Звіт до лабораторної роботи №1 з дискретної математики

Виконали Швець Анастасія та Лушней Святослав.

Перед нами постало завдання порівняти ефективність алгоритмів Прима та Краскала для побудови каркасу для графа. Для вирішення цієї проблеми ми реалізували дані алгоритми на мові програмування Python і провели тестування на різних даних.
Експерименти проводились на ноутбуці Acer Swift-314sf з процесором AMD Ryzen 4500U (6 ядер, 6 потоків) із базовою частотою 2,3 Ггц і максимальною 4,2 Ггц, та RAM 16Gb. Операційна система Manjaro 21..

Код із короткими коментарями реалізації наведений нижче:

Алгоритм Прима із асимптотичною складністю O(E log(V)) за рахунок реалізації збереження даних в бінарному дереві із використанням heapq .

```python
import heapq
import networkx as nx


def prim_mst(graph: nx.Graph):
    """

    Find and return the graph (the tree) with tha minimum
    spanning tree of the graph by using of the prim's algorithm
    :graph - networkx Graph object
    Return: tree with minimum edge costs of the set graph

    """
    number_of_vertixes = graph.number_of_nodes()
    # create an emthy networkx graph to store results
    tree = nx.Graph()
    tree.add_nodes_from(range(number_of_vertixes))
    # change the graph into format of dictionary with dictioanary
    # ex.: {'A': {'B: {'weight': 1}, 'C': {'weight': 5}...}, 'B'...}
    nx.to_dict_of_dicts(graph)
    # initiate list with bool value that show wheather the node was used ( first )
    used = [True] + [False]*(number_of_vertixes-1)
    # create edge_list with увпуі that has at least one used node
    edges = [(dct['weight'], 0, node) for node, dct in graph[0].items()]
    # make it into binary tree
    heapq.heapify(edges)
    # do while there are still edges with one
```

```
# do while there are still edges with one
while edges:
    weight, prev_node, new_node = heapq.heappop(edges)
    # cheack wheather node on the other side of edge was used
    # (if current edge is valid)
    if not used[new_node]:
        # change value in used list ot the new_node
        used[new_node] = True
        # add the edge to the ansver graph
        tree.add_edge(prev_node, new_node, weight=weight)
        # iterate on edges that are incident to the new node
        # a heaped date with potentialy valid edges
        for nxt, edge_data_dct in graph[new_node].items():
            # add edges that has one used node and one not used (at the present time)
            # to the heap dataset with potentialy valid edges
            if not used[nxt]:
                heapq.heappush(
                    edges, (edge_data_dct['weight'], new_node, nxt))
return tree
```

Алгоритм Краскала, за рахунок об’єднання множин із врахування батьків вершин і сортування ребер реалізовано із складністю O(E log(V)).

```
import networkx as nx


def kruskal_mst(graph: nx.Graph) -> nx.Graph:
    def find_class(node):
        """
        return class of the node
        """
        # find class of the node while index
        while node != classes[node]:
            # change node to its 'parent' node
            node = classes[node]
        return node

    def union(i_class: int, j_class: int) -> None:
        """
        unite two classes of the nodes
        """
        # check which level of tree is more
        # add tree of the lower level to the tree of the higher
        if level[i_class] > level[j_class]:
            classes[j_class] = i_class
        else:
            classes[i_class] = j_class
            # if two tree levels are equel change one of the levele
            if level[i_class] == level[j_class]:
                level[j_class] += 1
```

```
    number_of_vertixes = graph.number_of_nodes()
    # initialize starting classes (each node in the other set/class)
    classes = {i: i for i in range(number_of_vertixes)}
    # initialize starting level of each class(tree) equal to 0
    level = {i: 0 for i in range(number_of_vertixes)}
    # create ansver graph, add nodes
    tree = nx.Graph()
    tree.add_nodes_from(range(number_of_vertixes))
    # sort edges by its
    edges = sorted(graph.edges.data('weight'), key=lambda edge: edge[2])
    # iterate for edges
    for cur_edge in edges:
        # stop if we already created a tree
        if number_of_vertixes == 1:
            break
        # get new edge min cost frome the list of sorted edges
        i, j, weigt = cur_edge
        # find classes for each point and compare them
        i_class, j_class = find_class(i), find_class(j)
        # if nodes are not in the sama set/class
        if i_class != j_class:
            number_of_vertixes -= 1
            # add edge to the ansver graph
            tree.add_edge(i, j, weight=weigt)
            # unite two classes
            union(i_class, j_class)
    return tree
```

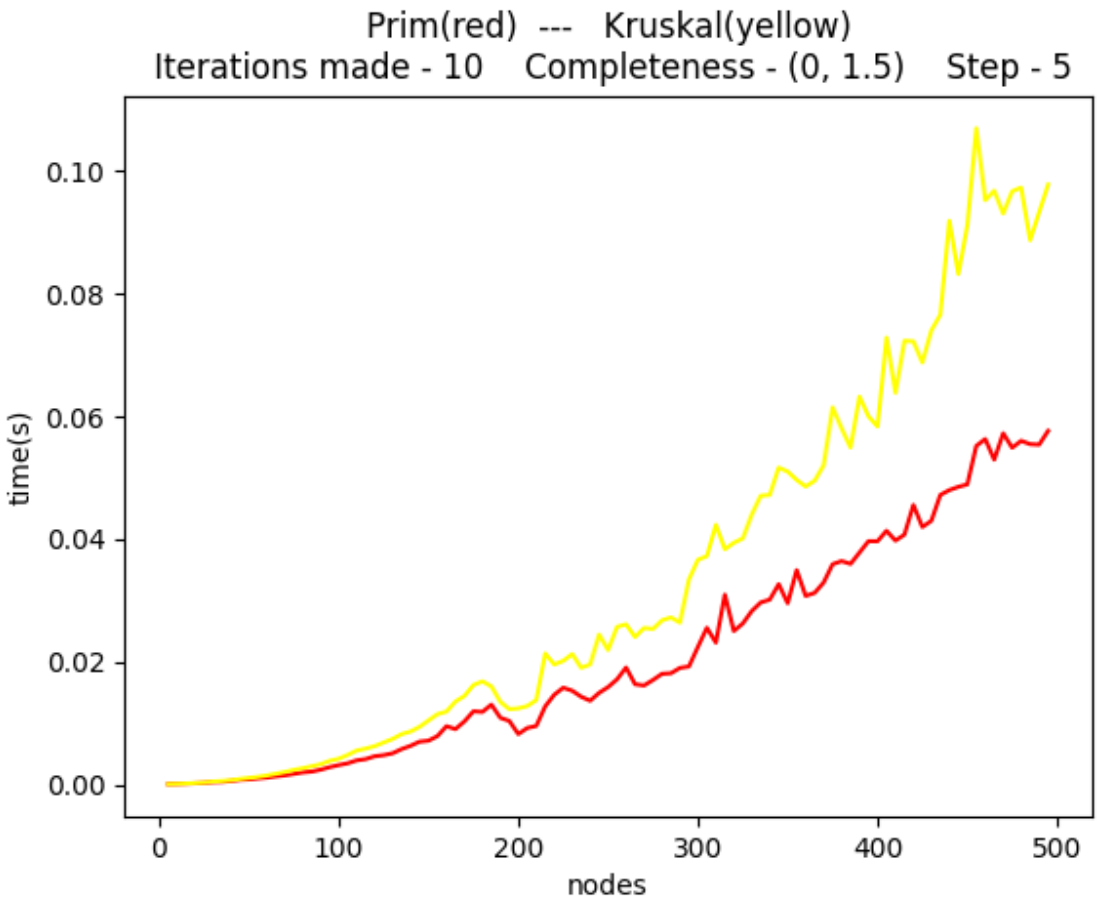
Для проведення тестування був створений окремий скрипт, із кодом якого можна ознайомитись за посиланням https://github.com/sviat-l/discrete_trees. Нижче наведений код з нього:

```python
import create_graph
import kruskal
import prim
import time
import matplotlib.pyplot as plt

number_of_iterations = 10
number_of_nodes = 500
completeness = 1
step = 5

plt.xlabel('nodes')
plt.ylabel('time(s)')
plt.title(f'Prim(red)  ---   Kruskal(yellow)\n\
Iterations made - {number_of_iterations}\
    Completeness - {completeness}    Step - {step}')
x_arguments, prim_times, kruskal_times = [], [], []
```

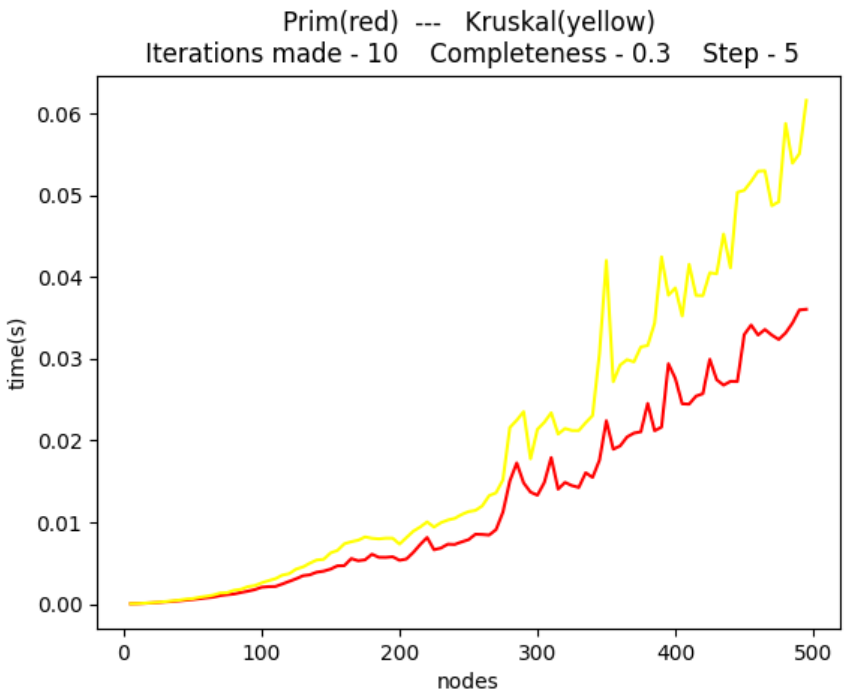
```python
def find_time(cur_nodes_num, flag='kruskal'):
    # choose algorithm
    if flag == 'prim':
        mst_algo = prim.prim_mst
    else:
        mst_algo = kruskal.kruskal_mst
    sum_times = 0
    for _ in range(number_of_iterations):
        graph = create_graph.gnp_random_connected_graph(
            cur_nodes_num, completeness)
        # calculate time
        st = time.time()
        mst = mst_algo(graph)
        sum_times += time.time() - st
    return sum_times/number_of_iterations

# iterate on nun nodes from 0 to max number
for vertecies in range(step, number_of_nodes, step):
    x_arguments.append(vertecies)
    prim_times.append(find_time(vertecies, "prim"))
    kruskal_times.append(find_time(vertecies, "kruskal"))

plt.plot(x_arguments, prim_times, label="prim", color="red")
plt.plot(x_arguments, kruskal_times, label="kruskal", color="yellow")
plt.savefig("plot")
```

Аналіз результатів

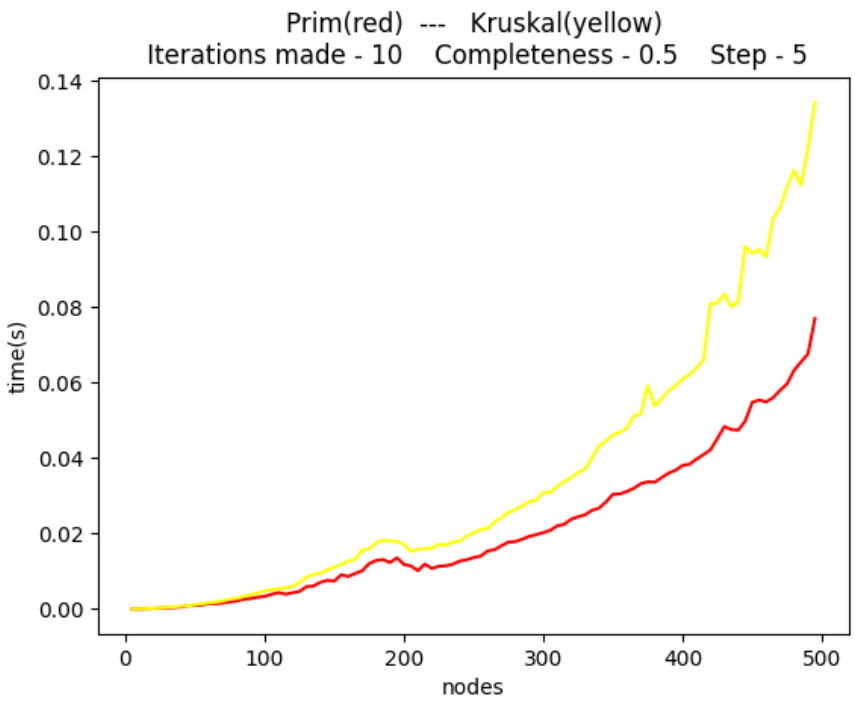
Алгоритми були протестовані на великі кількості різних наборів даних.
Примітка: після проведення великої кількості тестів, було помічено, що в назві графіків допущена помилка, і насправді червоним кольором у всіх графіках позначно дані, що відповідають алгоритму Краскала, а жовтим відповідно - алгоритму Прима.Отримані результати наведені нижче:



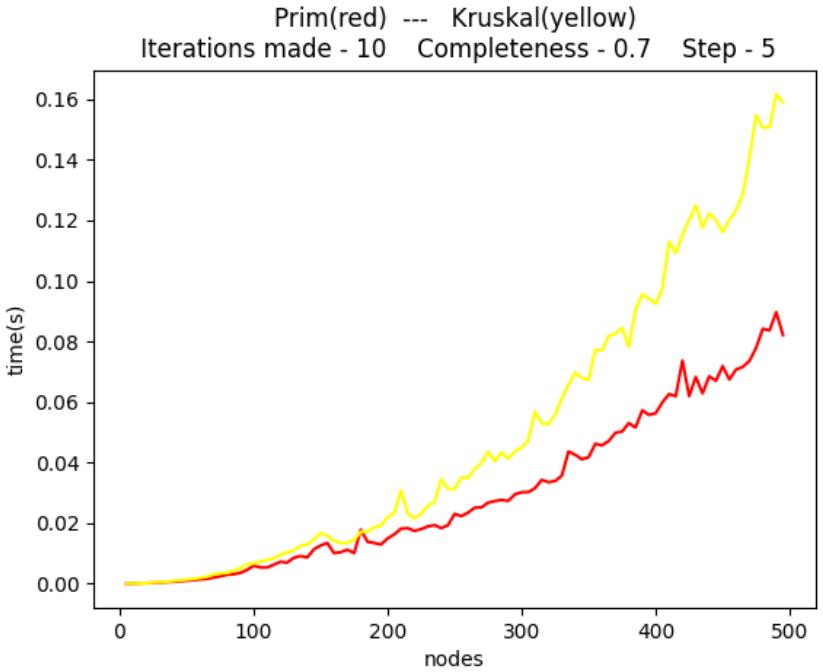
На першому графіку аналізувались дані для графів із ‘завершеністю’ 0.1, для графів із кількістю вершин від 5 до 500 із кроком 5. Алгоритм Крускала працює ефективніше ніж Пріма на всіх значеннях, різниця стає сильно поматною після 300 вершин. Різкий спад після ~200 вершин можна пояснити скиданням частоти процесора. Неплавність кривих також можна цим пояснити.



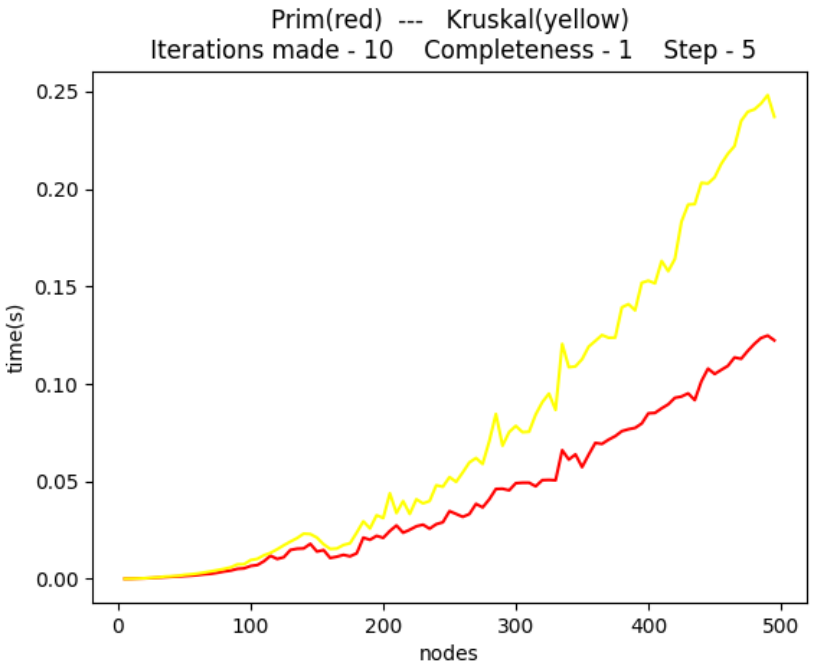
На другому графіку аналізувались дані для графів із ‘завершеністю’ 0.3. Алгоритм Крускала працює ефективніше ніж Пріма на всіх значеннях, втім різниця стає помітною не одразу, а лише пішля ~400 вершин.



Тут ми спостерігаємо плавне зростання часу виконання, і наглядно бачимо, що алгоритм Краскала є ефективнішим ніж Пріма, особливо на великих числах.



Із 4-го графіку, при заповненості 0,7 бачимо, що алгоритм Краскала має значну перевагу для значень кількості вершин до 200 і помітну перевагу при кількості вершин більшій ніж 400.



На останньому графіку, при повному графі, алгоритм Краскала показує ще більшу різницю в часі побудови каркасу.

Підсумок

Після проведення тестів, бачимо, що алгоритм Краскала є більш ефективним, ніж алгоритм Прима для побудови каркасів. При невеликій кількості вершин(до 300) обидва алгоритми показують доволі схожі результати(відношення часу виконання менше ніж 1.1), що не залежать від коефіцієнту заповненості. А при великих значеннях кількості ребер алгоритм Краскала є набагато ефективнішим(у 2-3 рази) за алгоритм Прима. Це легко пояснюється тим, що при збільшенні кількості даних, зростає кількість ребер, відповідно зростає і кількість ребер із меншими вагами. Для невеликого розкиду ймовірних ваг кожного ребра Це особливо помітно, коли при малих кількостях вершин,різниця між алгоритмами і мізерною бо кількість ребер, вага яких повторюється є малою і при ітеруванні по ребрах, часто цикл переривається на невеликій частині пройдених ребер. Це, на нашу думку, і пояснює чому алгоритм Прима працює повільніше ніж Краскала. Також в ході тестування спостерігалися різкі спади і стрибки часу виконання алгоритмів, що можна пояснити скиданням частоти роботи процесора, а також випадковістю генерування графів.