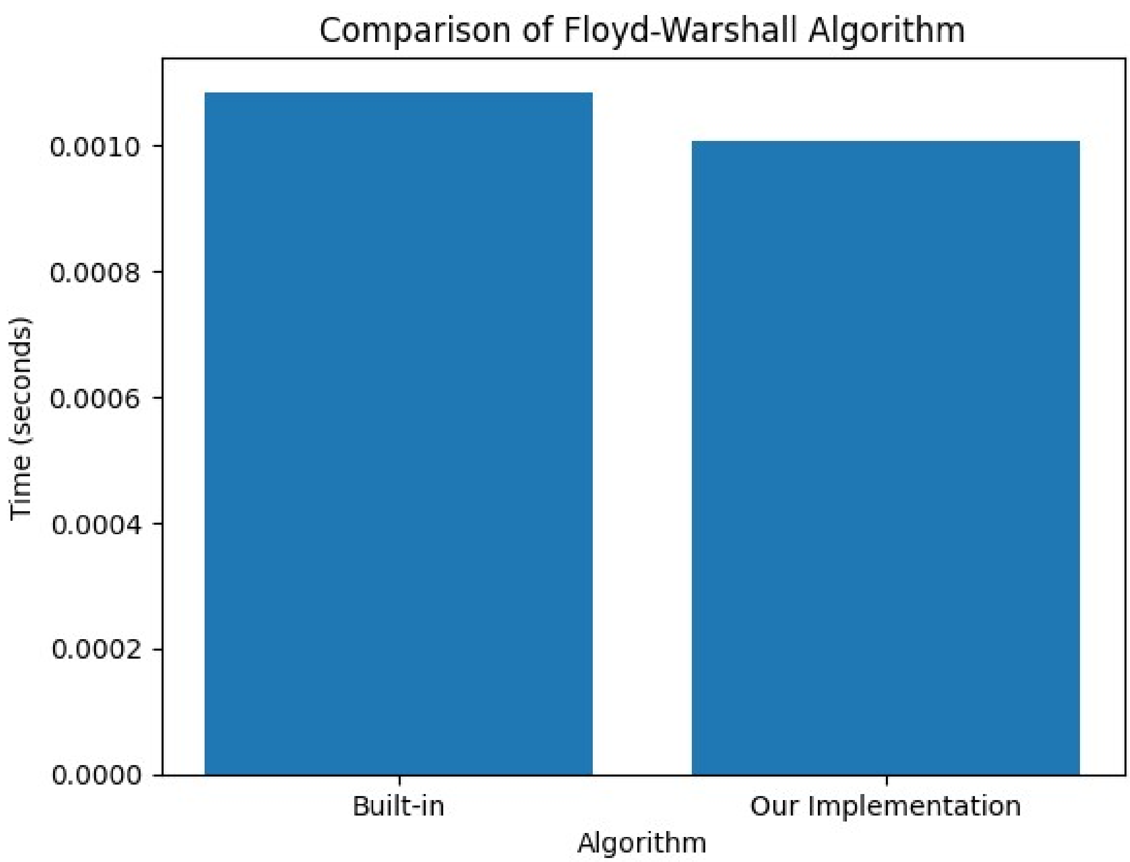
```
# custom floyd-warshall algorithm
dist_matrix = floyd_warshall(G)
for source in range(G.number_of_nodes()):
    distances = {i: dist_matrix[source][i] for i in range(G.number_of_nodes())}
    print(f'Distances with {source} source: {distances}')
```

```
Distances with 0 source: {0: 0, 1: 19, 2: 34, 3: 11, 4: 7, 5: 24, 6: 30, 7: 15, 8: 9, 9: 10}
Distances with 1 source: {0: 19, 1: 0, 2: 15, 3: -8, 4: -12, 5: 5, 6: 11, 7: -4, 8: 4, 9: -9}
Distances with 2 source: {0: 4, 1: 16, 2: 0, 3: 1, 4: -1, 5: 14, 6: 20, 7: 12, 8: 0, 9: 2}
Distances with 3 source: {0: 34, 1: 15, 2: 30, 3: 0, 4: -2, 5: 13, 6: 19, 7: 11, 8: 12, 9: 1}
Distances with 4 source: {0: 38, 1: 19, 2: 34, 3: 4, 4: 0, 5: 17, 6: 23, 7: 15, 8: 16, 9: 5}
Distances with 5 source: {0: 21, 1: 2, 2: 17, 3: -6, 4: -10, 5: 0, 6: 6, 7: -2, 8: -1, 9: -7}
Distances with 6 source: {0: 15, 1: -4, 2: 11, 3: -12, 4: -16, 5: -2, 6: 0, 7: -8, 8: -3, 9: -13}
Distances with 7 source: {0: 30, 1: 11, 2: 26, 3: -4, 4: -8, 5: 9, 6: 15, 7: 0, 8: 8, 9: -5}
Distances with 8 source: {0: 44, 1: 25, 2: 40, 3: 10, 4: 6, 5: 23, 6: 29, 7: 21, 8: 0, 9: 9}
Distances with 9 source: {0: 35, 1: 16, 2: 31, 3: 1, 4: -3, 5: 14, 6: 20, 7: 12, 8: 13, 9: 0}
```

```
start1 = time.time()
floyd_warshall_predecessor_and_distance(G)
end1 = time.time()

start2 = time.time()
floyd_warshall(G)
end2 = time.time()

time_taken1 = end1 - start1
time_taken2 = end2 - start2
```

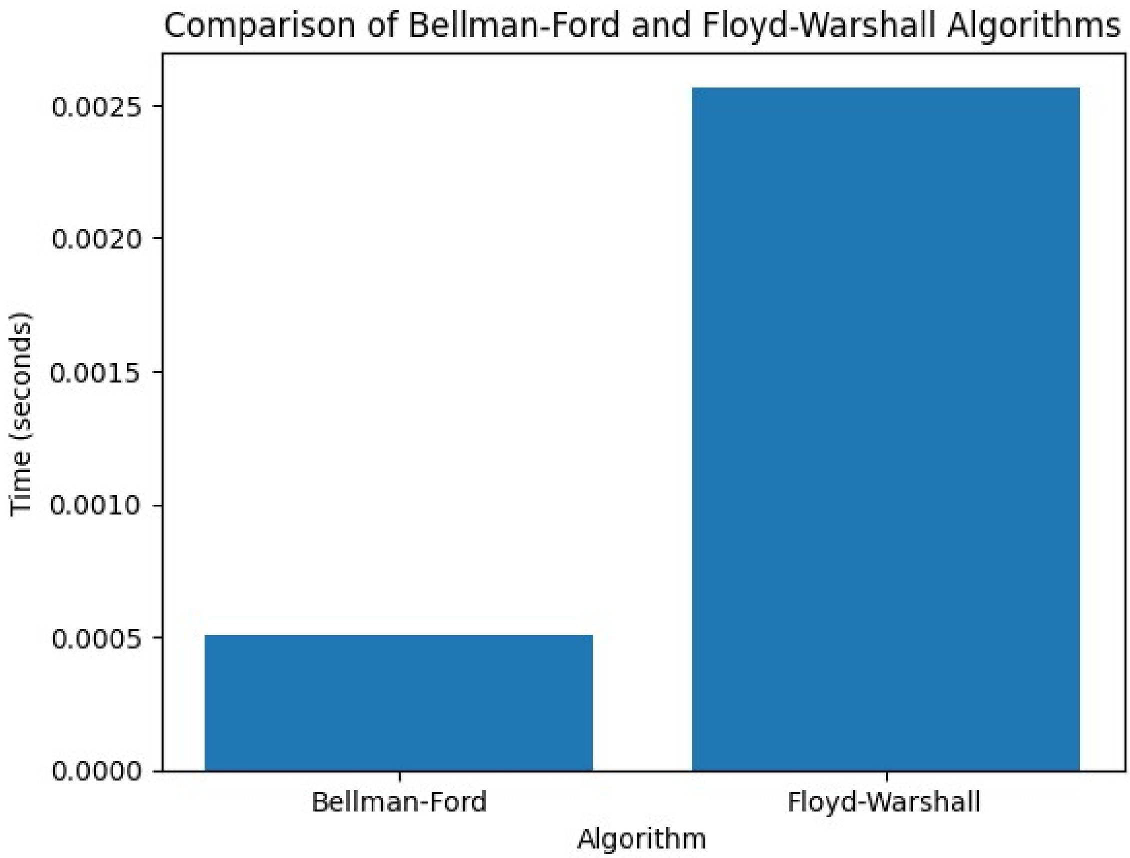
```
print(f'Built-in Floyd-Warshall took {time_taken1} seconds')
print(f'Our Floyd-Warshall took {time_taken2} seconds')
try:
    if time_taken1 < time_taken2:
        print(f'Our implementation is {time_taken2 / time_taken1}
times faster than the built-in one')
    else:
        print(f'Our implementation is {time_taken1 / time_taken2}
times slower than the built-in one')
except:
    print('Division by zero')

plt.bar(['Built-in', 'Our Implementation'], [time_taken1,
time_taken2])
plt.xlabel('Algorithm')
plt.ylabel('Time (seconds)')
plt.title('Comparison of Floyd-Warshall Algorithm')
plt.show()
```

```
Built-in Floyd-Warshall took 0.0010838508605957031 seconds
Our Floyd-Warshall took 0.001007080078125 seconds
Our implementation is 1.0762310606060606 times slower than the built-
in one
```



```python
start1 = time.time()
floyd_warshall(G)
end1 = time.time()

start2 = time.time()
for i in range(G.number_of_nodes()):
    bellman_ford(G, i)
end2 = time.time()

time_taken1 = end1 - start1
time_taken2 = end2 - start2

print(f'Bellman-Ford took {time_taken1} seconds')
print(f'Floyd-Warshall took {time_taken2} seconds')
try:
    if time_taken1 > time_taken2:
        print(f'Bellman-Ford is {time_taken1 / time_taken2} times
faster than Floyd-Warshall')
    else:
        print(f'Floyd-Warshall is {time_taken2 / time_taken1} times
faster than Bellman-Ford')
except:
    print('Division by zero')

plt.bar(['Bellman-Ford', 'Floyd-Warshall'], [time_taken1,
time_taken2])
plt.xlabel('Algorithm')
plt.ylabel('Time (seconds)')
plt.title('Comparison of Bellman-Ford and Floyd-Warshall Algorithms')
plt.show()
```

```
Bellman-Ford took 0.0005090236663818359 seconds
Floyd-Warshall took 0.002566099166870117 seconds
Floyd-Warshall is 5.041217798594848 times faster than Bellman-Ford
```



Bellman-Ford: Наша власна реалізація Bellman-Ford виявилася менш ефективною за швидкістю ніж вбудована. На великій кількості вершин вдудований варіант алгоритму переважно працює швидше у кілька разів.

Floyd-Warshall: Наша реалізація Floyd-Warshall частіше працює повільніше ніж вбудована імплементація, але в середньому різниця в часі складає до 25%.

Bellman-Ford vs Floyd-Warshall: Floyd-Warshall значно швидший за Bellman-Ford. У цьому конкретному випадку Floyd-Warshall працює швидше приблизно у 5 разів швидше за Bellman-Ford. Таким чином, якщо маємо справу зі значними обсягами даних або великими графами, використання Floyd-Warshall може бути більш доцільним з точки зору швидкодії.