Звіт

до лабораторної роботи #1 з дискретної математики

Романа Мутеля і Марка Рузака

Вступ

У рамках першої частини даної лабораторної роботи необхідно було дослідити і порівняти ефективність алгоритмів Прима і Крускала, які використовуються для пошуку каркасного дерева мінімальної ваги для заданого графу.

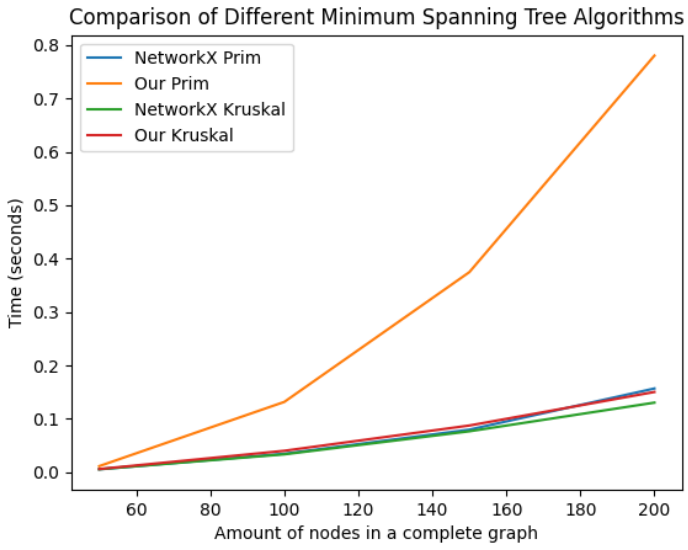
Експериментальна частина полягала в запуску алгоритмів на повних зважених графах з випадково згенерованими вагами ребер і різною кількістю вершин (50-100-...-700). Програмні коди надані нижче у звіті.

Специфікація обладнання

Експерименти проводились на ноутбуці з процесором Intel Core I5-10210u @ 1.60GHz (4 ядра, 8 потоків), 16 Гб ОЗУ, ОС Windows 10.

Результати експериментів

У першу чергу було протестовано алгоритми на повних графах. Паралельно з алгоритмами, реалізованими нами власноруч, тестувались алгоритми, вбудовані в модуль networkx.

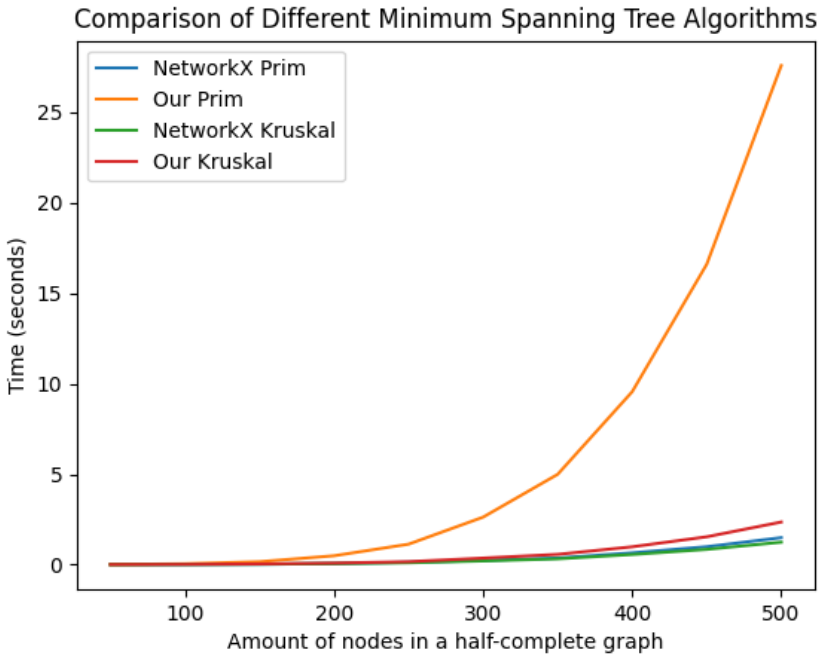


З даного графіку можна зробити висновок, що **наша реалізація алгоритму Прима помітно повільніша за алгоритм Краскала і вбудований алгоритм Прима.** Причиною є те, що для оптимізації пошуку використовують специфічні структури даних (min heap), які не використовувались у нашій реалізації.

Також, можна помітити, що на більших графах алгоритм Краскала працює швидше за алгоритм Прима. Більше того, якщо на повних графах з кількістю вершин до 160 наш алгоритм Краскала працював довше, то після позначки у 160 вершин, він почав працювати швидше за алгоритм Прима, вбудований у бібліотеку networkx

Розглянемо дані експерименту, під час якого алгоритми застосовувалися не на повному графі, а на такому, що ймовірність, що дві випадково вибрані вершини будуь з'єднаними, дорівнює 0.5.

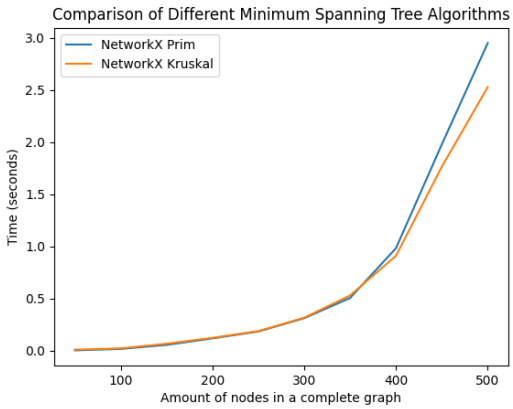
У такому випадку, **вбудований алгоритм Прима є швидшим за наш алгоритм Краскала, але все одно дещо повільнішим за вбудований алгоритм Краскала.**

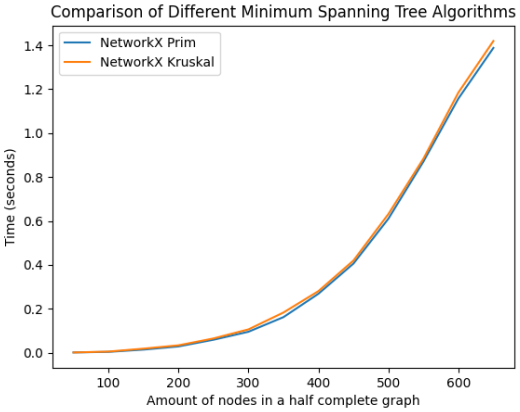


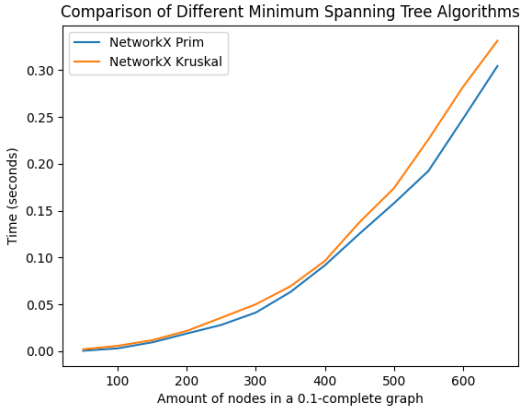
Для того, щоб об'єктивно порівняти саме алгоритми, а не якість їхньої реалізації, ми провели аналогічні експерименти для вбудованих реалізацій алгоритмів. Крім повного і "напівповного" графів, експеримент додатково

проводився на графі, в якому ймовірність суміжності двох випадкових вершин дорівнює 0.1.

З наступних трьох графіків можна зробити **ключовий** висновок, що **алгоритм Прима ефективніший на менш “щільних” графах**. Тобто на повному графі доцільніше використовувати алгоритм Краскала. На “напівповному” графі алгоритми працюватимуть приблизно за один і той самий час. Натомість, якщо середній степінь вершин у графі малий (наприклад, 0.1*n як у випадку з 0.1-complete graph на третій діаграмі), ефективнішим виявиться алгоритм Прима.







Підсумок

Алгоритми Прима і Краскала виконують однакову роботу та працюють майже з однаковою швидкістю. Головним критерієм при виборі між ними є кількість ребер у графі. Якщо граф скоріше повний, варто користуватись алгоритмом Краскала. Якщо ж граф скоріше дерево (тобто ступені вершин доволі малі), то варто користуватися алгоритмом Прима.

Програмний код алгоритмів

Алгоритм Прима (Марко Рузак)

```
def prims_algo(nxgraph: nx.Graph) -> int:
    data = list(nxgraph.edges.data())
    visited = set()
    unvisited = set()
    for node in nxgraph.nodes:
        unvisited.add(node)
    num_of_nodes = len(unvisited)
    starting_edge = random.choice(list(unvisited))
    visited.add(starting_edge)
    unvisited.remove(starting_edge)
    adjacency = tuple(nxgraph.adjacency())
    num_of_visited = 1
    carcass_weight = 0
    while num_of_visited < num_of_nodes:
        minimum = float("inf")
        best_choice = -1
        for node in range(num_of_nodes):
            if node in visited:
                for unv_node in unvisited:
                    if unv_node in adjacency[node][1]:
                        if minimum > adjacency[node][1][unv_node]["weight"]:
                            minimum = adjacency[node][1][unv_node]["weight"]
                            best_choice = unv_node
        num_of_visited += 1
        carcass_weight += minimum
        unvisited.remove(best_choice)
        visited.add(best_choice)
```

```
    return carcass_weight
```

Алгоритм Краскала (Роман Мутель)

```
def kruskalls_tree(G: nx.Graph) -> nx.Graph:
    nodes_set = set()
    weight = 0
    spanning = nx.Graph()
    for node in G.nodes:
        nodes_set.add(frozenset([node]))
    for edge in sorted(list(G.edges(data=True)), key=lambda x:
x[2]['weight']):
        if len(spanning.edges()) == len(G.nodes) - 1:
            break
        for u_node_group in nodes_set:
            if edge[0] in u_node_group:
                if edge[1] in u_node_group:
                    break
                else:
                    for v_node_group in nodes_set:
                        if edge[1] in v_node_group:
                            spanning.add_edge(edge[0], edge[1], **edge[2])
                            weight += edge[2]['weight']
                            nodes_set -= {u_node_group, v_node_group}
                            nodes_set.add(frozenset(u_node_group | v_node_group))
                            break
                    break
    return spanning, weight
```

Програмний код експериментів

```
time_nx_prim = 0
y_nx_prim = []
time_prim = 0
y_prim = []
time_nx_kruskal = 0
y_nx_kruskal = []
time_kruskal = 0
y_kruskal = []
ITERATIONS = 10
nodes = [50 * i for i in range(1,14)]
fig, ax = plt.subplots()

for n in nodes:
    for i in tqdm(range(ITERATIONS)):
        G = gnp_random_connected_graph(n, 0.1, False)

        start = time.time()
        nx.minimum_spanning_tree(G, algorithm='prim')
        end = time.time()
        time_nx_prim += end - start

        start = time.time()
        prims_algo(G)
        end = time.time()
        time_prim += end - start

        start = time.time()
        nx.minimum_spanning_tree(G, algorithm='kruskal')
        end = time.time()
        time_nx_kruskal += end - start

        start = time.time()
        kruskalls_tree(G)
        end = time.time()
        time_kruskal += end - start

    y_nx_prim.append(time_nx_prim/ITERATIONS)
    y_prim.append(time_prim/ITERATIONS)
    y_nx_kruskal.append(time_nx_kruskal/ITERATIONS)
    y_kruskal.append(time_kruskal/ITERATIONS)

ax.plot(nodes, y_nx_prim, label='NetworkX Prim')
ax.plot(nodes, y_prim, label='Our Prim')
ax.plot(nodes, y_nx_kruskal, label='NetworkX Kruskal')
ax.plot(nodes, y_kruskal, label='Our Kruskal')
ax.legend()
plt.title('Comparison of Different Minimum Spanning Tree Algorithms')
plt.xlabel('Amount of nodes in a 0.1-complete graph')
plt.ylabel('Time (seconds)')
plt.savefig('01_complete_graph_nx.png')
plt.show()
```