Матвій Шумилович і Богдан Мілян

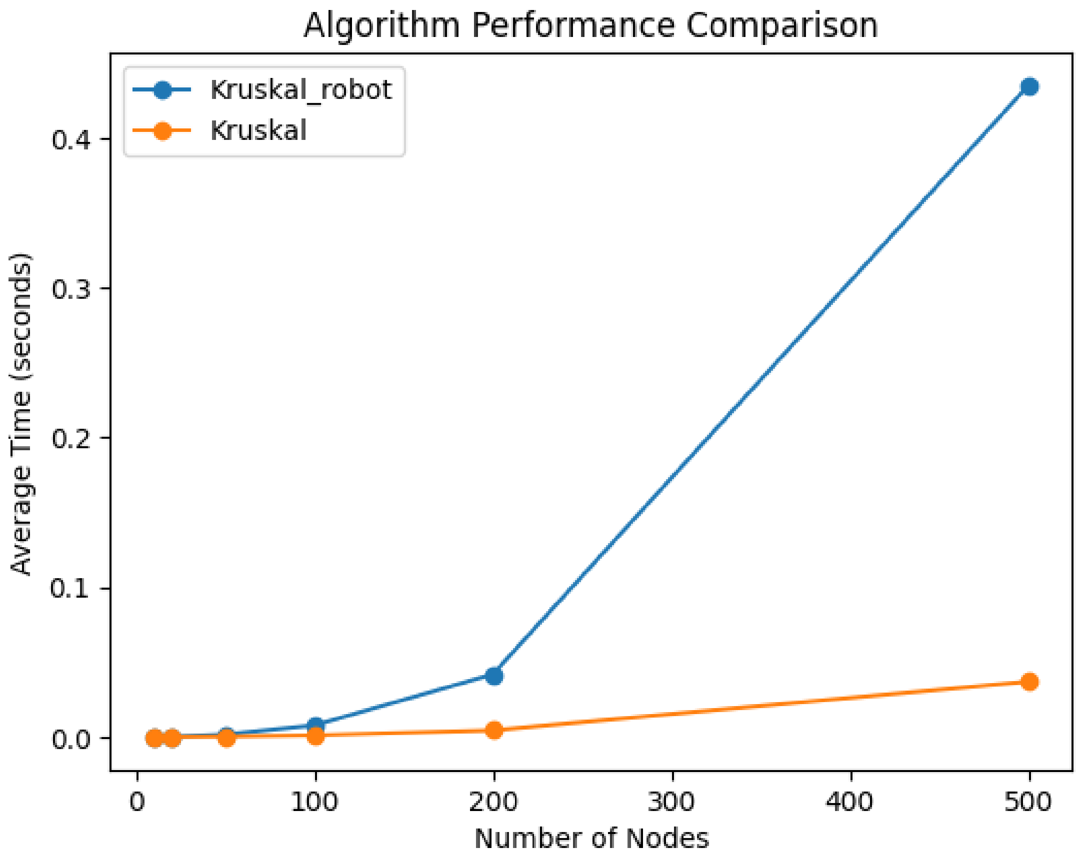
```
import random
import time
from itertools import combinations, groupby
import networkx as nx
import matplotlib.pyplot as plt
from tqdm import tqdm
from networkx.algorithms import tree

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:
            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

"""Kruskal Algoritm"""
class Kruskal:
    """Class to make algoritm"""
    def __init__(self, graph) -> None:
        """Kruskall init"""
        self.graph = graph
        self.vertices = {i: i for i in range(len(graph.nodes))}
        self.edges = list(graph.edges)
        self.edges.sort(key=lambda edge: graph[edge[0]][edge[1]]
['weight'])
        self.minimum_spanning_tree = []
    def find(self, vertex):
        """finding roots for vertexes"""
        if self.vertices[vertex] != vertex:
            self.vertices[vertex] = self.find(self.vertices[vertex])
        return self.vertices[vertex]
    def union(self, u, v):
        """uniting roots in trees, уникаємо циклів"""
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            self.vertices[root_u] = root_v
    def kruskal_algorithm(self):
        """Запускає алгоритм"""
        for edge in self.edges:
            u, v = edge
            if self.find(u) != self.find(v):
                self.minimum_spanning_tree.append(edge)
                self.union(u, v)
        return self.minimum_spanning_tree

"""Prims algorithm"""
def prim(graph: nx.classes.graph.Graph) -> nx.classes.graph.Graph:
    '''
    Prim's algorithm
    Uses and returns an object of class nx.classes.graph.Graph
    '''
    length = len(graph.nodes())
    output_graph = nx.Graph()
    edges = sorted(graph.edges(data=True), key= lambda x: x[2]
['weight'])
    output_graph.add_edge(edges[0][0], edges[0][1], weight = edges[0]
[2]['weight'])
    edges.pop(0)
    visited_nodes=output_graph.nodes()
    while len(visited_nodes) < length:
```

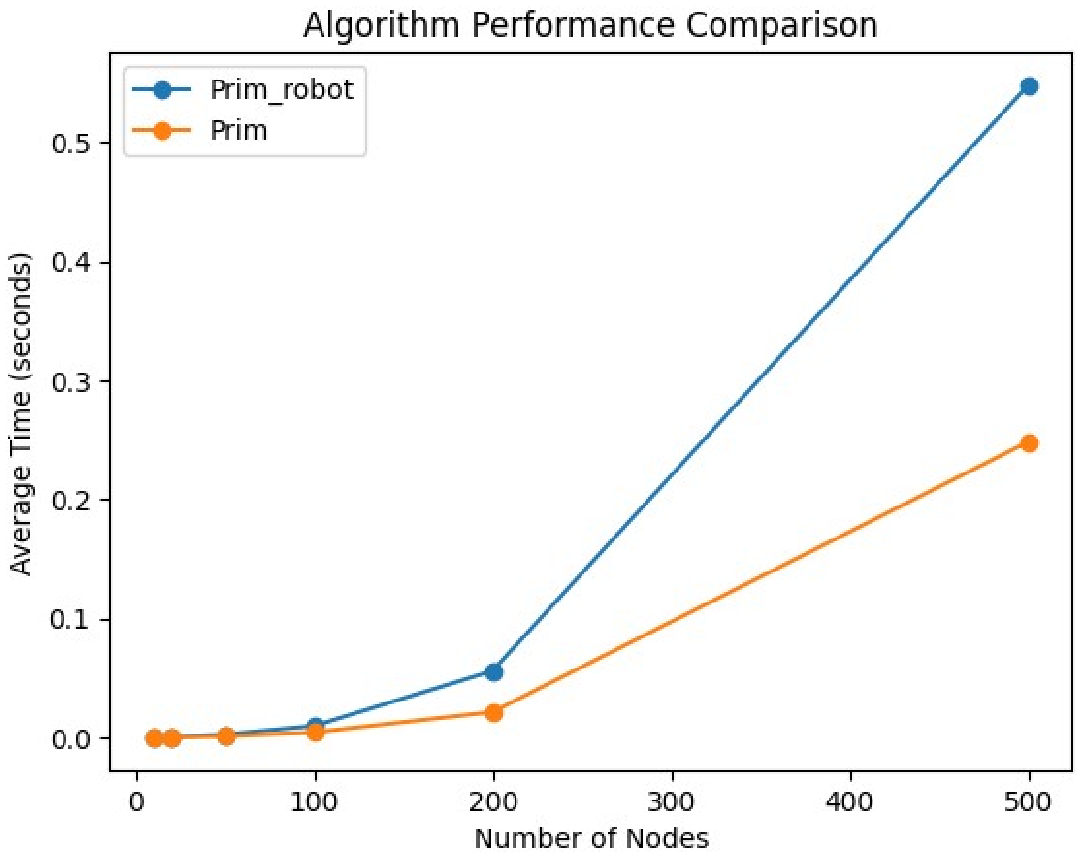
```
        for num, key in enumerate(edges):
            if key[0] in visited_nodes and key[1] in visited_nodes:
                continue
            elif key[0] in visited_nodes or key[1] in visited_nodes:
                output_graph.add_edge(key[0], key[1], weight = key[2]
['weight'])
                edges.pop(num)
                visited_nodes = output_graph.nodes()
                break
    return output_graph

#For kruskal and prim
nums=[10,20,50,100,200,500]
average_times_k=[]
average_times_p=[]
average_times_rp=[]
average_times_rk=[]
for i in nums:
    G = gnp_random_connected_graph(i, 0.5, False, False)
    NUM_OF_ITERATIONS = 100
    time_taken = 0

    for _ in tqdm(range(NUM_OF_ITERATIONS)):
        kruskal = Kruskal(G)
        start = time.time()
        minimum_spanning_tree = kruskal.kruskal_algorithm()
        end = time.time()
        time_taken += end - start
    average_time_k = time_taken / NUM_OF_ITERATIONS
    average_times_k.append(average_time_k)

    for _ in tqdm(range(NUM_OF_ITERATIONS)):
        start = time.time()
        minimum_spanning_tree = prim(G)
        end = time.time()
        time_taken += end - start
    average_time_p = time_taken / NUM_OF_ITERATIONS
    average_times_p.append(average_time_p)

    for _ in tqdm(range(NUM_OF_ITERATIONS)):
        start = time.time()
        tree.minimum_spanning_tree(G, algorithm="kruskal")
        end = time.time()
        time_taken += end - start
    average_time_rk = time_taken / NUM_OF_ITERATIONS
    average_times_rk.append(average_time_rk)

    for _ in tqdm(range(NUM_OF_ITERATIONS)):
        start = time.time()
        tree.minimum_spanning_tree(G, algorithm="prim")
```

```
        end = time.time()
        time_taken += end - start
    average_time_rp = time_taken / NUM_OF_ITERATIONS
    average_times_rp.append(average_time_rp)
plt.plot(nums, average_times_rk, marker='o', label='Kruskal_robot')
plt.plot(nums, average_times_k, marker='o', label='Kruskal')
plt.title('Algorithm Performance Comparison')
plt.xlabel('Number of Nodes')
plt.ylabel('Average Time (seconds)')
plt.legend()
plt.show()
plt.plot(nums, average_times_rp, marker='o', label='Prim_robot')
plt.plot(nums, average_times_p, marker='o', label='Prim')
plt.title('Algorithm Performance Comparison')
plt.xlabel('Number of Nodes')
plt.ylabel('Average Time (seconds)')
plt.legend()
plt.show()
plt.plot(nums, average_times_k, marker='o', label='Kruskal')
plt.plot(nums, average_times_p, marker='o', label='Prim')
plt.title('Algorithm Performance Comparison')
plt.xlabel('Number of Nodes')
plt.ylabel('Average Time (seconds)')
plt.legend()
plt.show()
```

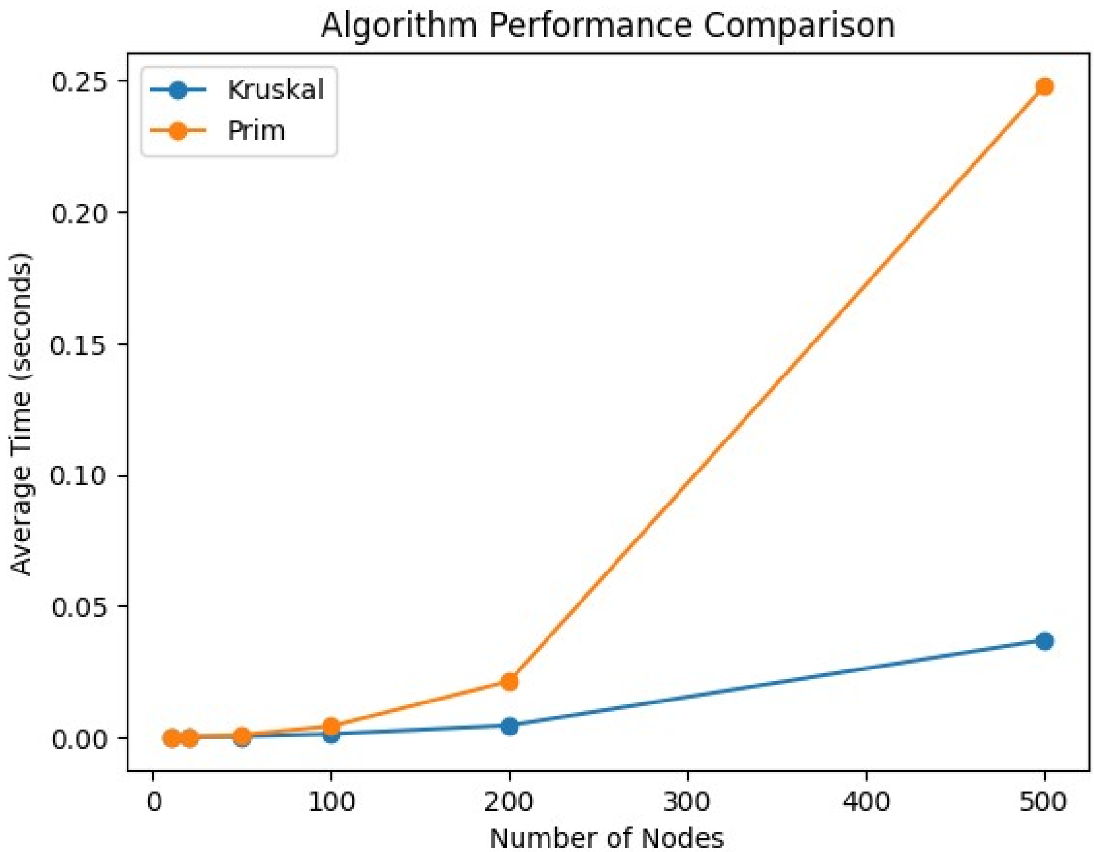
```
100%|██████████| 100/100 [00:00<00:00, 33338.40it/s]
100%|██████████| 100/100 [00:00<00:00, 24998.83it/s]
100%|██████████| 100/100 [00:00<00:00, 12486.39it/s]
100%|██████████| 100/100 [00:00<00:00, 24923.07it/s]
100%|██████████| 100/100 [00:00<00:00, 9090.98it/s]
100%|██████████| 100/100 [00:00<00:00, 10000.01it/s]
100%|██████████| 100/100 [00:00<00:00, 5263.41it/s]
100%|██████████| 100/100 [00:00<00:00, 8328.48it/s]
100%|██████████| 100/100 [00:00<00:00, 1587.32it/s]
100%|██████████| 100/100 [00:00<00:00, 1952.96it/s]
100%|██████████| 100/100 [00:00<00:00, 1015.31it/s]
100%|██████████| 100/100 [00:00<00:00, 1922.96it/s]
100%|██████████| 100/100 [00:00<00:00, 378.67it/s]
100%|██████████| 100/100 [00:00<00:00, 331.11it/s]
100%|██████████| 100/100 [00:00<00:00, 270.99it/s]
100%|██████████| 100/100 [00:00<00:00, 556.18it/s]
100%|██████████| 100/100 [00:01<00:00, 92.38it/s]
100%|██████████| 100/100 [00:01<00:00, 59.12it/s]
100%|██████████| 100/100 [00:02<00:00, 48.21it/s]
100%|██████████| 100/100 [00:01<00:00, 71.57it/s]
100%|██████████| 100/100 [00:08<00:00, 12.03it/s]
100%|██████████| 100/100 [00:21<00:00,  4.71it/s]
100%|██████████| 100/100 [00:18<00:00,  5.32it/s]
100%|██████████| 100/100 [00:11<00:00,  8.83it/s]
```



Algorithm Performance Comparison
Kruskal_robot
Kruskal
Average Time (seconds)
0.0
0.1
0.2
0.3
0.4
0
100
200
300
400
500
Number of Nodes



Algorithm Performance Comparison
Prim_robot
Prim
Average Time (seconds)
0.0
0.1
0.2
0.3
0.4
0.5
0
100
200
300
400
500
Number of Nodes



Видно, що алгоритм Прима встроєний в пайтон набагато довший за Богдановий так само з Крускалом, взагалом помітно,що Крускал швидший за Прима. Це впригнципі було очікувано оскільки складність Прима=O(V2), а Крускала=O(E log V).