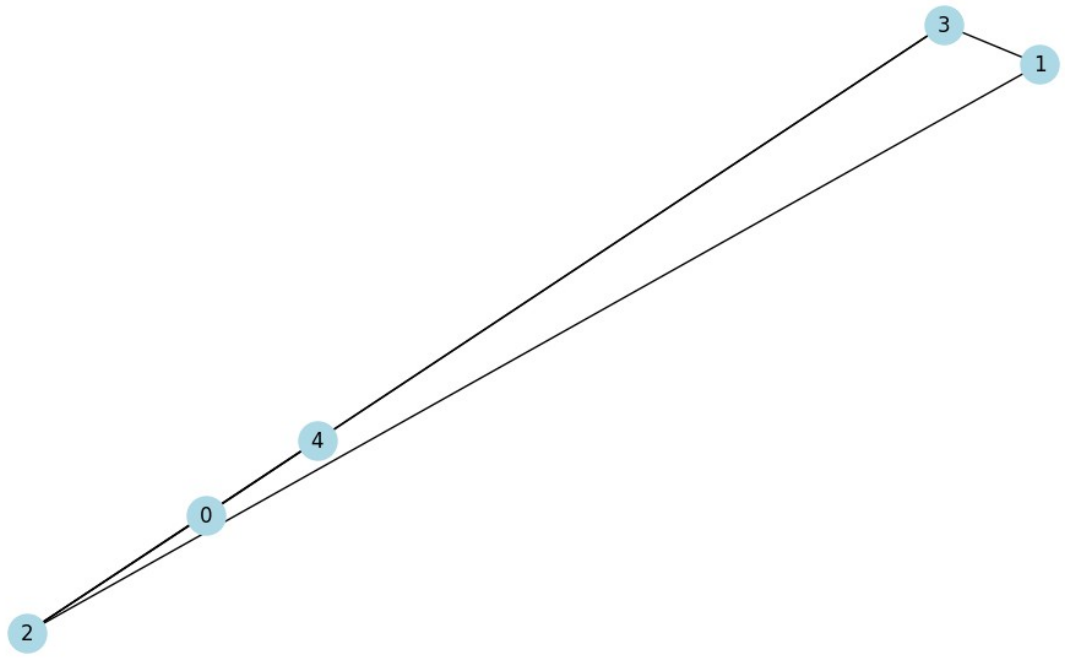
```
# !pip install networkx
# !pip install matplotlib
# !pip install tqdm

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

## Generating graph

```
# 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:
```

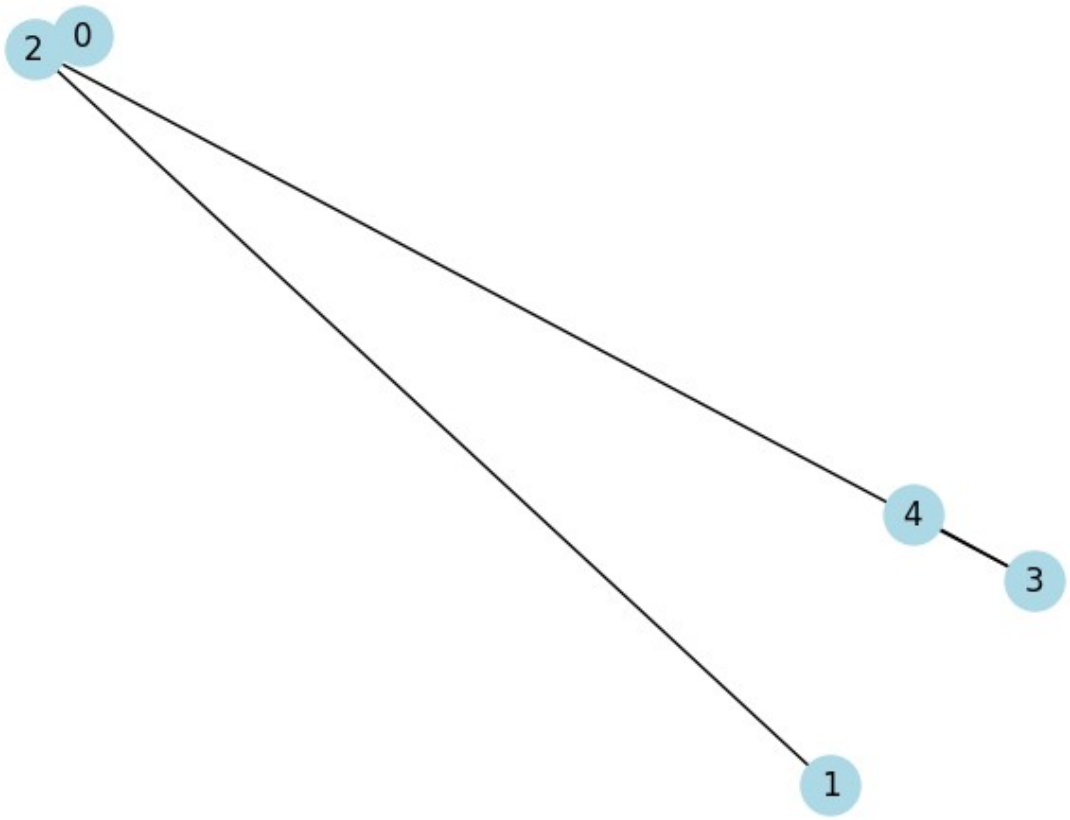
```
        plt.figure(figsize=(10,6))
        if directed:
            # draw with edge weights
            pos = nx.arf_layout(G)
            nx.draw(G,pos, node_color='lightblue',
                    with_labels=True,
                    node_size=500,
                    arrowsize=20,
                    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

G = gnp_random_connected_graph(5, 0.5, False, True)
```



# For Task 1

## Kruskal's algorithm

```
from networkx.algorithms import tree

mstk = tree.minimum_spanning_tree(G, algorithm="kruskal")

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




```
mstk.edges(), len(mstk.edges())
```

```
(EdgeView([(0, 2), (1, 2), (2, 3), (3, 4)]), 4)
```

```
def weight(nodes, list_nodes) -> int:
    """
    Function to count weight of generated min frames

    """
```

```
    min_weight = 0

    nodes = dict(nodes)
    for tup in list_nodes:
        min_weight+=nodes[tup[0]][tup[1]]['weight']

    return min_weight
```

Додатково була реалізована функція для знаходження ваги мінімального каркасу вбудованих алгоритмів, щоб їх вагу можна було порівнювати з вагою мінімальних каркасів реалізованих нами. Адже часто трапляються ребра з однаковими вагами і тоді список ребер мінімального каркасу згенерованого вбудованим алгоритмом може відрізнятися від результату наших алгоритмів.

```
def kruskal_algorithm(tree: object) -> list[tuple[int]]:

    """
    Kruskal's algorithm
    """

    tree_dict = dict(tree.adjacency())
    edges = []

    for node in tree_dict:
        for neighbor, data in tree_dict[node].items():
            edges.append((node, neighbor, data['weight']))

    edges.sort(key=lambda x: x[2])

    total_weight = 0
    min_frame = []
    disjoint_sets = [{node} for node in tree_dict]

    def find_set(node):
        for subset in disjoint_sets:
            if node in subset:
                return subset

    for u, v, weight in edges:
        set_u = find_set(u)
        set_v = find_set(v)

        if set_u != set_v:
            min_frame.append((u, v))
            total_weight += weight
            disjoint_sets.remove(set_u)
            disjoint_sets.remove(set_v)
            disjoint_sets.append(set_u.union(set_v))

    return total_weight, min_frame
```

```
kruskal_algorithm(G) #our kruskal's algorithm implementation weight and path
```

```
(8, [(2, 3), (1, 2), (3, 4), (0, 2)])
```

```
(weight(mstk.adjacency(), mstk.edges()), mstk.edges())  #built-in kruskal algorithm weight and path
```

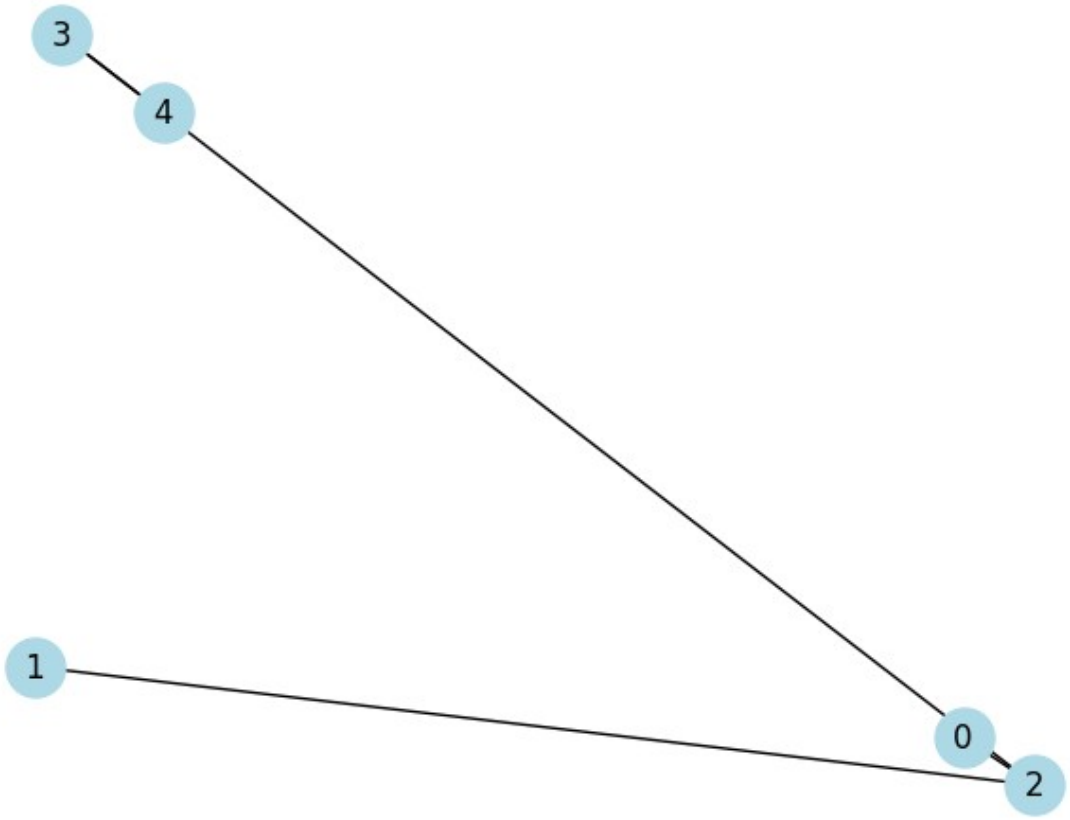
```
(8, EdgeView([(0, 2), (1, 2), (2, 3), (3, 4)]))
```

Цей код реалізує алгоритм Крускала для пошуку мінімального каркасу у згенерованому зваженому графі: Функція приймає зважений граф у вигляді об'єкта tree (G), що є згенерованим раніше та повертає вагу мінімального каркасу у вигляді числа. Граф представляється у вигляді словника, де ключі - це вершини графа, а значення - це словники з сусідами та їх атрибутами. Далі створюється порожній список для збереження всіх ребер графа. Потім відбувається проходження циклом по вершинах графа і цикл по сусідах поточної вершини з її атрибутами. Далі у список ребер додається кожне ребро (вершина, сусід, вага). Далі ці ребра сортуються за їх вагою у зростаючому порядку. Створюється порожній список для збереження ребер мінімального каркасу. Далі створюється список неперетинних множин для кожної вершини графа. Далі відбувається цикл по відсортованих ребрах та розпаковка даних з ребра. Знаходження множини, до якої належить вершина u та v, та функція, що перевіряє чи належать вузли різним множинам, щоб не створювати простих циклів. Коли ребро пройшло усі ці невеликі перевірки воно додається до мінімального каркасу. А його вагу додаємо до загальної ваги мінімального каркасу. Далі видаляються старі множини та додається нова множини, що об'єднує попередні дві. Далі відбувається відсортування ребер мінімального каркасу та зміна порядку вершин у кожному ребрі. Функція повертає загальну вагу мінімального каркасу та шлях.

# Prim's algorithm

```
mstp = tree.minimum_spanning_tree(G, algorithm="prim")
```

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




```
mstp.edges(), len(mstp.edges())
```

```
(EdgeView([(0, 2), (1, 2), (2, 3), (3, 4)]), 4)
```

```
import heapq

def prim_algorithm(tree: object) -> tuple[list[tuple[int]], int]:
    """
    Prim's algorithm
    """
    tree_dict = dict(tree.adjacency())
    num_nodes = len(tree_dict)
    visited = set()
    visited.add(0)
    min_edges = []
    min_weight = 0
    min_heap = []

    def add_neighbors(node):
        nonlocal min_heap
        for neighbor, data in tree_dict[node].items():
            if neighbor not in visited:
```

```
                heapq.heappush(min_heap, (data['weight'], node,
neighbor))

    add_neighbors(0)
    while len(visited) < num_nodes:
        weight, u, v = heapq.heappop(min_heap)
        if v not in visited:
            min_edges.append((u, v))
            min_weight += weight
            visited.add(v)
            add_neighbors(v)

    return min_weight, min_edges

prim_algorithm(G) #our prim's algorithm implementation weight and path
```

```
(8, [(0, 2), (2, 3), (2, 1), (3, 4)])
```

```
(weight(mstp.adjacency(), mstp.edges()), mstp.edges()) #built-in
prim's algorithm weight and path
```

```
(8, EdgeView([(0, 2), (1, 2), (2, 3), (3, 4)]))
```

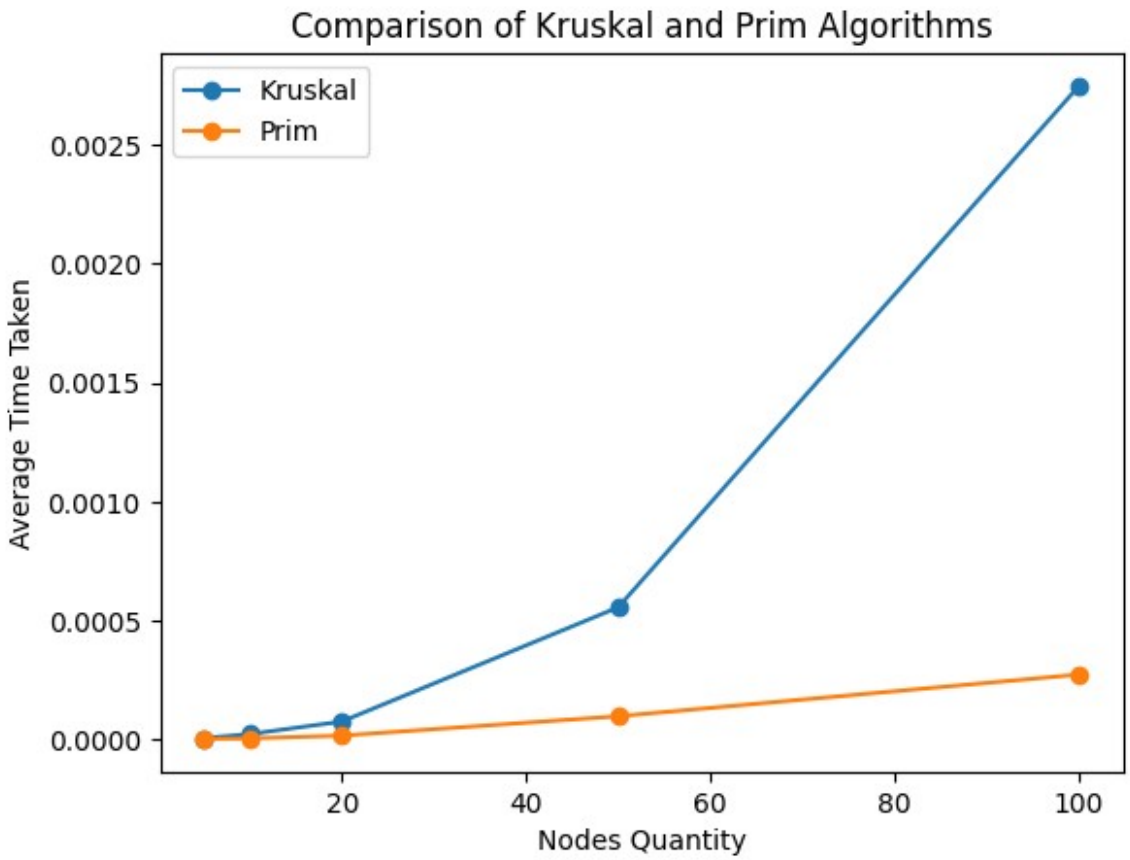
Цей код реалізує алгоритм Пріма для знаходження мінімального каркасу в зваженому зв'язаному графі: Створюється словник tree_dict, який містить представлення графа у вигляді словника суміжності. Створюється порожній набір visited та додається початкова вершина 0, з якої ми починаємо створювати шлях. Поки всі вершини не будуть відвідані, визначається ребро з найменшою вагою, що з'єднує вже відвідану частину графа з його невідвіданою частиною. Це ребро видаляється з пріоритетної черги min_heap, а його вага додається до ваги мінімального каркасу. Вершина, яку ребро з'єднує з вже відвіданими, додається до visited, до множини вершин, що вже були відвіданим. Потім до цієї нової вершини в min_heap додаються всі невідвідані сусіди. Як результат, повертається загальна вага мінімального каркаса та шлях. У цьому алгоритмі ми використали бібліотеку heapq – приоритетна черга. Основна перевага використання цієї черги у порівнянні з іншими структурами даних, такими як звичайні списки, полягає в тому, що вона дозволяє додавати та вилучати елементи з мінімальною (або максимальною) вагою швидко. Це важливо для алгоритму Пріма, оскільки на кожному кроці нам потрібно вибрати ребро з мінімальною вагою, і приоритетна черга допомагає зробити це ефективно.

Щоб перевірити чи добре працюють наші алгоритми, ми під кожним з них виводимо результати реалізованих нами та вбудованих методів.

```
import time
from tqdm import tqdm

NUM_OF_ITERATIONS = 1000
result1 = {}
time_taken = 0
nodes = [5, 10, 20, 50, 100]
```

```
for node in nodes:
    for i in tqdm(range(NUM_OF_ITERATIONS)):
        time_taken = 0
        G = gnp_random_connected_graph(node, 0.4, False)
        start = time.time()
        kruskal_algorithm(G)
        end = time.time()
        time_taken += end - start
        result1[node] = time_taken

time_taken = 0
result2 = {}

for node in nodes:
    for i in tqdm(range(NUM_OF_ITERATIONS)):
        time_taken = 0
        G = gnp_random_connected_graph(node, 0.3, False)
        start = time.time()
        prim_algorithm(G)
        end = time.time()
        time_taken += end - start
        result2[node] = time_taken

plt.plot(nodes, list(result1.values()), label='Kruskal', marker='o')
plt.plot(nodes, list(result2.values()), label='Prim', marker='o')
plt.xlabel('Nodes Quantity')
plt.ylabel('Average Time Taken')
plt.title('Comparison of Kruskal and Prim Algorithms')
plt.legend()
plt.show()
```

```
100%|██████████| 1000/1000 [00:00<00:00, 48952.56it/s]
100%|██████████| 1000/1000 [00:00<00:00, 15944.65it/s]
100%|██████████| 1000/1000 [00:00<00:00, 5442.32it/s]
100%|██████████| 1000/1000 [00:01<00:00, 969.64it/s]
100%|██████████| 1000/1000 [00:04<00:00, 207.91it/s]
100%|██████████| 1000/1000 [00:00<00:00, 75370.70it/s]
100%|██████████| 1000/1000 [00:00<00:00, 30206.14it/s]
100%|██████████| 1000/1000 [00:00<00:00, 10290.80it/s]
100%|██████████| 1000/1000 [00:00<00:00, 2106.28it/s]
100%|██████████| 1000/1000 [00:01<00:00, 543.89it/s]
```



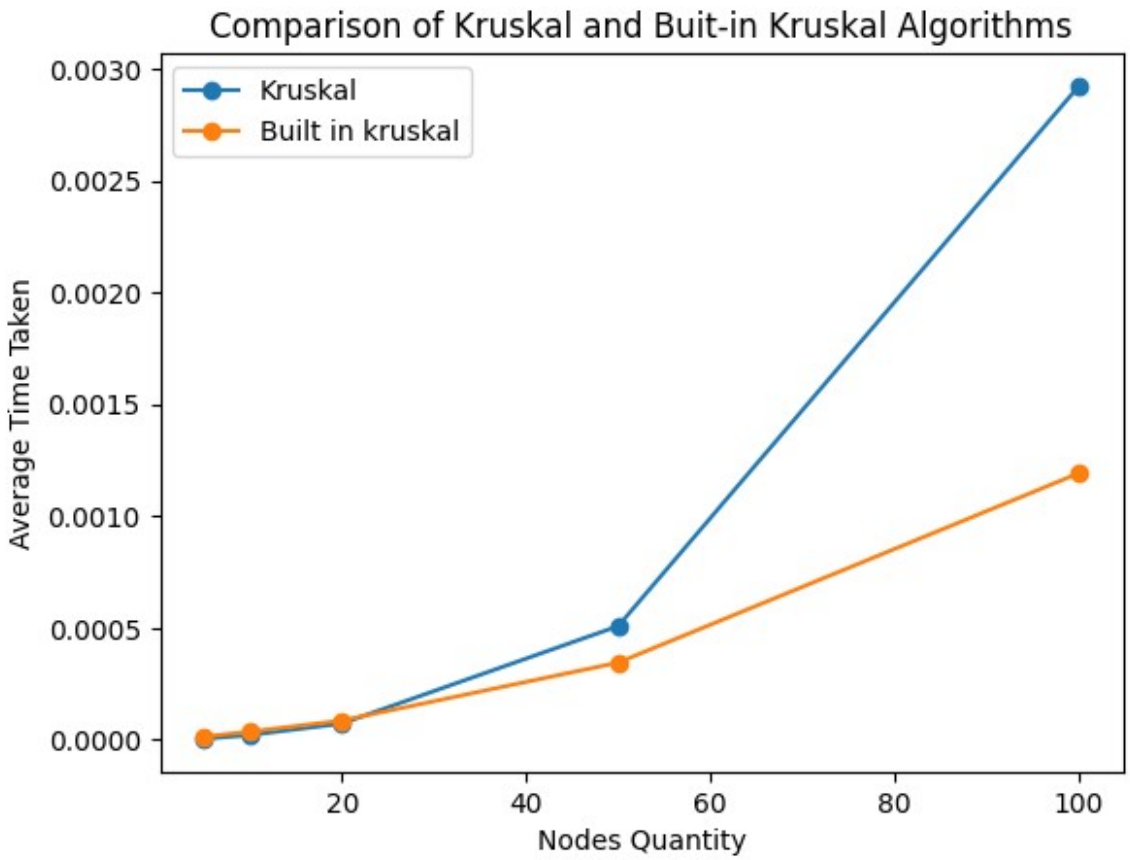

Цей код вимірює час за який алгоритм Прима та Крускала знаходить мінімальний каркас для графів з різною кількістю вершин, що вказані у списку. А також він виводить графік порівняння швидкості виконання коду алгоритмів.

На поданому графіку видно, що Алгоритм Крускала та Прима для графів з кількістю вершин >= 20, працює майже з однаковою ефективністю, але для більших значень алгоритм Прима є більш ефективним. На нашу думку це тому, що алгоритм Прима проходиться саме по найменших інцидентних ребрах до вершин, що вже були відвідані та використовує приоритетну чергу, яку ми застосували в реалізації нашого алгоритму, в той час як Крускала бере усі найменші вершини пари яких ще немає в множині вершин з усіх частин графа.

```
NUM_OF_ITERATIONS = 1000
result1 = {}
time_taken = 0
nodes = [5, 10, 20, 50, 100]

for node in nodes:
    for i in tqdm(range(NUM_OF_ITERATIONS)):
        time_taken = 0
        G = gnp_random_connected_graph(node, 0.4, False)
        start = time.time()
        kruskal_algorithm(G)
```

```
        end = time.time()
        time_taken += end - start
        result1[node] = time_taken

time_taken = 0
result2 = {}

for node in nodes:
    for i in tqdm(range(NUM_OF_ITERATIONS)):
        time_taken = 0
        G = gnp_random_connected_graph(node, 0.3, False)
        start = time.time()
        tree.minimum_spanning_tree(G, algorithm="kruskal")
        end = time.time()
        time_taken += end - start
        result2[node] = time_taken

plt.plot(nodes, list(result1.values()), label='Kruskal', marker='o')
plt.plot(nodes, list(result2.values()), label='Built in kruskal',
marker='o')
plt.xlabel('Nodes Quantity')
plt.ylabel('Average Time Taken')
plt.title('Comparison of Kruskal and Buit-in Kruskal Algorithms')
plt.legend()
plt.show()
```

```
100%|██████████| 1000/1000 [00:00<00:00, 51067.22it/s]
100%|██████████| 1000/1000 [00:00<00:00, 17335.20it/s]
100%|██████████| 1000/1000 [00:00<00:00, 5631.48it/s]
100%|██████████| 1000/1000 [00:01<00:00, 984.40it/s]
100%|██████████| 1000/1000 [00:04<00:00, 206.09it/s]
100%|██████████| 1000/1000 [00:00<00:00, 37071.15it/s]
100%|██████████| 1000/1000 [00:00<00:00, 16071.42it/s]
100%|██████████| 1000/1000 [00:00<00:00, 5983.94it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1339.09it/s]
100%|██████████| 1000/1000 [00:02<00:00, 368.04it/s]
```

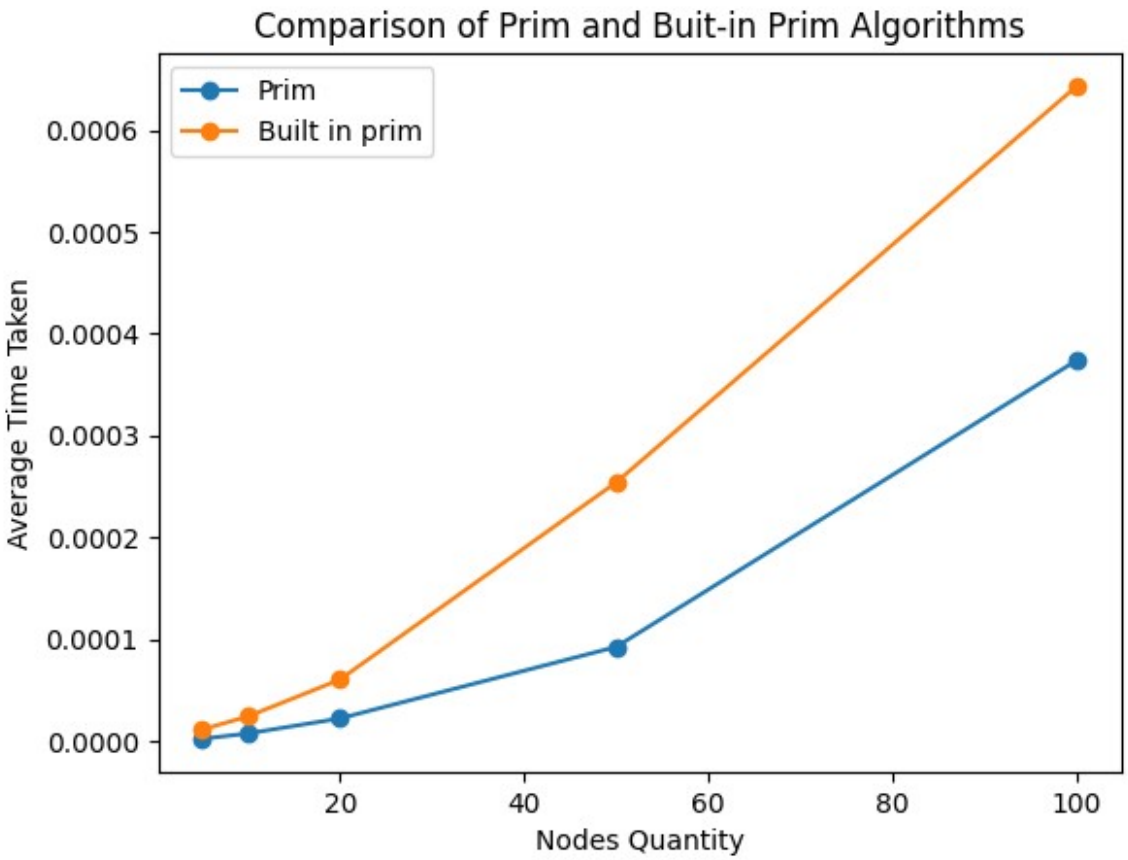



Тут зображено графік порівняння реалізованого нами та вбудованого алгоритму Крускала. На цьому графіку видно, що для кількості вершин >= 20 наш алгоритм та вбудований працюють однаково швидко, але для більших кількостей реалізований нами алгоритм Крускала працює повільніше ніж вбудований. На нашу думку така різниця тому, що наша реалізація алгоритму може обирати ребра неоптимальним чином, хоч і правильно.

```
NUM_OF_ITERATIONS = 1000
result1 = {}
time_taken = 0
nodes = [5, 10, 20, 50, 100]

for node in nodes:
    for i in tqdm(range(NUM_OF_ITERATIONS)):
        time_taken = 0
        G = gnp_random_connected_graph(node, 0.4, False)
        start = time.time()
        prim_algorithm(G)
        end = time.time()
        time_taken += end - start
        result1[node] = time_taken

time_taken = 0
result2 = {}
```

```python
for node in nodes:
    for i in tqdm(range(NUM_OF_ITERATIONS)):
        time_taken = 0
        G = gnp_random_connected_graph(node, 0.3, False)
        start = time.time()
        tree.minimum_spanning_tree(G, algorithm="prim")
        end = time.time()
        time_taken += end - start
        result2[node] = time_taken

plt.plot(nodes, list(result1.values()), label='Prim', marker='o')
plt.plot(nodes, list(result2.values()), label='Built in prim',
marker='o')
plt.xlabel('Nodes Quantity')
plt.ylabel('Average Time Taken')
plt.title('Comparison of Prim and Buit-in Prim Algorithms')
plt.legend()
plt.show()
```

```
100%|██████████| 1000/1000 [00:00<00:00, 61061.35it/s]
100%|██████████| 1000/1000 [00:00<00:00, 25912.84it/s]
100%|██████████| 1000/1000 [00:00<00:00, 8395.68it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1751.06it/s]
100%|██████████| 1000/1000 [00:02<00:00, 448.93it/s]
100%|██████████| 1000/1000 [00:00<00:00, 43241.14it/s]
100%|██████████| 1000/1000 [00:00<00:00, 19418.25it/s]
100%|██████████| 1000/1000 [00:00<00:00, 4698.56it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1626.56it/s]
100%|██████████| 1000/1000 [00:02<00:00, 433.65it/s]
```




Тут зображено графік порівняння реалізованого нами та вбудованого алгоритму Прима. Дуже добре помітно, що вбудована реалізація працює трохи гірше та повільніше ніж наша реалізація, це тому, що ми використовуємо модуль heapq – пріоритетну чергу, що дозволяє додавати та вилучати елементи з мінімальною (або максимальною) вагою з графа досить швидко, що робить наш алгоритм швидшим.