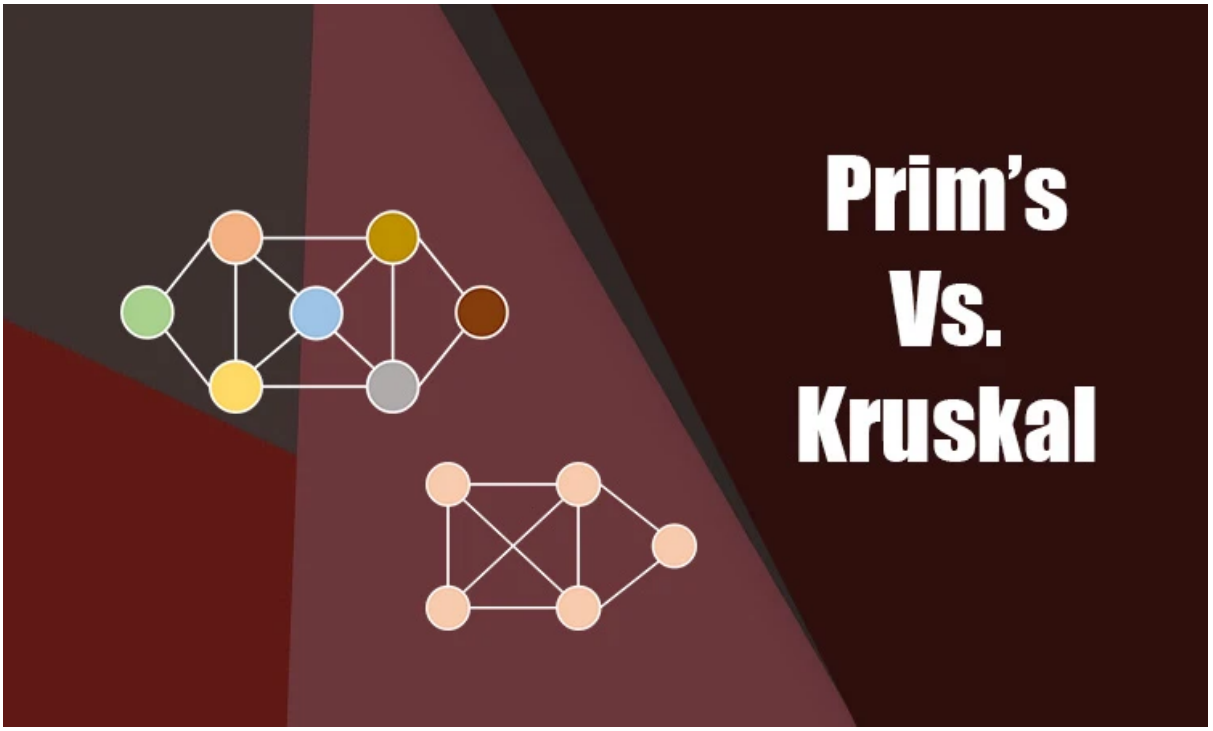
Звіт



Суть експерименту

Потрібно визначитись, який із двох алгоритмів - Краскала чи Прима - працює швидше. На вхід написана нами Python-програма приймає дані від користувача (кількість вершин у графі, його “наповненість” та кількість ітерацій для виконання) та повинна порівняти ефективність кожного алгоритму.

Ми проводили експеримент на Asus Tuf Gaming A15 з натупними характеримтиками:

- 16 GB RAM
- 8 ядер, 16 потоків

- мінімальна тактова частота: 2.9 ГГц

Код

В нашому проеткі є 4 модулі Python, а саме:

create_graph.py - модуль, який містить функцію для створення випадкових графів - `gnp_random_connected_graph()`.

kruskal_algorithm.py - модуль, який містить функцію `kruskal_algorithm()` для пошуку каркаса в графах і ще дві допоміжні функції:

```
def kruskal_algorithm(graph: nx.Graph):
    """
    An algorithm that returns a minimum-weight
    spanning-tree of a connected graph 'graph'.
    Args:
        graph (nx.Graph): a graph (class networkx.Graph)
    Returns:
        a graph: spanning tree with minimum weight (class networkx.Graph)
    """
    spanning_tree = nx.Graph()

            edges_weight   =   sorted(graph.edges(data=True),   key=lambda   x:
x[2]["weight"])
    quantity_of_vertcies = len(graph.nodes())
    all_sets = [set([vertice]) for vertice in range(quantity_of_vertcies)]

    chosen_edges = 0
    for elem in edges_weight:
        if different_sets(all_sets, elem[0], elem[1]):
            spanning_tree.add_edge(elem[0], elem[1])
            chosen_edges += 1
            if chosen_edges == quantity_of_vertcies - 1:
                return spanning_tree
            all_sets = change_sets(all_sets, elem[0], elem[1])
```

prim_algorithm.py - модуль, який містить функцію `prim_algorithm()` для пошуку каркаса в графах:

```
      def prim_algorithm(graph:nx.Graph) -> nx.Graph:
          """
          Generates a minimum spanning tree and returns it.
          Args:
              graph (nx.Graph): a graph in which we want
              to find the minimum spanning tree.
          Returns:
              nx.Graph: a spanning tree found.
          """
          spanning_tree = nx.Graph()
          spanning_tree.add_node(list(graph.nodes())[0])
          incident_edges = set()
          last_added_node = list(graph.nodes())[0]
          while len(spanning_tree.nodes) != len(graph.nodes):
                # Searching for new incident edges to nodes of already built
tree
               new_edges = {(edge[0], edge[1], edge[2]['weight']) for edge in
graph.edges(last_added_node, data=True)}
              incident_edges = incident_edges.union(new_edges)
              # print(f"incident_edges: {incident_edges}")
              # Deleting those that would form a circuit
               incident_edges -= {edge for edge in incident_edges if edge[0]
in spanning_tree.nodes and edge[1] in spanning_tree.nodes}
              # print(f"incident_edges: {incident_edges}")
               # Searching for a cheapest edge, which is incident to already
added nodes
              min_edge = min(incident_edges, key=lambda edge: edge[2])
              # Adding it to our tree
                  spanning_tree.add_edge(min_edge[0], min_edge[1], weight =
min_edge[2])
                     last_added_node = min_edge[0] if min_edge[0] not in
list(spanning_tree.nodes) else min_edge[1]
              # Removing it from our graph
              graph.remove_edge(min_edge[0], min_edge[1])
          return spanning_tree
```

time_measurement.py - модуль для порівняння двох алгоритмів. Оскільки його сирцевий код дуже великий, ми не помістили його сюди. Аби отримати його, відвідайте GitHub-репозиторій проекту.

При написанні алгоритмів ми використовували networkx.Graph як тип подання графів та бібліотеку matplotlib для генерації графіків.

Результати

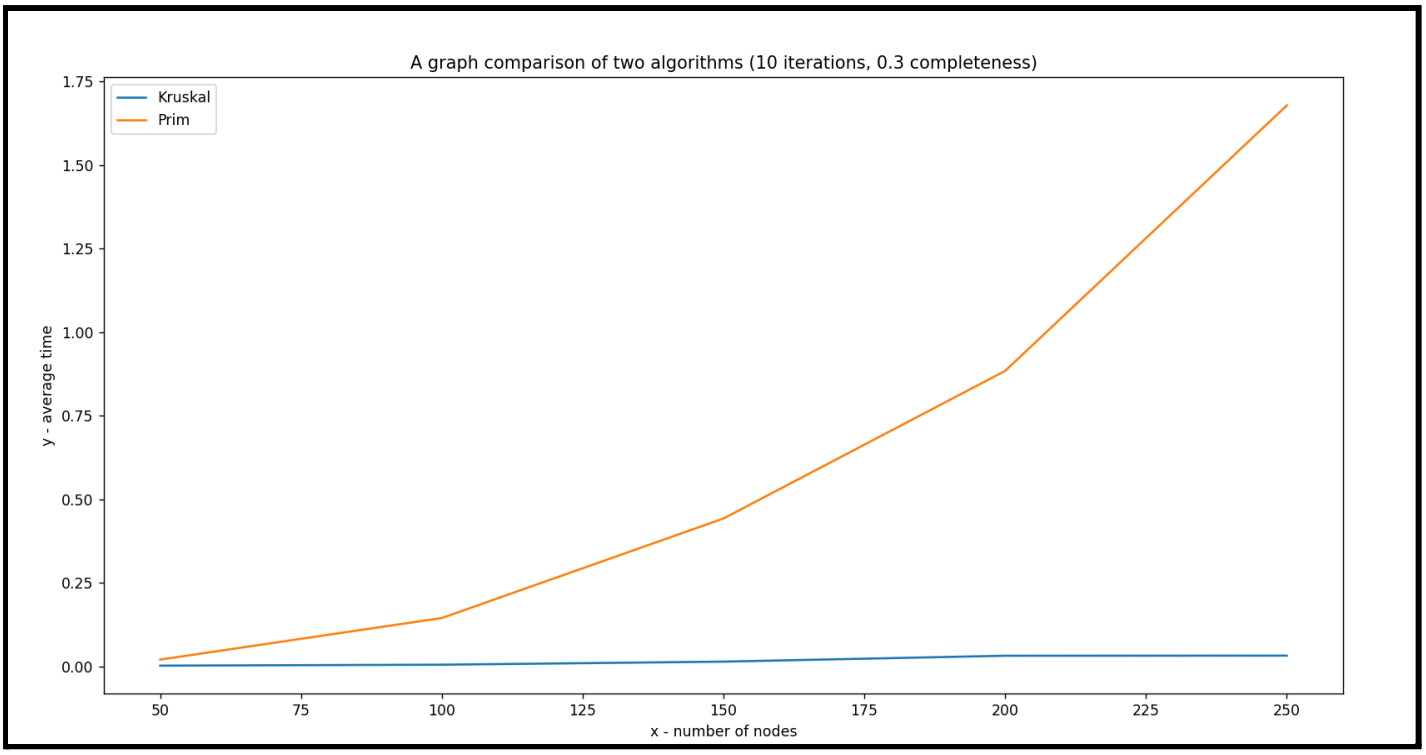
Запустивши програму для таких наборів даних:

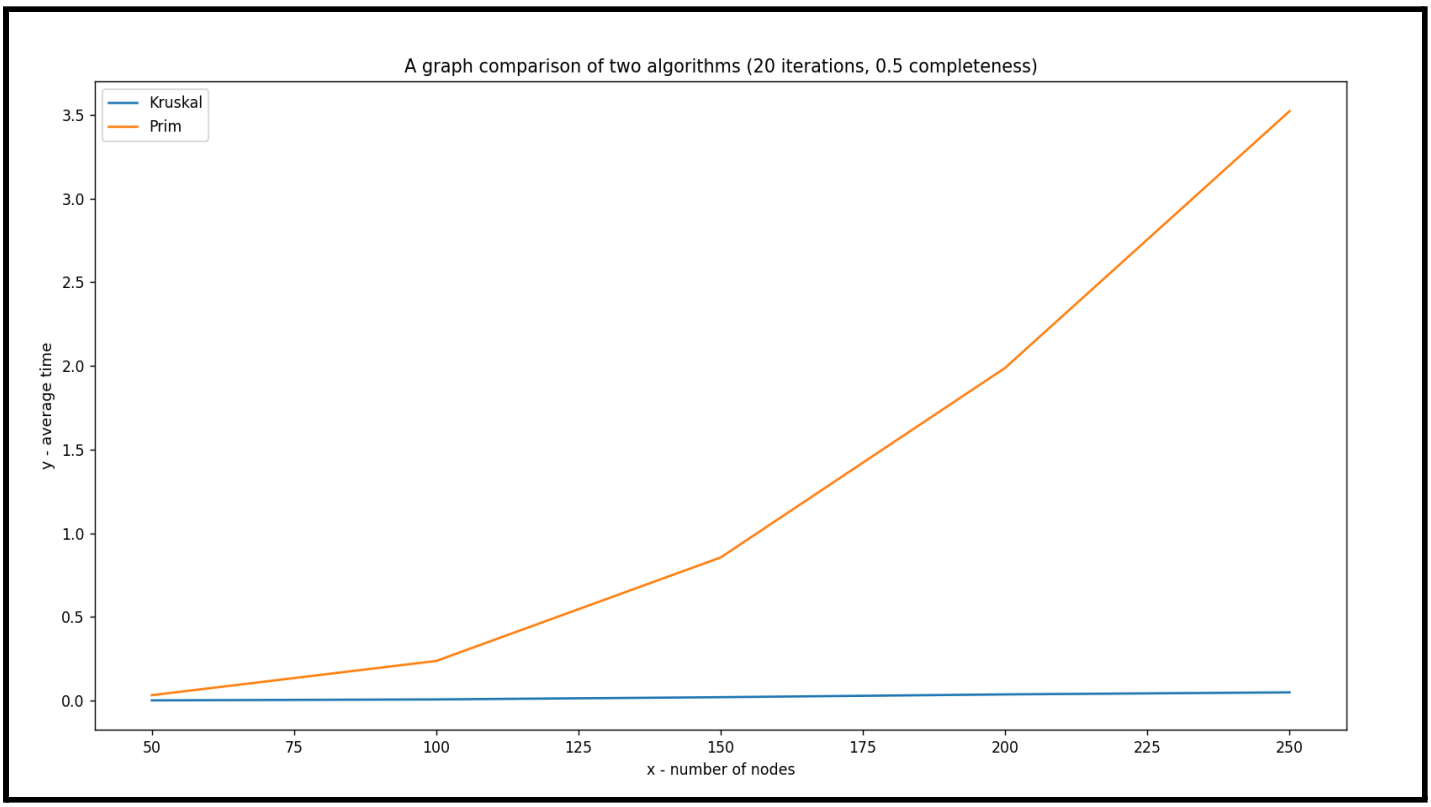
1. Кількість ітерацій: 10, Кількість вершин: 50, завершеність: 0.3;
2. Кількість ітерацій: 20, Кількість вершин: 70, завершеність: 0.5;
3. Кількість ітерацій: 30, Кількість вершин: 20, завершеність: 0.9;
4. Кількість ітерацій: 100, Кількість вершин: 10, завершеність: 0.2

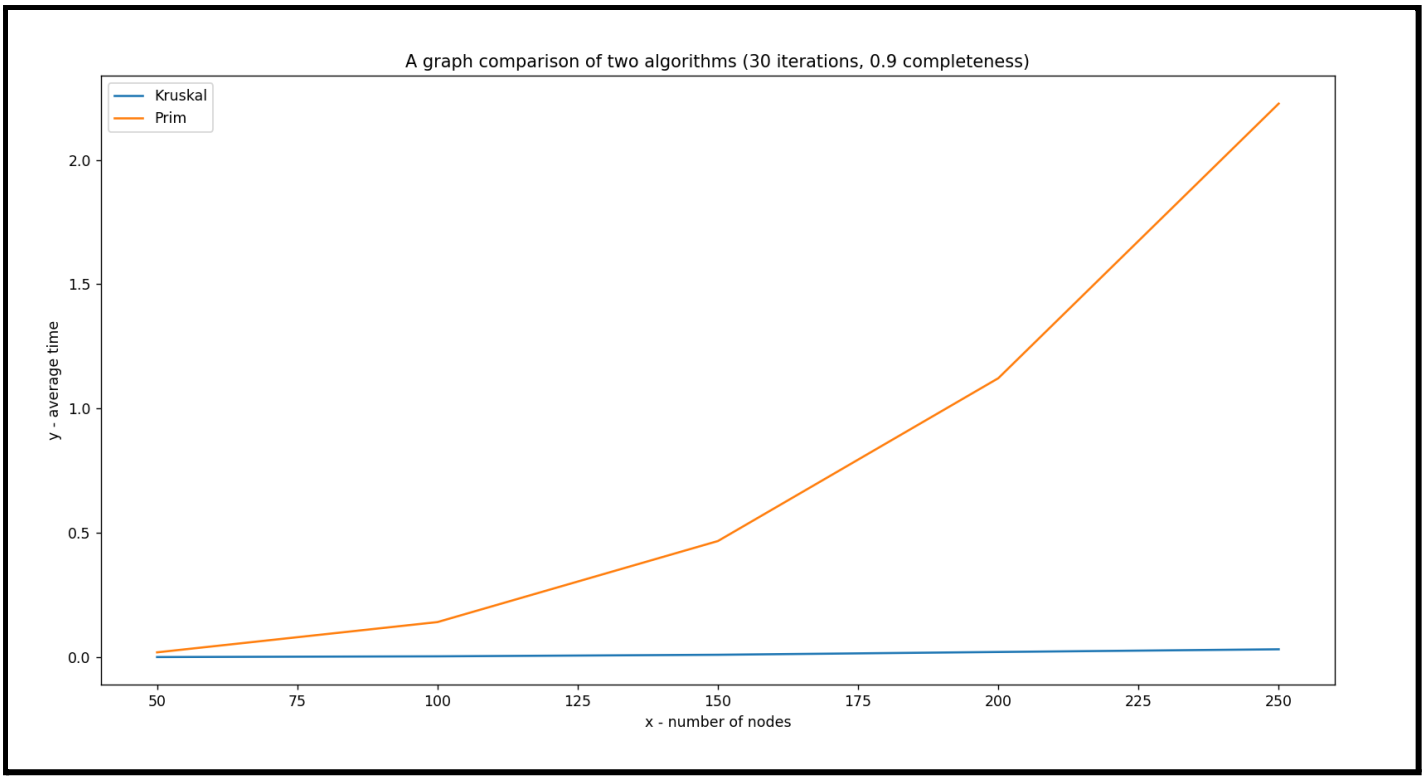
… ми отримали наступні результати середнього часу виконання (в секундах):

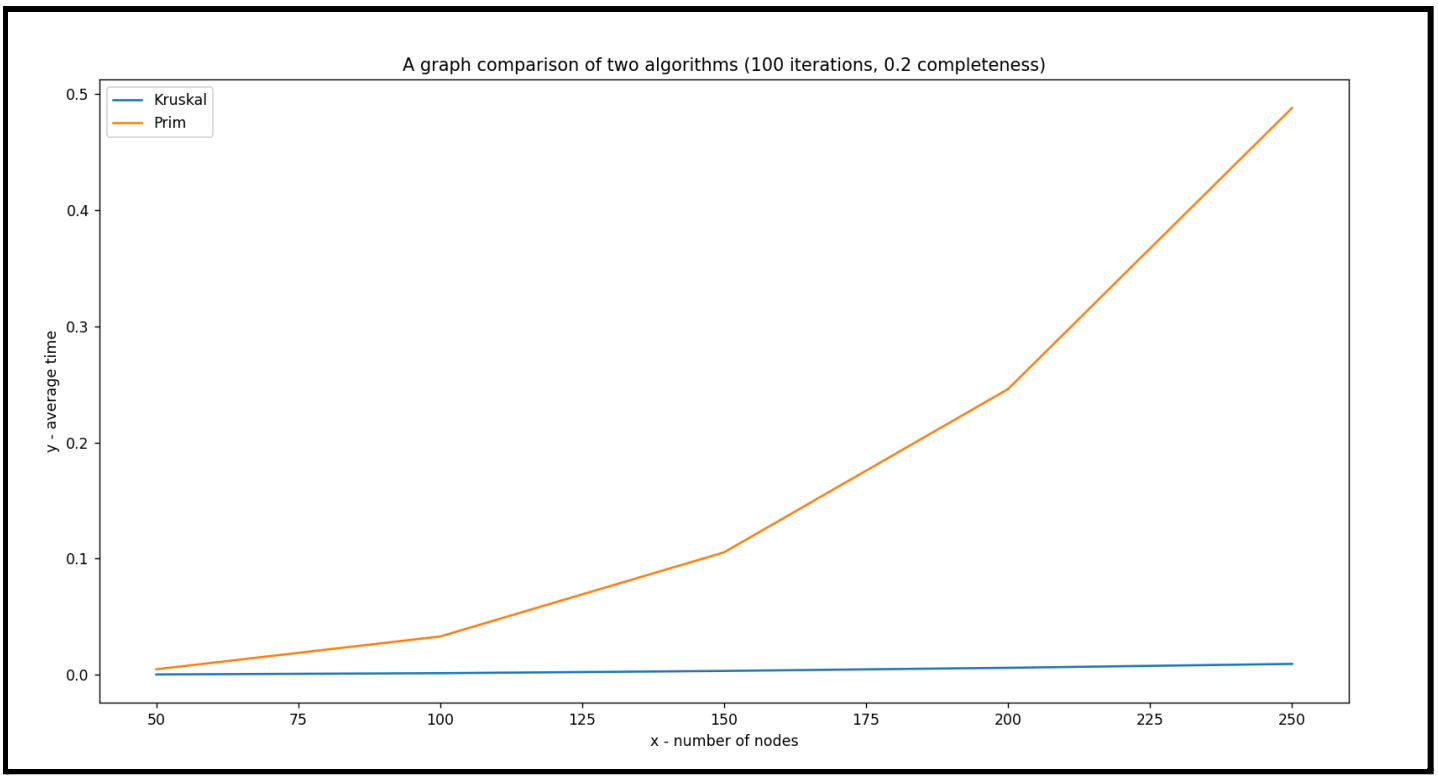
	1	2	3	4
Краскал	0.0011	0.0030	0.0003	0.00005
Прим	0.0208	0.0865	0.0015	0.00014

Спираючись на ті ж значення кількості ітерацій та завершеності графів, ми отримали наступні графіки:


A graph comparison of two algorithms (10 iterations, 0.3 completeness)
Kruskal
Prim
y - average time
x - number of nodes
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
50
75
100
125
150
175
200
225
250


A graph comparison of two algorithms (20 iterations, 0.5 completeness)
Kruskal
Prim
y - average time
x - number of nodes
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
50
75
100
125
150
175
200
225
250


A graph comparison of two algorithms (30 iterations, 0.9 completeness)
Kruskal
Prim
y - average time
x - number of nodes
0.0
0.5
1.0
1.5
2.0
50
75
100
125
150
175
200
225
250


A graph comparison of two algorithms (100 iterations, 0.2 completeness)
Kruskal
Prim
y - average time
x - number of nodes
0.0
0.1
0.2
0.3
0.4
0.5
50
75
100
125
150
175
200
225
250

Підсумок

Отже, серед написаних нами алгоритмів, найкращим виявився метод Крускала. Це пояснюється багатьма факторами, головні з яких:

- простіша реалізація. Відсортувавши масив ребер один раз, нам лише потрібно на кожному кроці вибирати мінімальне за вагою, відкидаючи лише ті, що при додаванні утворили б цикл з деревом.
- не потрібно шукати інцидентних ребер. В алгоритмі Прима ж на кожному кроці до множини інцидентних ребер ми додавали нові - інцидентні до останньо доданої вешини. Проте потім серед інцидентних ребер проводиться пошук “найкоротшого”, що в найгіршому випадку O(n*log(n)). Вирішення цієї проблеми є - потрібно не просто дописувати нові ребра, а вставляти їх у “правильні” місця - таким чином матимемо посортований контейнер, ребро в якому шукатиметься за O(log(n)).
- в алгоритмі Прима ми використовували метод edges([nodes]) класу networkx.Graph. При переданні непустого списку [nodes] у нього, ми можемо отримати інцидентні ребра до кожної з вершин у списку. Проте такий підхід неефективний - найкраще було б за O(1) отримувати інформацію про це за допомогою

спочатку підготованого словника. Ключем в такому словнику є номер вершини, а значенням - список ребер, інцидентних їй.

Якщо взяти до уваги покращення, які можна поширити на алгоритм Прима, то його ефективність буде навіть більшою при досить великій “наповненості” графа. В той же час доцільно використовувати Краскала, аби знайти каркас у графах, де більшість вершин має відносно малий степінь. Такі висновки ми зробили на основі тих же експериментальних даних, проте з використанням вбудованих в networkx алгоритмів Краскала і Прима.

Учасники команди: Ярослав Корч, Демчук Назар.

GitHub: https://github.com/frezario/KruskalVsPrim.git