min-path

February 14, 2024

[2]:
```
# !pip install networkx
# !pip install matplotlib
# !pip install tqdm
```

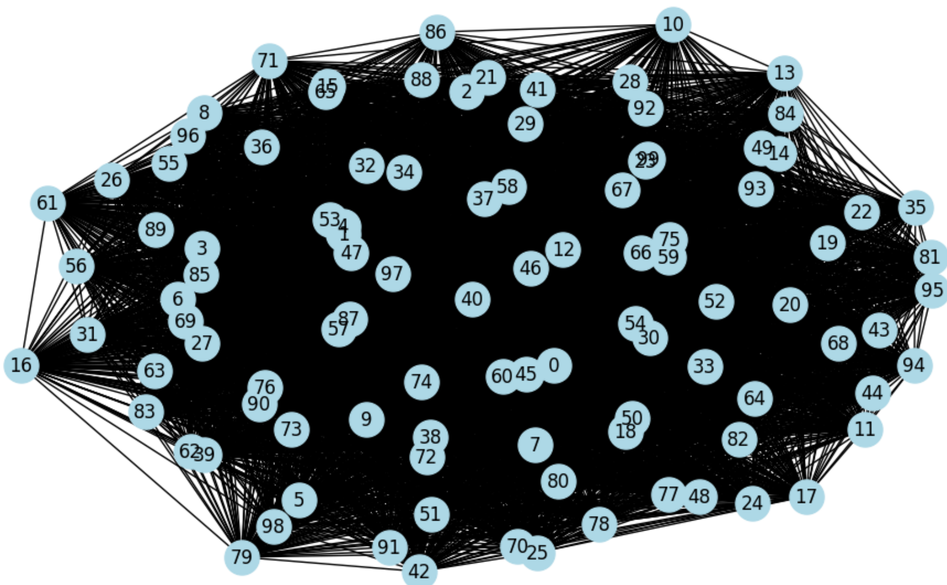
[3]:
```
import random
import networkx as nx
import matplotlib.pyplot as plt
from itertools import combinations, groupby
```

0.1 Generating graph

[4]:
```
# 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:
        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
```

[5]:
```
G = gnp_random_connected_graph(100, 1, False, True)
```



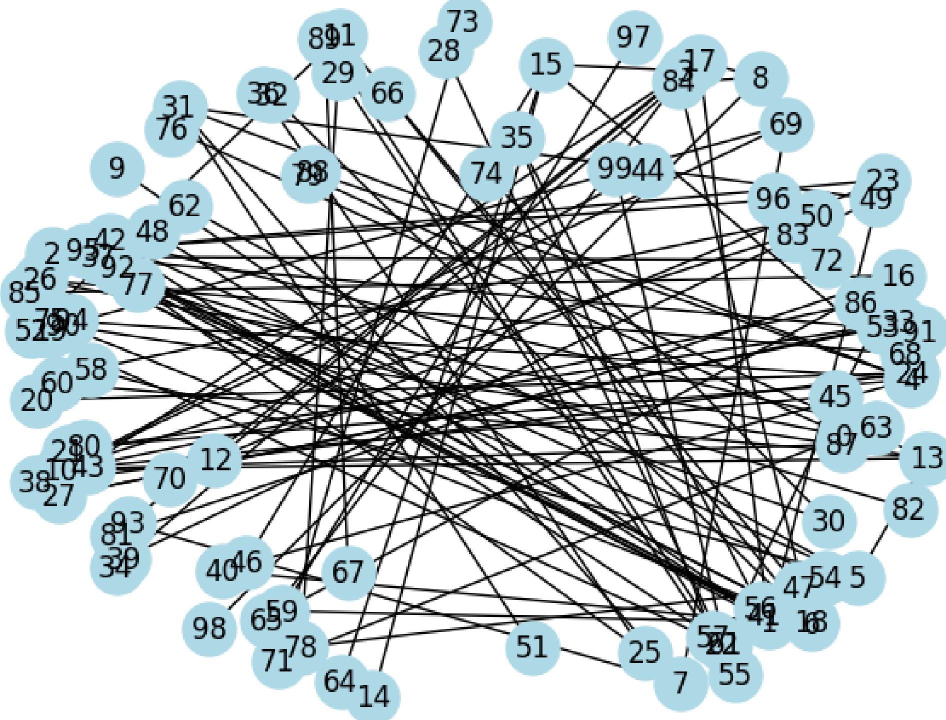
1 For Task 1

1.1 Kruskal's algorithm

[6]:
```
from networkx.algorithms import tree
```

[7]:
```
mstk = tree.minimum_spanning_tree(G, algorithm="kruskal")
```

[8]:
```
nx.draw(mstk, node_color='lightblue',
        with_labels=True,
        node_size=500)
```



[9]:
```python
mstk.edges(), len(mstk.edges())
```

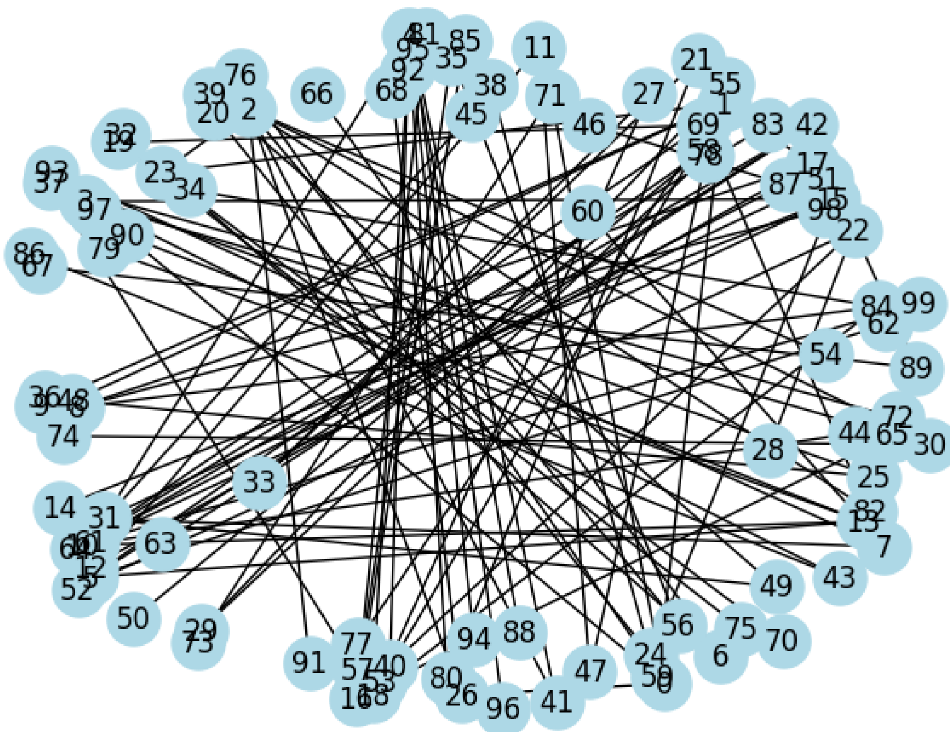
[9]:
```
(EdgeView([(0, 26), (0, 71), (0, 76), (1, 9), (1, 17), (1, 23), (1, 40), (1,
59), (1, 77), (1, 48), (2, 23), (2, 26), (2, 30), (2, 41), (2, 54), (2, 56), (2,
82), (2, 91), (3, 15), (3, 43), (3, 65), (3, 93), (4, 31), (4, 40), (4, 80), (4,
88), (4, 94), (5, 13), (5, 42), (5, 58), (5, 62), (6, 75), (6, 32), (6, 66), (7,
31), (7, 69), (7, 93), (8, 17), (8, 84), (8, 98), (10, 13), (10, 44), (10, 45),
(10, 69), (10, 83), (10, 87), (11, 47), (11, 52), (12, 17), (13, 60), (13, 79),
(13, 90), (14, 15), (15, 24), (15, 64), (15, 46), (16, 22), (16, 65), (16, 81),
(16, 85), (18, 37), (18, 84), (18, 92), (19, 69), (19, 75), (20, 24), (21, 33),
(21, 80), (22, 28), (22, 52), (22, 89), (23, 70), (24, 38), (25, 36), (25, 37),
(26, 68), (27, 33), (27, 53), (29, 55), (29, 78), (30, 74), (31, 49), (31, 51),
(34, 84), (35, 56), (35, 57), (37, 72), (39, 49), (41, 95), (42, 61), (42, 63),
(43, 86), (47, 78), (50, 58), (59, 97), (61, 99), (67, 89), (71, 73), (95,
96)]),
 99)
```

[10]:
```python
def kruskal(graph):
    if isinstance(graph, nx.DiGraph):
        G = nx.DiGraph()
```

```python
    else:
        G = nx.Graph()

    G.add_nodes_from(graph.nodes)

    edges = sorted(graph.edges(data=True), key=lambda x: x[2]['weight'])

    parent = {node: node for node in graph.nodes}

    def find(node):
        if parent[node] != node:
            parent[node] = find(parent[node])
        return parent[node]

    def union(u, v):
        root_u = find(u)
        root_v = find(v)
        parent[root_u] = root_v

    for u, v, w in edges:
        if find(u) != find(v):
            G.add_edge(u, v, weight=w['weight'])
            union(u, v)
    nx.draw(G, node_color='lightblue',
            with_labels=True,
            node_size=500)
    return G
```

[11]:
```python
kruskal(G)
```

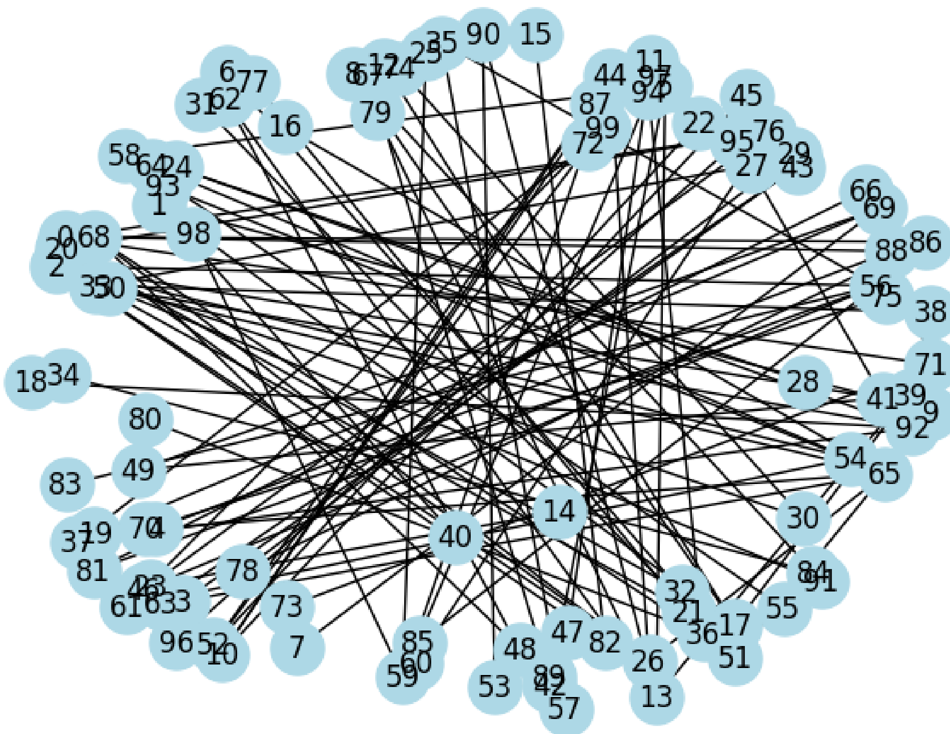
[11]: <networkx.classes.graph.Graph at 0x12151e3d0>



1.2 Prim's algorithm

[12]:
```
mstp = tree.minimum_spanning_tree(G, algorithm="prim")
```

[13]:
```
nx.draw(mstp, node_color='lightblue',
        with_labels=True,
        node_size=500)
```




```
[14]: mstp.edges(), len(mstp.edges())
```

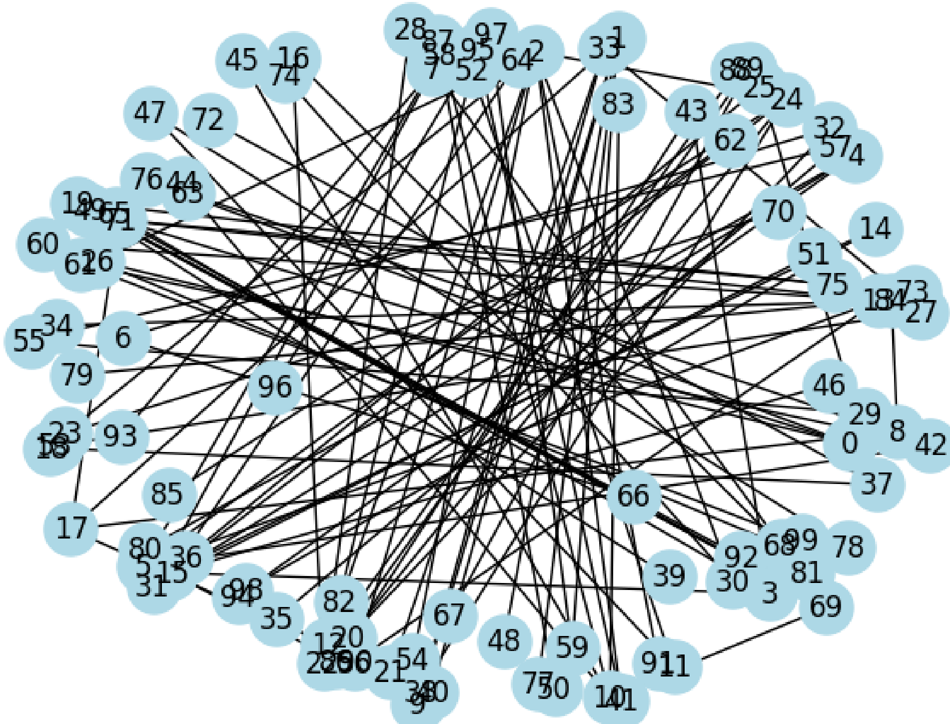
```
[14]: (EdgeView([(0, 26), (0, 71), (0, 76), (1, 17), (1, 9), (1, 40), (2, 26), (2,
30), (2, 41), (2, 54), (2, 56), (2, 82), (2, 91), (3, 65), (3, 43), (4, 88), (5,
13), (5, 42), (5, 58), (6, 32), (7, 69), (8, 17), (8, 84), (10, 44), (10, 45),
(10, 87), (11, 52), (11, 47), (12, 17), (13, 71), (13, 62), (13, 79), (13, 90),
(14, 23), (15, 82), (16, 65), (17, 71), (18, 92), (19, 69), (20, 76), (20, 32),
(21, 33), (22, 52), (23, 76), (24, 92), (24, 28), (25, 59), (26, 44), (26, 65),
(26, 68), (27, 33), (29, 78), (30, 74), (31, 51), (33, 65), (33, 38), (34, 84),
(35, 56), (35, 57), (36, 68), (36, 77), (36, 80), (36, 94), (37, 92), (37, 72),
(39, 49), (41, 95), (43, 63), (44, 52), (44, 59), (46, 56), (48, 62), (49, 69),
(50, 92), (51, 65), (53, 90), (54, 61), (54, 64), (55, 58), (56, 81), (56, 83),
(56, 85), (59, 97), (59, 98), (60, 76), (61, 99), (65, 70), (65, 93), (66, 81),
(67, 89), (68, 86), (68, 88), (68, 89), (69, 71), (70, 75), (71, 73), (71, 92),
(78, 86), (95, 96)]),
 99)
```

```
[15]: def prim(graph):
          U = set()
          V_U = set(graph.nodes)
```

```
    start_vertex = next(iter(V_U))
    U.add(start_vertex)
    V_U.remove(start_vertex)

    if isinstance(graph, nx.DiGraph):
        G = nx.DiGraph()
    else:
        G = nx.Graph()
    while V_U:
        min_weight = float('inf')
        min_edge = None
        for u in U:
            for v in V_U:
                if graph.has_edge(u, v):
                    weight = graph[u][v]['weight']
                    if weight < min_weight:
                        min_weight = weight
                        min_edge = (u, v)

        G.add_edge(*min_edge, weight=min_weight)

        U.add(min_edge[1])
        V_U.remove(min_edge[1])
    nx.draw(G, node_color='lightblue',
                with_labels=True,
                node_size=500)
    return G
```

[16]:
```
prim(G)
```

[16]: <networkx.classes.graph.Graph at 0x126b60e10>



1.3 Example on time measuring

Read more on this: https://realpython.com/python-timer/

Recall that you should measure times for 5, 10, 20, 50, 100, 200, 500 nodes 1000 times (and take mean of time taken for each node amount).

Then you should build the plot for two algorithms (x - data size, y - mean time of execution).

[17]:
```
import time
from tqdm import tqdm
```

[18]:
```
NUM_OF_ITERATIONS=100
def count_time(function, size):
    time_taken = 0
    for _ in tqdm(range(NUM_OF_ITERATIONS)):

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

        start = time.time()
        function(G)
```

```
        end = time.time()

        time_taken += end - start

    return time_taken / NUM_OF_ITERATIONS
```

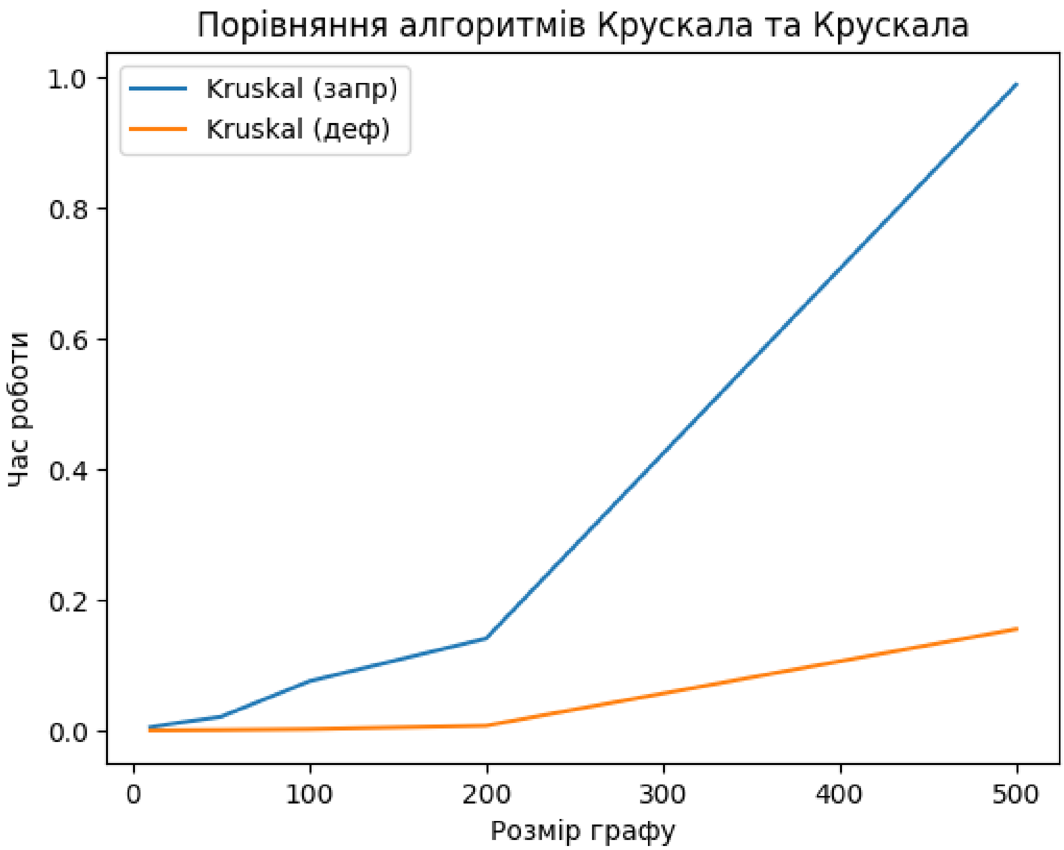
Kruscal's algorithm

1)

, . find()
, . Union .
: , ,
. . , find
union, . 100 , 3
.

[20]:
```
def kruskal_def(G):
    return tree.minimum_spanning_tree(G, algorithm="kruskal")

graph_sizes = [10, 20, 50, 100, 200, 500]
kruskal_mine = [count_time(kruskal, size) for size in graph_sizes]  #
↪                    (         )
kruskal_defined = [count_time(kruskal_def, size) for size in graph_sizes]  #
↪                            (     )
plt.clf()
plt.plot(graph_sizes, kruskal_mine, label='Kruskal (   )')
plt.plot(graph_sizes, kruskal_defined, label='Kruskal (  )')
plt.xlabel('       ')
plt.ylabel('      ')
plt.title('                         ')
plt.legend()
plt.show()
```

```
  0%|          | 0/100 [00:00<?, ?it/s]100%|      | 100/100 [00:00<00:00,
211.66it/s]
100%|      | 100/100 [00:00<00:00, 107.34it/s]
100%|      | 100/100 [00:02<00:00, 46.76it/s]
100%|      | 100/100 [00:07<00:00, 12.93it/s]
100%|      | 100/100 [00:18<00:00,  5.38it/s]
100%|      | 100/100 [01:43<00:00,  1.04s/it]
100%|      | 100/100 [00:00<00:00, 12655.17it/s]
100%|      | 100/100 [00:00<00:00, 4502.69it/s]
100%|      | 100/100 [00:00<00:00, 921.07it/s]
100%|      | 100/100 [00:00<00:00, 258.62it/s]
100%|      | 100/100 [00:01<00:00, 66.76it/s]
100%|      | 100/100 [00:20<00:00,  4.90it/s]
```



, ,

Prim's algorithm

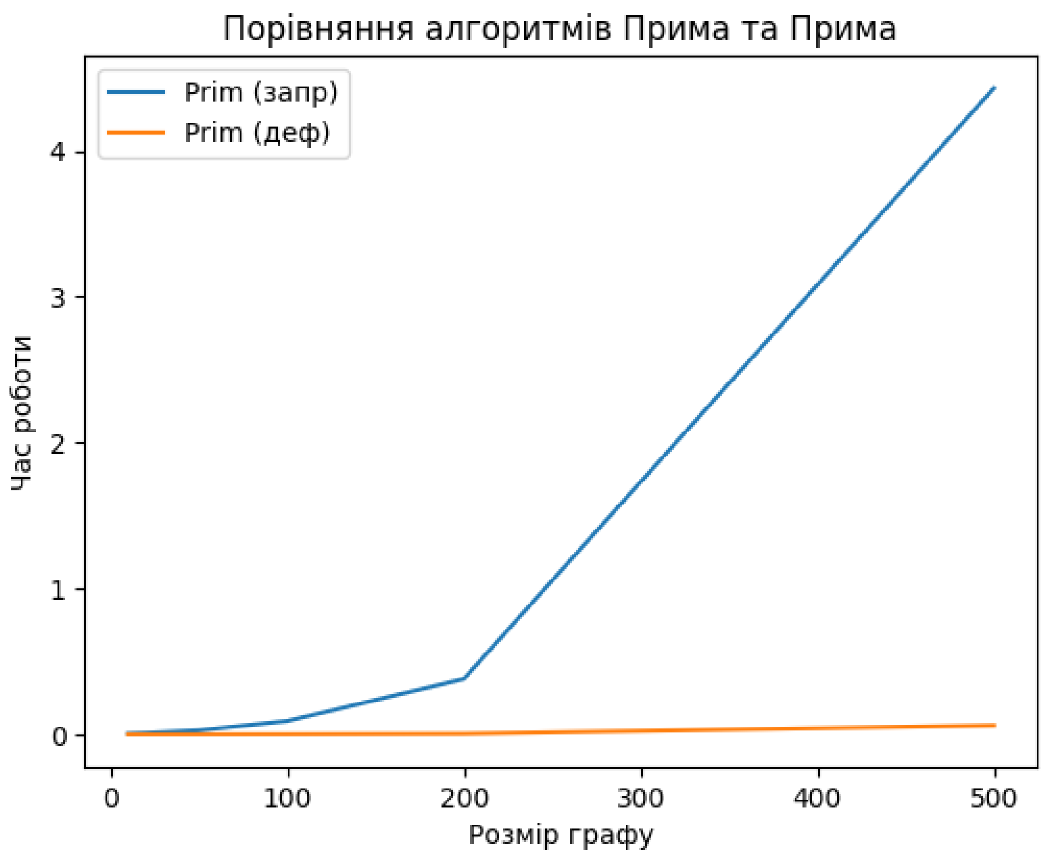
2)

2 , - , - .
, , . ,
, , .

```
[ ]: def prim_def(G):
         return tree.minimum_spanning_tree(G, algorithm="prim")
     graph_sizes = [10, 20, 50, 100, 200, 500]
     prim_mine = [count_time(prim, size) for size in graph_sizes]   #
      ↪          (          )
     prim_defult = [count_time(prim_def, size) for size in graph_sizes]   #
      ↪                      (      )
     plt.clf()
     plt.plot(graph_sizes, prim_mine, label='Prim (   )')
     plt.plot(graph_sizes, prim_defult, label='Prim (  )')
     plt.xlabel('        ')
```

```
plt.ylabel('        ')
plt.title('                        ')
plt.legend()
plt.show()
```

```
100%|          | 100/100 [00:00<00:00, 213.51it/s]
100%|          | 100/100 [00:00<00:00, 110.12it/s]
100%|          | 100/100 [00:02<00:00, 36.15it/s]
100%|          | 100/100 [00:09<00:00, 10.75it/s]
100%|          | 100/100 [00:41<00:00,  2.43it/s]
100%|          | 100/100 [07:28<00:00,  4.48s/it]
100%|          | 100/100 [00:00<00:00, 14790.55it/s]
100%|          | 100/100 [00:00<00:00, 5366.85it/s]
100%|          | 100/100 [00:00<00:00, 1110.12it/s]
100%|          | 100/100 [00:00<00:00, 325.13it/s]
100%|          | 100/100 [00:01<00:00, 81.07it/s]
100%|          | 100/100 [00:12<00:00,  8.14it/s]
```



, 40 , ,