З В І Т

До лабораторної роботи №1

“Порівняння ефективності роботи

алгоритмів Прима та Крускала”

Харабара Юрій, Зінчук Юрій

Перед нами поставили завдання:

1. Порівняти ефективність алгоритмів (Крускала та Прима) для знаходження найлегшого кістякового дерева графа.

яке ми успішно виконали, провели експерименти на різних даних та зробили висновок.

Суть експерименту до 1 завдання полягала у виконанні програми на різній кількості вершин велику кількість разів. Для кожної кількості вершин алгоритми проганялись по 100 раз кожен. Завдяки цьому ми мали можливість порівняти ефективність цих алгоритмів на різних графах.

Специфікація комп'ютера, на якому проводилось тестування

Кількість ядер – 8 (8 потоків)
Тактова частота (2.0-4.1 ГГц)
RAM - 16 DDR4(3200 МГц)
OS - Arch Linux

Prim and Kruskal algorithms

Алгоритм Крускала:

Вхід: *інформація про граф у форматі tuple(list of edges, list of nodes)*

Вихід*: tuple(Дерево (список його ребер), вага)*

```
    E = sorted(graph_info[0], key=lambda x: x[2])
    connected_nodes = set()
    isolated_groups = {}
    T = list()

    for edge in E:
        v1, v2 = edge[0], edge[1]
        if v1 not in connected_nodes or v2 not in connected_nodes:
            if v1 not in connected_nodes and v2 not in connected_nodes:
                isolated_groups[v1] = [v1, v2]
                isolated_groups[v2] = isolated_groups[v1]
            else:
                if v1 in connected_nodes:
                    isolated_groups[v1].append(v2)
                    isolated_groups[v2] = isolated_groups[v1]
                else:
                    isolated_groups[v2].append(v1)
                    isolated_groups[v1] = isolated_groups[v2]
            connected_nodes.add(v1)
            connected_nodes.add(v2)
            T.append(edge)

    for edge in E:
        v1, v2 = edge[0], edge[1]
        if v2 not in isolated_groups[v1]:
            isolated_groups[v1] += isolated_groups[v2]
            gr2 = set(isolated_groups[v2])
            for node in gr2:
                isolated_groups[node] = isolated_groups[v1]

            T.append(edge)

    weight = 0
    for i in T:
        weight += i[2]

    return T, weight
```

Алгоритм Прима:

Вхід: *інформація про граф у форматі tuple(list of edges, list of nodes)*

Вихід: *tuple(Дерево (список його ребер), вага)*

```
def prim_algorithm(graph, weight=0):
    length = len(graph[1])
    connected_nodes = [0]
    tree = []

    while len(connected_nodes) != length:
        edge = get_minimal_weigth(graph[0], connected_nodes, tree)
        if edge == math.inf:
            break
        tree.append(edge)
        if edge[0] not in connected_nodes:
            connected_nodes.append(edge[0])
        if edge[1] not in connected_nodes:
            connected_nodes.append(edge[1])
    for i in tree:
        weight += i[2]
    tree = sorted(tree, key=lambda x: x[2])
    return tree, weight
```

Для цього алгоритму використовується допоміжна функція “get_minimal_weigth”, яка приймає список ребер графа, список вже використаних (з’єднаних) вершин, та список ребер дерева (уже готової його частини).

```python
def get_minimal_weigth(graph_edges, connected_nodes, tree):
    used_points = set()
    for verticles in connected_nodes:
        edge = min(graph_edges, key=lambda x: x[2] if
                   ((x[0] == verticles or x[1] == verticles) and
                   (x[0] not in connected_nodes or
                    x[1] not in connected_nodes)) else math.inf)
        used_points.add(edge)
    for i in tree:
        if i in used_points:
            used_points.remove(i)
    dont_needed = set()
    for j in used_points:
        if j[0] in connected_nodes and j[1] in connected_nodes:
            dont_needed.add(j)
    used_points = used_points-dont_needed
    edge = min(used_points, key=lambda x: x[2])
    return edge
```

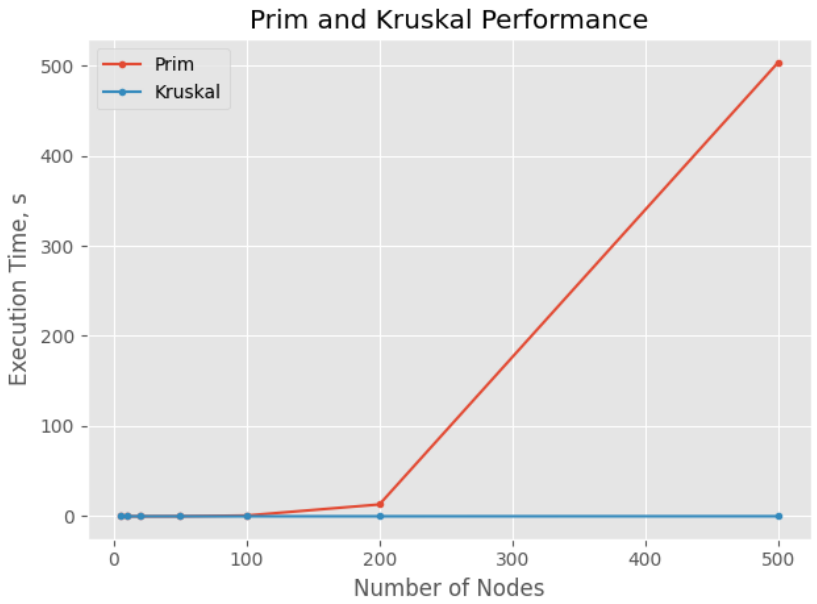
Програмний код проведення експериментів:

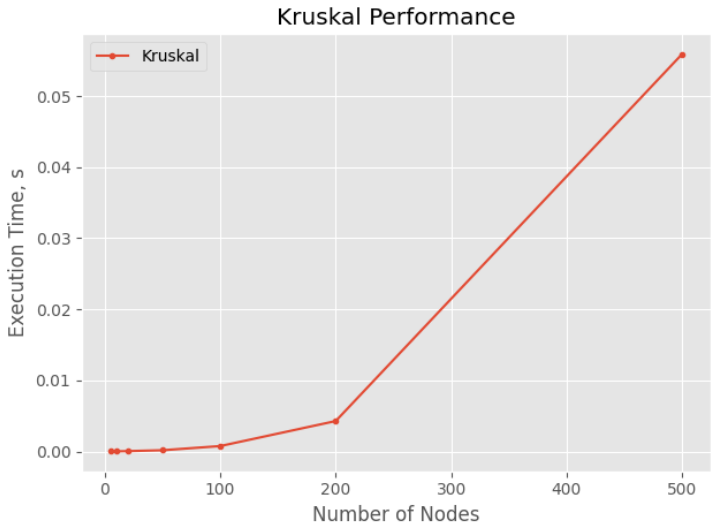
```python
NODES = [5, 10, 20, 50, 100, 200, 500]
stat = {
    'Prim': [],
    'Kraskal': []
}

for num_of_nodes in NODES:
    print('\n{} nodes'.format(num_of_nodes))

    # Test Prim
    time_taken = 0
    for _ in tqdm(range(num_of_iterations)):
        graph_info = get_info(num_of_nodes, completeness=0.25)
        start = time.perf_counter()
        prim_algorithm(graph_info)
        end = time.perf_counter()

        time_taken += end - start

    avg_time = time_taken/num_of_iterations
    stat['Prim'].append(
        {
            'num_of_nodes': num_of_nodes,
            'avg_time': avg_time
        }
    )

    # Test Kruskal
    time_taken = 0
    for _ in tqdm(range(num_of_iterations)):
        graph_info = get_info(num_of_nodes, completeness=0.3)
        start = time.perf_counter()
        kruskal_algorithm(graph_info)
        end = time.perf_counter()

        time_taken += end - start

    avg_time = time_taken/num_of_iterations
    stat['Kraskal'].append(
        {
            'num_of_nodes': num_of_nodes,
            'avg_time': avg_time
        }
    )
```

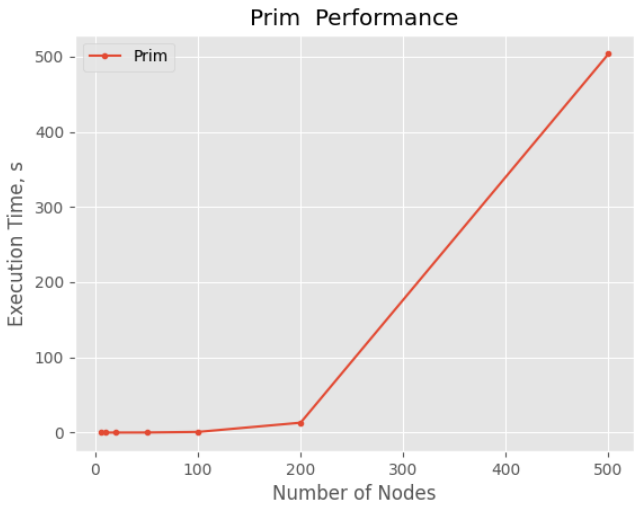
Для порівняння швидкості алгоритмів та побудови графіків був створений json файл:


```
{
    "Prim": [
        {
            "num_of_nodes": 5,
            "avg_time": 3.4096381132258105e-05
        },
        {
            "num_of_nodes": 10,
            "avg_time": 0.00024534239986678585
        },
        {
            "num_of_nodes": 20,
            "avg_time": 0.002717934950342169
        },
        {
            "num_of_nodes": 50,
            "avg_time": 0.08082968006958254
        },
        {
            "num_of_nodes": 100,
            "avg_time": 0.8550576343698776
        },
        {
            "num_of_nodes": 200,
            "avg_time": 13.126579095500347
        },
        {
            "num_of_nodes": 500,
            "avg_time": 503.563670481841
        }
    ],
    "Kraskal": [
        {
            "num_of_nodes": 5,
            "avg_time": 8.627789793536066e-06
        },
        {
            "num_of_nodes": 10,
            "avg_time": 1.7719230236252776e-05
        },
        {
            "num_of_nodes": 20,
            "avg_time": 4.761103962664492 4e-05
        },
        {
            "num_of_nodes": 50,
            "avg_time": 0.0001696554299269337
        },
        {
            "num_of_nodes": 100,
            "avg_time": 0.0007572049395821522
        },
        {
            "num_of_nodes": 200,
            "avg_time": 0.004276702203998 9
        },
        {
            "num_of_nodes": 500,
            "avg_time": 0.055903752179874575
        }
    ]
}
```


Після чого була реалізована програма для візуалізації цих даних:







Висновок

На основі проведених експериментів, можна сказати, що алгоритм Крускала на великій кількості вершин справляєтсья значно краще, адже має лінійну алгоритмічну складність відносно ребер, чого не можна сказати про алгоритм Прима. Також це прекрасно видно з графіків. Отже якщо невелика кількість вершин (100 і менше) то алгоритм Прима та Крускала досить близькі, проте на більшій кількості варто використовувати Крускала. Отже, незважаючи на те, що ручкою на папері зручніший алгоритм Прима, Алгоритм Крускала все ж ефективніший!