ДИСКРЕТНА МАТЕМАТИКА ЛАБОРАТОРНА РОБОТА №1

Порівняння ефективності роботи алгоритму Прима та Крускала.

Гоцко Ліліана
Польова Анна

Завдання: реалізувати алгоритми Прима та Краскала, порівняти їх ефективність для різних графів, враховуючи кількість вершин та ймовірності заповненості графа.

Специфікація комп'ютера:
Кількість ядер: 4
Тактова частота: до 3,70 ГГц
Пам'ять: 8
ОС: Windows 10

Kruskal:

```
nodes_set = set()
edge_dict = {}
base = []

#adding the edge from the graph to base if it
#is minimum weighted and not in the cycle
for edge in graph:
    #check if possible edge(nodes not in one set)
    if edge[0] not in nodes_set or edge[1] not in nodes_set:
        if edge[0] not in nodes_set and edge[1] not in nodes_set:
            edge_dict[edge[0]] = [edge[0], edge[1]]
            edge_dict[edge[1]] = edge_dict[edge[0]]
        else:
            if not edge_dict.get(edge[0]):
                edge_dict[edge[1]].append(edge[0])
                edge_dict[edge[0]] = edge_dict[edge[1]]
            else:
                edge_dict[edge[0]].append(edge[1])
                edge_dict[edge[1]] = edge_dict[edge[0]]

        base.append(edge)
        nodes_set.add(edge[0])
        nodes_set.add(edge[1])

#second round check to join sets
for edge in graph:
    if edge[1] not in edge_dict[edge[0]]:
        base.append(edge)
        gr1 = edge_dict[edge[0]]
        edge_dict[edge[0]] += edge_dict[edge[1]]
        edge_dict[edge[1]] += gr1
return base
```

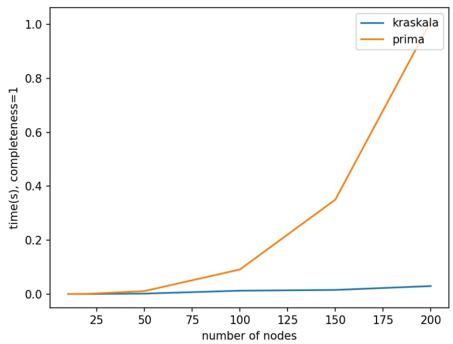
Prim:

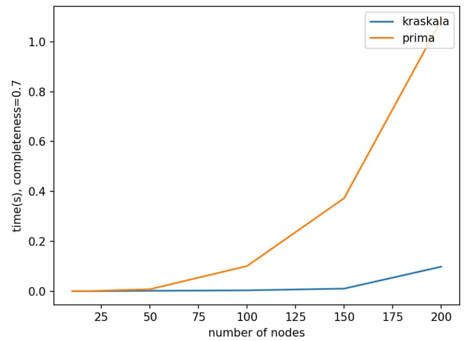
```python
def get_min(graph, nodes_set):
    """
    the function gets the minimum
    possible weighted edge
    """
    for node in nodes_set:
        current_example = min(graph, \
            key=lambda x: x[2] if \
            (x[0] not in nodes_set or x[1] not in nodes_set) and\
            (x[0] == node or x[1] == node) else
            math.inf)
        if current_example[2] != math.inf :
            return current_example
    return (0,0, math.inf)
```

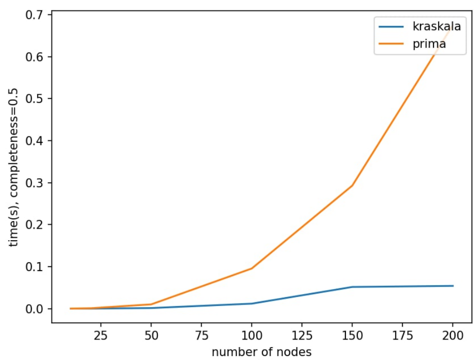
```python
def prim(graph, nodes_number):
    """
    creates the base of the graph by prim's algorythm
    current Nodes_set is the set that must be full of nodes
    then base is the list of edges in MST

    for future computing function sets inf for the weight
    of the first point to avoid weight compting problem and
    edge choosing problem

    return: base
    """
    Nodes_set = {0} #starting node: 0
    base = []
    graph[0][2] = math.inf
    while len(Nodes_set) < nodes_number:
        edge = get_min(graph, Nodes_set)
        if edge[2] == math.inf: #no minimum weight possible
            break
        base.append(edge)       #adding edge to the base graph
        Nodes_set.add(edge[0])  #addind nodes to the set of used nodes
        Nodes_set.add(edge[1])
    return base
```

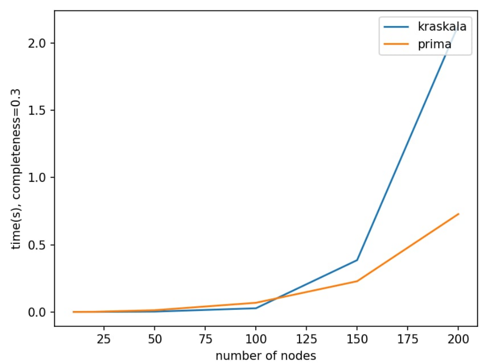
Experiments:

```python
def visual(completeness):
    """
    vunction creates dictionaries for
    timing of algorythms for nodes in the list
    """
    list2 = [10, 20, 50, 100,150, 200]
    visual_kruskal={} #dicts {nodes_number:time_calculating}
    visual_prim={}

    for nodes in list2:
        graph, nodes = create_graph(nodes,completeness)
        start = timeit.default_timer()
        kruskal(graph)
        stop = timeit.default_timer()
        visual_kruskal[nodes] = stop - start

    for nodes in list2:
        graph, nodes = create_graph(nodes,completeness)
        graph[0][2] = math.inf
        start = timeit.default_timer()
        prim(graph, nodes)
        stop = timeit.default_timer()
        visual_prim[nodes] = stop - start

    print(visual_prim, '\n', visual_kruskal)

    x=list(visual_kruskal.keys())
    x1=list(visual_prim.keys())
    fig, ax = plt.subplots()
    l1, = ax.plot(x, [visual_kruskal[i] for i in x], label='kraskala')
    l2, = ax.plot(x1, [visual_prim[i] for i in x1], label = 'prima')
    ax.legend(handles=[l1, l2], loc='upper right')
    plt.ylabel(f'time(s), completeness={completeness}')
    plt.xlabel('number of nodes')
    plt.show()
```









Підсумки до експериментів та їх порівняння:

Загалом алгоритм Прима виявився менш ефективним при його реалізації у вигляді програмного коду.

Це було очікувано, бо складність алгоритму Прима - O(V^2), коли складність Краскала - O(logV) - другий працює більш “жадно”, вибираючи вигідніші ребра на кожному кроці, хоч довжина кінцевого результату однакова.

Зі зниженням параметру completeness час виконання алгоритму Краскала збільшувався для відповідної кількості вершин, а час виконання алгоритму Прима зменшувався.

При completeness=0.3(4 графік) алгоритм Краскала навіть став менш ефективним за Прима.