<table>
<tr><td colspan="4">Piotr Mikołajczyk</td></tr>
<tr><td rowspan="2">AISDE
LAB 5</td><td>272018
Nr Indeksu</td><td colspan="2">2021.05.23 – 2021.05.29
Data wykonania ćwiczenia</td><td>2021.05.29
Nominalna data oddania sprawozdania</td></tr>
<tr><td>Punkty do wykonania</td><td colspan="3">Wykonujemy zadanie 4 z instrukcji, tzn. badamy algorytmy budowy minimalnego drzewa rozpinającego grafu nieskierowanego. Następnie, na podstawie analizy kodu i wiadomości z materiałów wykładowych, próbujemy określić złożoność obliczeniową zbadanych algorytmów. Można także podjąć próbę usprawnienia algorytmów. Przed zbadaniem wydajności własnej wersji należy przetestować jej poprawność a wyniki tych testów zamieścić w sprawozdaniu. Zamieszczamy w nim także zmodyfikowany kod.</td></tr>
</table>

SPRAWOZDANIE:

Algorytmy grafowe

W celu zbadania złożoności obliczeniowej algorytmów, wygenerowano grafy nieskierowane o wierzchołkach i krawędziach przyrastających kwadratowo w celu uzyskania dużej liczby wierzchołków dla stosunkowo nie dużej ilości punktów. Ze względu na problemy związane z uruchomieniem algorytmów Boruvki i Kruksala MST na maszynie wirtualnej z Linuksem, zdecydowano się przebadać wskazane algorytmy w środowisky Python. Poniżej algorytmy Boruvki i Kruskala oraz kod którym przebadano dane implementacje algorytmów.

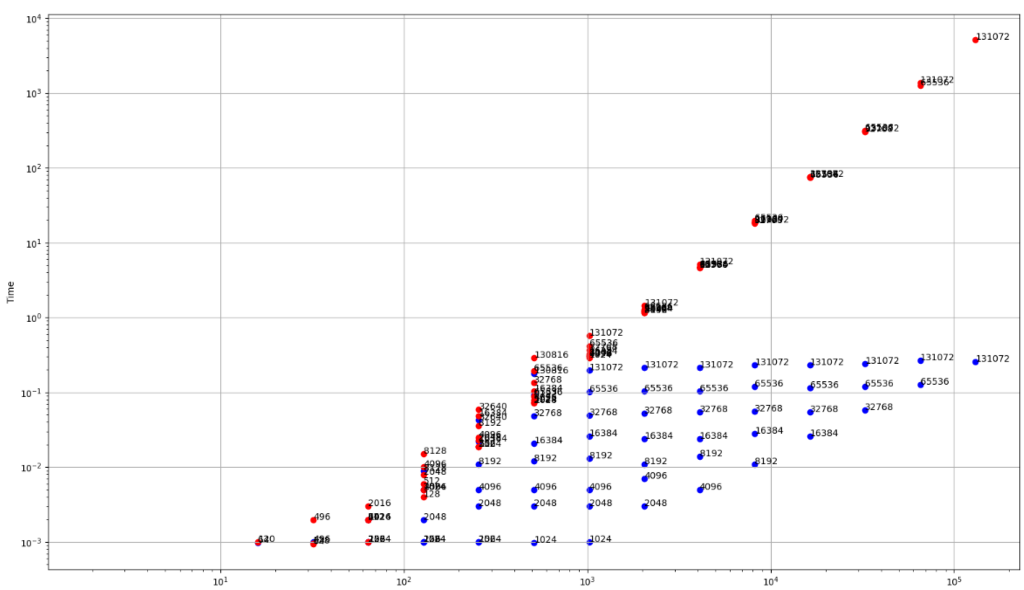
```
# nr_wierzchołka_1 <sp> nr_wierzchołka_2 <sp> waga_krawędzi
import time
import os
from collections import Counter
import matplotlib.pyplot as plt
import numpy as np
from nums_from_string import nums_from_string


class GraphB:
    def __init__(self, num_of_nodes):
        self.m_v = num_of_nodes
        self.m_edges = []
        self.m_component = {}

    def add_edge(self, u, v, weight):
        self.m_edges.append([u, v, weight])

    def find_component(self, u):
        if self.m_component[u] == u:
            return u
        return self.find_component(self.m_component[u])

    def set_component(self, u):
        if self.m_component[u] == u:
            return
        else:
            for k in self.m_component.keys():
                self.m_component[k] = self.find_component(k)
```

```python
    def union(self, component_size, u, v):
        if component_size[u] <= component_size[v]:
            self.m_component[u] = v
            component_size[v] += component_size[u]
            self.set_component(u)

        elif component_size[u] >= component_size[v]:
            self.m_component[v] = self.find_component(u)
            component_size[u] += component_size[v]
            self.set_component(v)

        # print(self.m_component)

    def boruvka(self):
        component_size = []
        mst_weight = 0

        minimum_weight_edge = [-1] * self.m_v

        for node in range(self.m_v):
            self.m_component.update({node: node})
            component_size.append(1)

        num_of_components = self.m_v

        print("---------Forming MST------------")
        while num_of_components > 1:
            for i in range(len(self.m_edges)):

                u = self.m_edges[i][0]
                v = self.m_edges[i][1]
                w = self.m_edges[i][2]

                u_component = self.m_component[u]
                v_component = self.m_component[v]

                if u_component != v_component:
                    if minimum_weight_edge[u_component] == -1 or \
                            minimum_weight_edge[u_component][2] > w:
                        minimum_weight_edge[u_component] = [u, v, w]
                    if minimum_weight_edge[v_component] == -1 or \
                            minimum_weight_edge[v_component][2] > w:
                        minimum_weight_edge[v_component] = [u, v, w]

            for node in range(self.m_v):
                if minimum_weight_edge[node] != -1:
                    u = minimum_weight_edge[node][0]
                    v = minimum_weight_edge[node][1]
                    w = minimum_weight_edge[node][2]

                    u_component = self.m_component[u]
                    v_component = self.m_component[v]

                    if u_component != v_component:
                        mst_weight += w
                        self.union(component_size, u_component, v_component)
                        # print("Added edge [" + str(u) + " - "
                        #       + str(v) + "]\n"
                        #       + "Added weight: " + str(w) + "\n")
                        num_of_components -= 1

            minimum_weight_edge = [-1] * self.m_v
        print("----------------------------------")
        print("The total weight of the minimal spanning tree is: " + str(mst_weight))


class Edge:
    def __init__(self, arg_src, arg_dst, arg_weight):
        self.src = arg_src
        self.dst = arg_dst
```

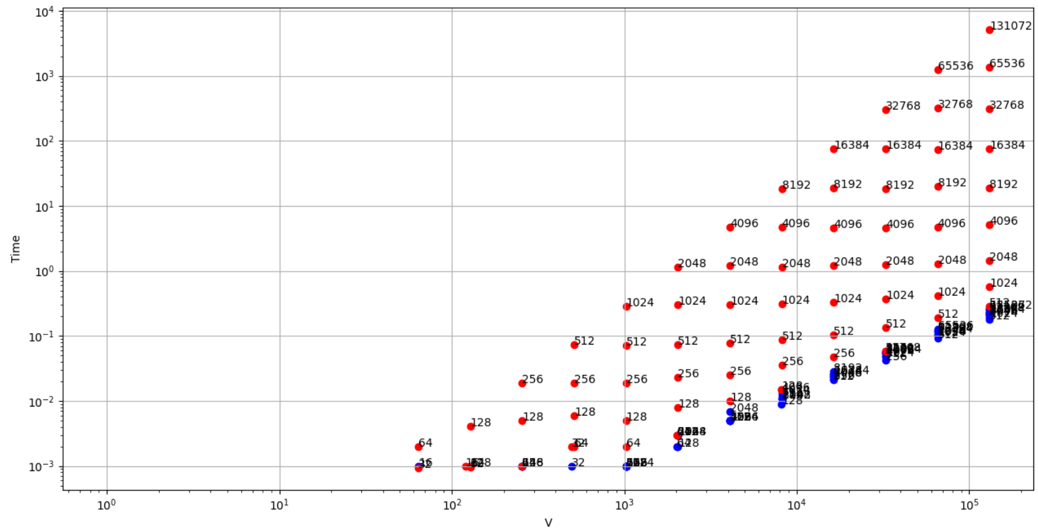
```python
        self.weight = arg_weight


class GraphK:

    def __init__(self, arg_num_nodes, arg_edgelist):
        self.num_nodes = arg_num_nodes
        self.edgelist = arg_edgelist
        self.parent = []
        self.rank = []
        self.mst = []

    def FindParent(self, node):

        if self.parent[node] == node:
            return node
        return self.FindParent(self.parent[node])

    def KruskalMST(self):

        self.edgelist.sort(key=lambda Edge: Edge.weight)

        self.parent = [None] * self.num_nodes
        self.rank = [None] * self.num_nodes

        for n in range(self.num_nodes):
            self.parent[n] = n
            self.rank[n] = 0

        for edge in self.edgelist:
            root1 = self.FindParent(edge.src)
            root2 = self.FindParent(edge.dst)
            if root1 != root2:
                self.mst.append(edge)
                if self.rank[root1] < self.rank[root2]:
                    self.parent[root1] = root2
                    self.rank[root2] += 1
                else:
                    self.parent[root2] = root1
                    self.rank[root1] += 1

        # print("\nEdges of minimum spanning tree in graph :", end=' ')
        cost = 0
        for edge in self.mst:
            # print("[" + str(edge.src) + "-" + str(edge.dst) + "](" + str(edge.weight) + ")",
end=' ')
            cost += edge.weight
        print("\nCost of minimum spanning tree : " + str(cost))


TimeKruskalMST = []
TimeBoruvkaMST = []
graphList = []
cur_path = os.path.dirname(__file__)

directory = '../'
names = []

for filename in os.listdir(directory):
    if filename.endswith(".txt"):
        names.append(filename)
        continue
    else:
        continue

s = []
i = 0
for k in names:
    s.append(nums_from_string.get_nums(names[i]))
    i += 1
```

```python
i = 0
# len(s)
V = []
E = []
for items in range(0, len(s)):
    # conv_i = f'{pow(2, i)}'
    # conv_j = f'{pow(2, j)}'
    fileName = '..\Graf' + str(s[i][0]) + 'V' + str(s[i][1]) + 'E.txt'
    V.append(s[i][0])
    E.append(s[i][1])
    f = open(fileName)
    print(fileName)
    rawGraph = f.read().split(" ")
    a = 0
    for x in rawGraph:
        if "\n" in rawGraph[a]:
            temp = rawGraph[a].split("\n")
            rawGraph[a] = x.replace(rawGraph[a], temp[0])
            rawGraph.insert(a + 1, temp[1])
        a += 1
    rawGraph = rawGraph[:-1]
    b = 0
    for x in rawGraph:
        rawGraph[b] = x.replace("\n", "")
        b += 1
    c = 0
    for c in range(0, len(rawGraph)):
        rawGraph[c] = int(rawGraph[c])
    i += 1

    graphList = rawGraph

    u = graphList[0::3]
    v = graphList[1::3]
    w = graphList[2::3]

    sList = u + v

    l1 = []
    count = 0
    for item in sList:
        if item not in l1:
            count += 1
            l1.append(item)

    e = []
    f = 0
    gb = GraphB(count)
    for x in u:
        gb.add_edge(u[f], v[f], w[f])
        e.append(Edge(u[f], v[f], w[f]))
        f += 1

    gk = GraphK(count, e)

    startk = time.time_ns() / (10 ** 9)
    gk.KruskalMST()
    endk = time.time_ns() / (10 ** 9)
    TimeKruskalMST.append(endk - startk)

    startb = time.time_ns() / (10 ** 9)
    gb.boruvka()
    endb = time.time_ns() / (10 ** 9)
    TimeBoruvkaMST.append(endb - startb)
    print(f'time KRUSKAL:', TimeKruskalMST)
    print(f'time BORUVKA:', TimeBoruvkaMST)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

xs = V
```

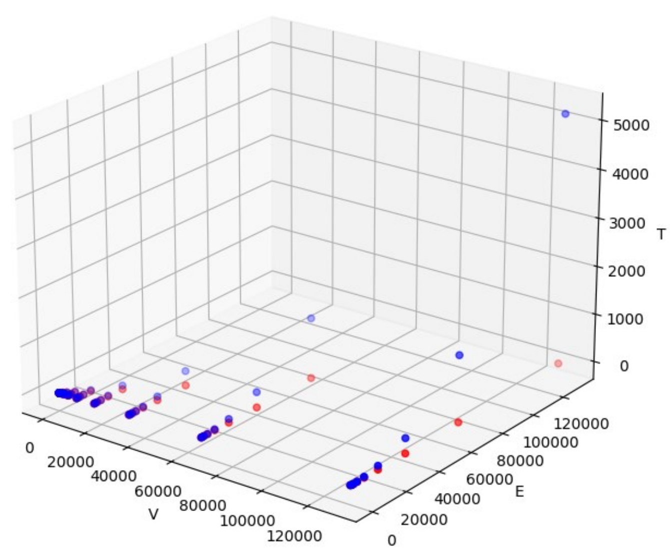
```
ys = E
zt1 = TimeKruskalMST
zt2 = TimeBoruvkaMST
print(xs)
print(ys)
ax.scatter(xs, ys, zt1, c='r')
ax.scatter(xs, ys, zt2, c='b')

ax.set_xlabel('V')
ax.set_ylabel('E')
ax.set_zlabel('T')

fig1, ay = plt.subplots()
ay.scatter(xs, zt1, c='b')
ay.scatter(xs, zt2, c='r')
for i, txt in enumerate(ys):
    ay.annotate(txt, (xs[i], zt1[i]))
for i, txt in enumerate(ys):
    ay.annotate(txt, (xs[i], zt2[i]))
ay.grid()
ay.set_xlabel('V')
ay.set_ylabel('Time')
ay.set_yscale('log')
ay.set_xscale('log')

fig2, az = plt.subplots()
az.scatter(ys, zt1, c='b')
az.scatter(ys, zt2, c='r')
for i, txt in enumerate(xs):
    az.annotate(txt, (ys[i], zt1[i]))
for i, txt in enumerate(xs):
    az.annotate(txt, (ys[i], zt2[i]))
az.grid()
az.set_xlabel('E')
az.set_ylabel('Time')
az.set_yscale('log')
az.set_xscale('log')
plt.show()
```



Rys. 1 – Złożoności czasowe algorytmów Boruvki (czerwony) i Kruskala (niebieski) dla rosnących krawędzi oraz różnych wierzchołków. Wykres loglog.



Rys. 2 – Złożoności czasowe algorytmów Boruvki (czerwony) i Kruskala (niebieski) dla rosnących wierzchołków oraz różnej liczby krawędzi. Wykres loglog.



Rys. 3 – Wykres 3d złożoności czasowej algorytmów Boruvki (niebieski) i Kruskala (czerwony) dla rosnących wierzchołków oraz różnej liczby krawędzi. Wykres liniowy.

Wnioski:

Na rysunku 1 oraz 2 zaznaczono przy każdym punkcie krawędzie dla wierzchołków na osi x (rys. 2) oraz wierzchołki dla krawędzi na osi x (rys. 1). Widać że algorytm Boruvki w zależności od ilości krawędzi zmienia się z zależnością kwadratową (silna zależność) dla krawędzi oraz co najmniej liniową dla rosnących wierzchołków (słaba zależność).Algorytm Kruskala natomiast posiada liniowo logarytmiczno liniową złożoność (nlogn) dla rosnących wierzchołków (silna zależność) natomiast dla rosnących krawędzi posiada złożoność w przybliżeniu liniową (słaba zależność). Wynika z tego że dla grafów z małym przyrostem krawędzi a dużym wierzchołków, można posłużyć się w celu utworzenia MST, algorytmem Boruvki ponieważ zbiega on do alg. Kruskala. Jednak w zestawieniu punktów całościowym, algorytm Boruvki wypadł znacznie gorzej niż Kruskala – punkty złożoności obliczeniowej algorytmu Kruskala są na wykresach dolną granicą algorytmu Boruvki. W praktyce oznacza to że nie opłacałoby się wykorzystywać algorytm Boruvki w porównaniu do Kruskala w zaprezentowanej implementacji. Rys. 3 pokazuje wykres 3d zależności między ilością wierzchołków i krawędzi oraz czasem obliczeniowym. Niestety nie jest wystarczająco czytelny jednak wywnioskować można z niego że algorytm Boruvki dla dużych zbiorów danych uzyskuje znacznie większe czasy niż algorytm Kruskala.

Oświadczam, że niniejsza praca stanowiąca podstawę do uznania osiągnięcia efektów uczenia się z przedmiotu Algorytmy i Struktury Danych została wykonana przez mnie samodzielnie.