Звіт

Команда: Сампара Софія і Шапошніков Михайло Артемій

Задача: реалізувати алгоритми (Прима і Крускала) для знаходження каркасів мінімальної ваги для графів різних розмірів та порівняти результати їхньої роботи з вбудованими алгоритмами networkx.algorithms.

1. Алгоритм Прима

```python
def Prima_algoritm(graph: 'nx.Graph'):
    spanning_tree = []
    sum_of_weights = 0
    list_of_nodes = list(graph.nodes)
    start = list_of_nodes[0]
    seen = set([start])
    list_of_nodes.remove(start)
    while list_of_nodes:
        lst = []
        for node in seen:
            for neighbour in graph.adj[node]:
                if neighbour not in seen:
                    lst.append((node, neighbour, graph.edges[node, neighbour]['weight']))
        node1, node2, weight = min(lst, key = lambda x: x[2])
        spanning_tree.append((node1, node2))
        seen.add(node2)
        sum_of_weights += weight
        list_of_nodes.remove(node2)
    return spanning_tree, sum_of_weights
```

Алгоритм Прима будує каркас найменшої ваги починаючи з заданої вершини. Наш алгоритм бере за початкову вершину 0. Повертає кортеж з двох елементів: список з кортежів, які представляють ребро та ціле число з мінімальної ваги цього каркасу.

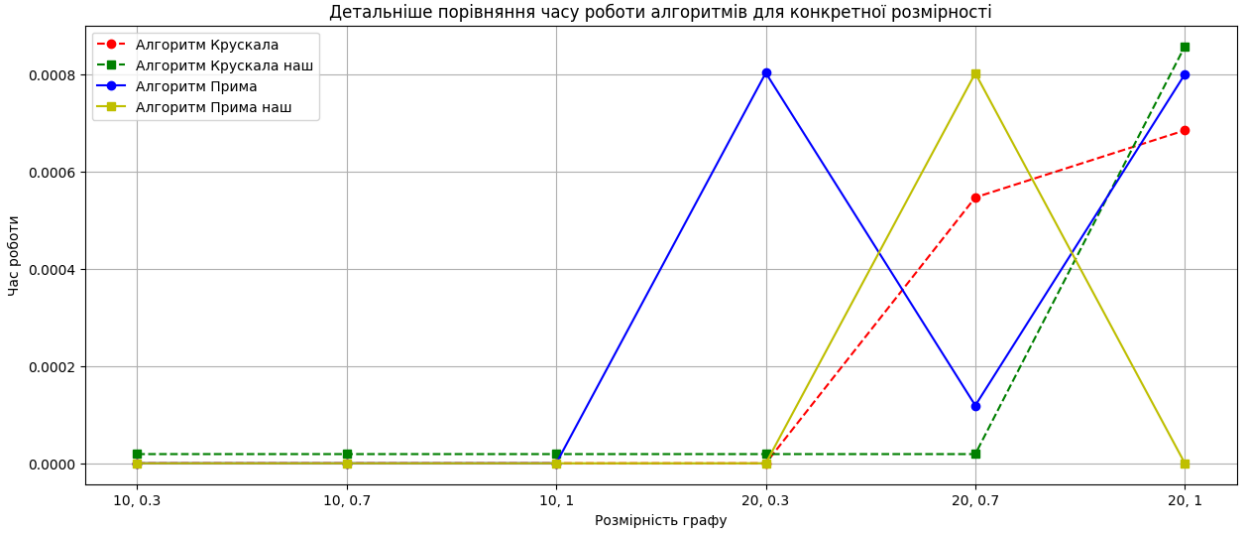
2. Алгоритм Крускала

```python
def find_set(node, sets, s_edges_i):
    for j, jth in enumerate(sets):
        if s_edges_i[node] in sets[j]:
            return jth
def Kruskal(graph):
    sum_of_weights = 0
    edges = list(graph.edges(data=True))
    s_edges = sorted(edges, key= lambda x : x[2]['weight'])
    sets = [set([x]) for x in range(len(graph.nodes))]
    res = []
    for i, ith in enumerate(s_edges):
        u = find_set(0, sets, s_edges[i])
        v = find_set(1, sets, s_edges[i])
        if u != v:
            res.append(ith[:2])
            sum_of_weights += ith[2]['weight']
            sets[sets.index(u)] = u.union(v)
            sets.remove(v)
    return res, sum_of_weights
```

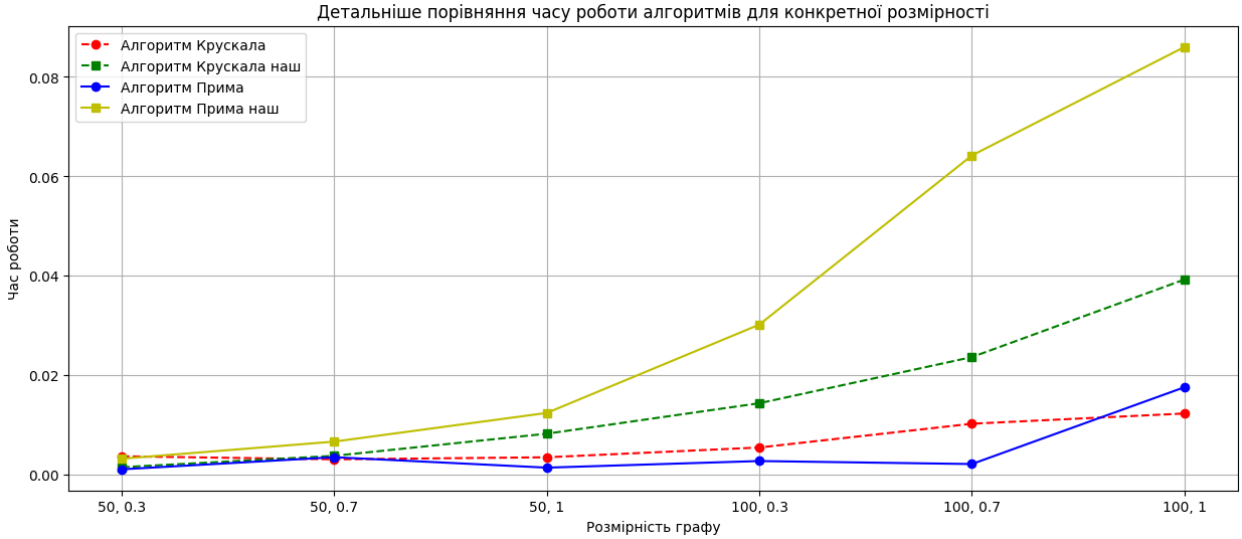
Алгоритм Крускала шукає каркас мінімальної ваги шукаючи ребра мінімальної ваги цілого графа. Алгоритм, імплементований нами, повертає кортеж з двох значень: список усіх ребер, що потрапили у мінімальний каркас та ціле число, як представляє вагу цього каркасу.

Порівняння результатів

Ми порівнювали наші алгоритми з вбудованими, експериментуючи на графах з різною кількістю вершин (10, 20, 50, 100, 200, 500) та різною заповненістю (0.3, 0.7, 1.0). Результати наступні.



Для цих значень наша реалізація алгоритму Крускала працює найкраще, окрім повного графа з 20 вершинами. А от алгоритм Прима дуже добре справився з повним графом на 20 вершин.



Для графів з більшими розмірами алгоритми працюють з очевидними закономірностями (чим більше вершин і ребер у графі, тим більше часу він працює). Наші алгоритми працюють довше, ніж вбудовані. Це, перш за все, пов’язано з тим, що багато часу тратиться на знаходження ребра мінімальної ваги. На цьому графіку видно, що наш алгоритм Прима не достатньо

оптимізований, адже алгоритм Крускала вимагає перегляд усіх ребер на кожному кроці, а от Прима лише частину.

Обрахунок результатів

```
NUM_OF_ITERATIONS = 10
time_kruskal = []
time_kruskal_our = []
time_prim = []
time_prim_our = []
for nodes in [10, 20, 50, 100, 200, 500]:
    for completeness in [0.3, 0.7, 1]:
        time_taken1 = 0
        time_taken2 = 0
        time_taken3 = 0
        time_taken4 = 0
        for i in tqdm(range(NUM_OF_ITERATIONS)):

            G = gnp_random_connected_graph(nodes, completeness, False)

            start = time.time()
            tree.minimum_spanning_tree(G, algorithm="kruskal")
            end = time.time()
            time_taken1 += end - start

            start = time.time()
            Kruskal(G)
            end = time.time()
            time_taken2 += end - start

            start = time.time()
            tree.minimum_spanning_tree(G, algorithm="prim")
            end = time.time()
            time_taken3 += end - start

            start = time.time()
            Prima_algoritm(G)
            end = time.time()
            time_taken4 += end - start

        time_kruskal.append(time_taken1 / NUM_OF_ITERATIONS)
        time_kruskal_our.append(time_taken2 / NUM_OF_ITERATIONS)
        time_prim.append(time_taken3 / NUM_OF_ITERATIONS)
        time_prim_our.append(time_taken4 / NUM_OF_ITERATIONS)

print(time_kruskal)
print(time_kruskal_our)
print(time_prim)
print(time_prim_our)
```

Висновок: Вбудовані алгоритми працюють майже з однаковою швидкістю. Наші імплементації відстають, хоча алгоритм Крускала досить оптимізований, в порівнянні з алгоритмом Прима. Бачимо закономірність, що чим більше вершин і ребер у графі, тим довше працюють алгоритми. Це пов’язано з тим, що ми постійно шукаємо ребра найменшої ваги або сортуємо їх, а це досить затратний по часу процес.