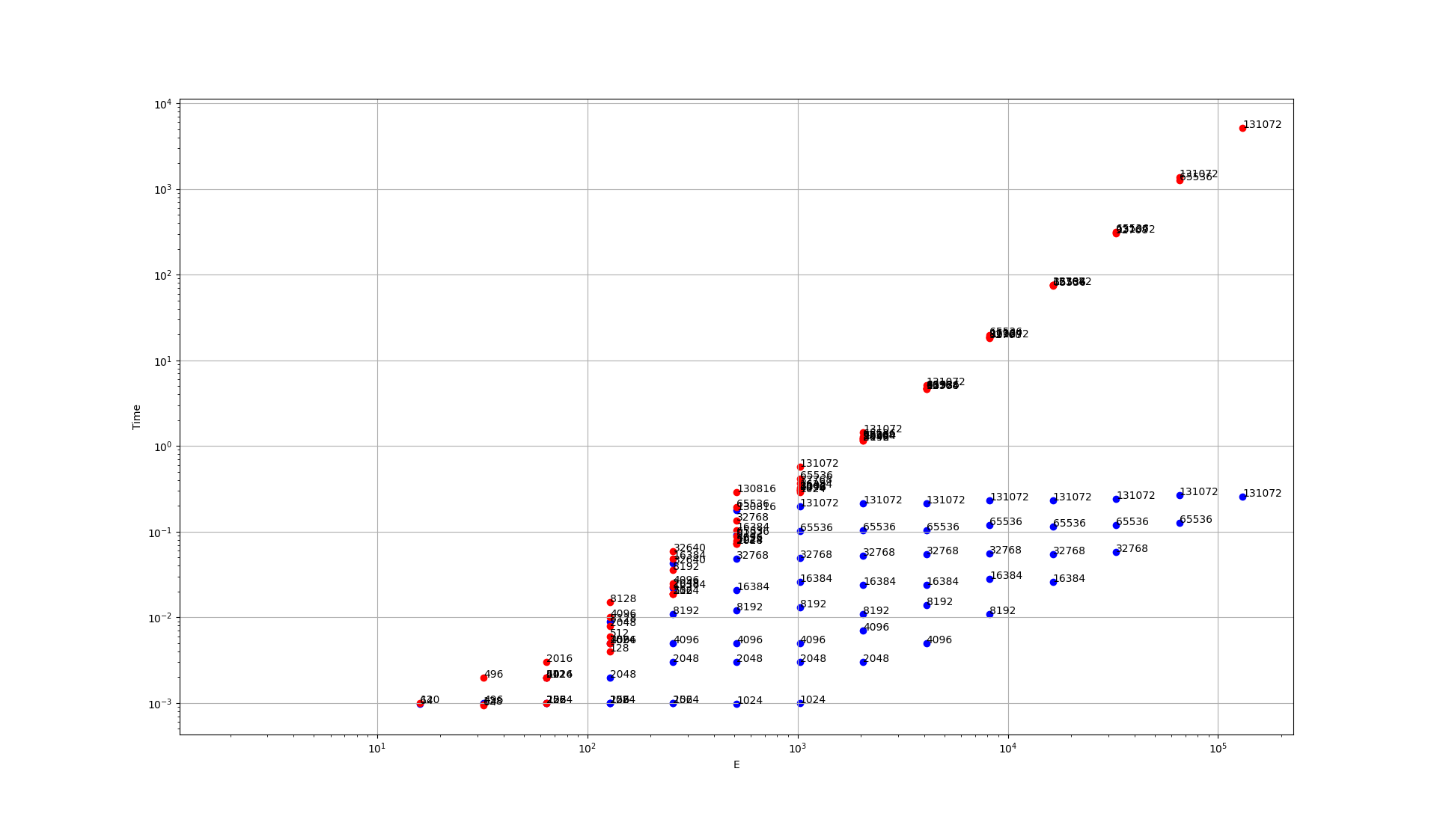
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Piotr Mikołajczyk | | | | |
| **AISDE**  **LAB 5** | 272018  Nr Indeksu | | 2021.05.23 – 2021.05.29  Data wykonania ćwiczenia | 2021.05.29  Nominalna data oddania sprawozdania |
| Punkty do wykonania | Wykonujemy zadanie 4 z instrukcji, tzn. badamy algorytmy 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. | | |

SPRAWOZDANIE:

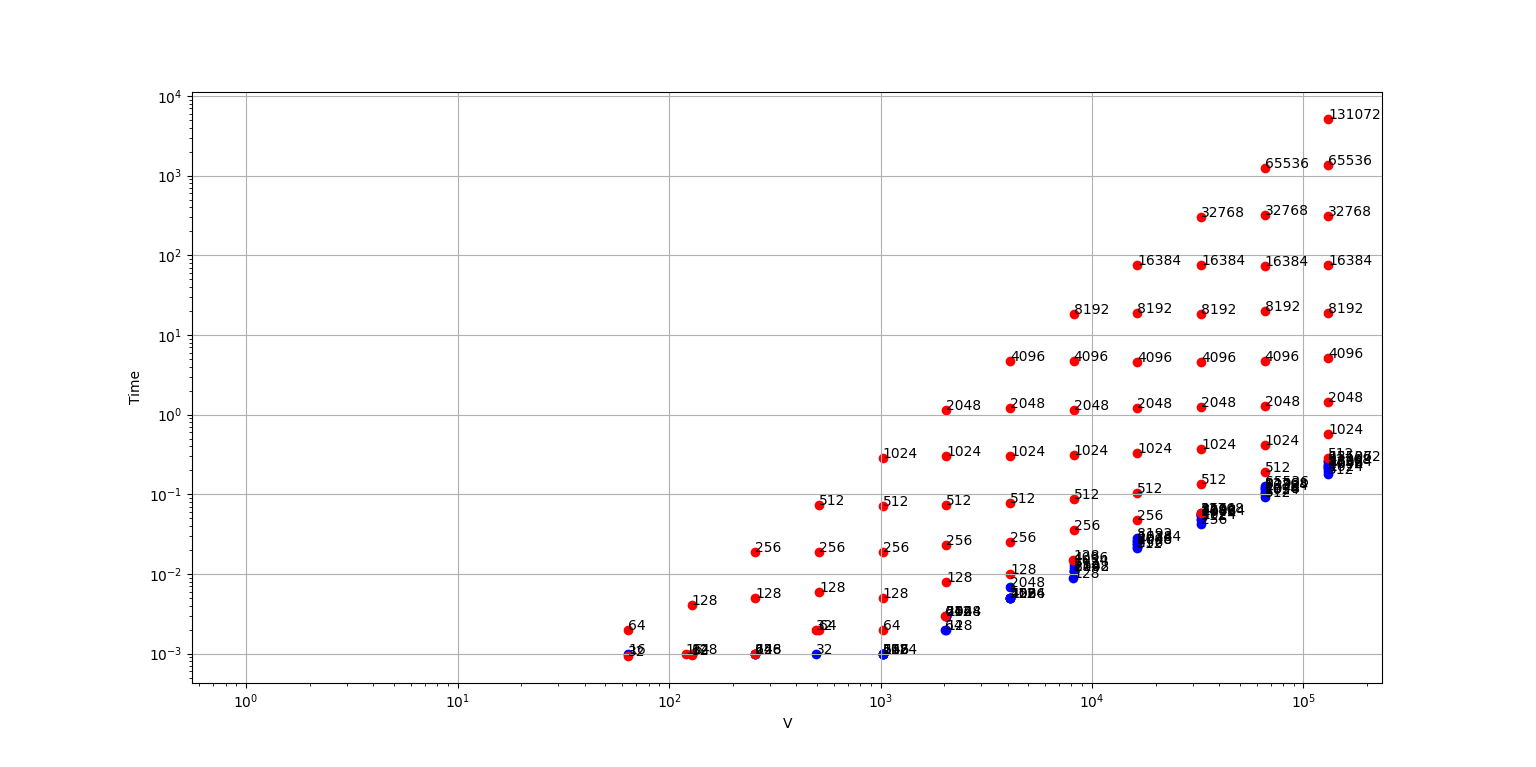
Algorytmy grafowe

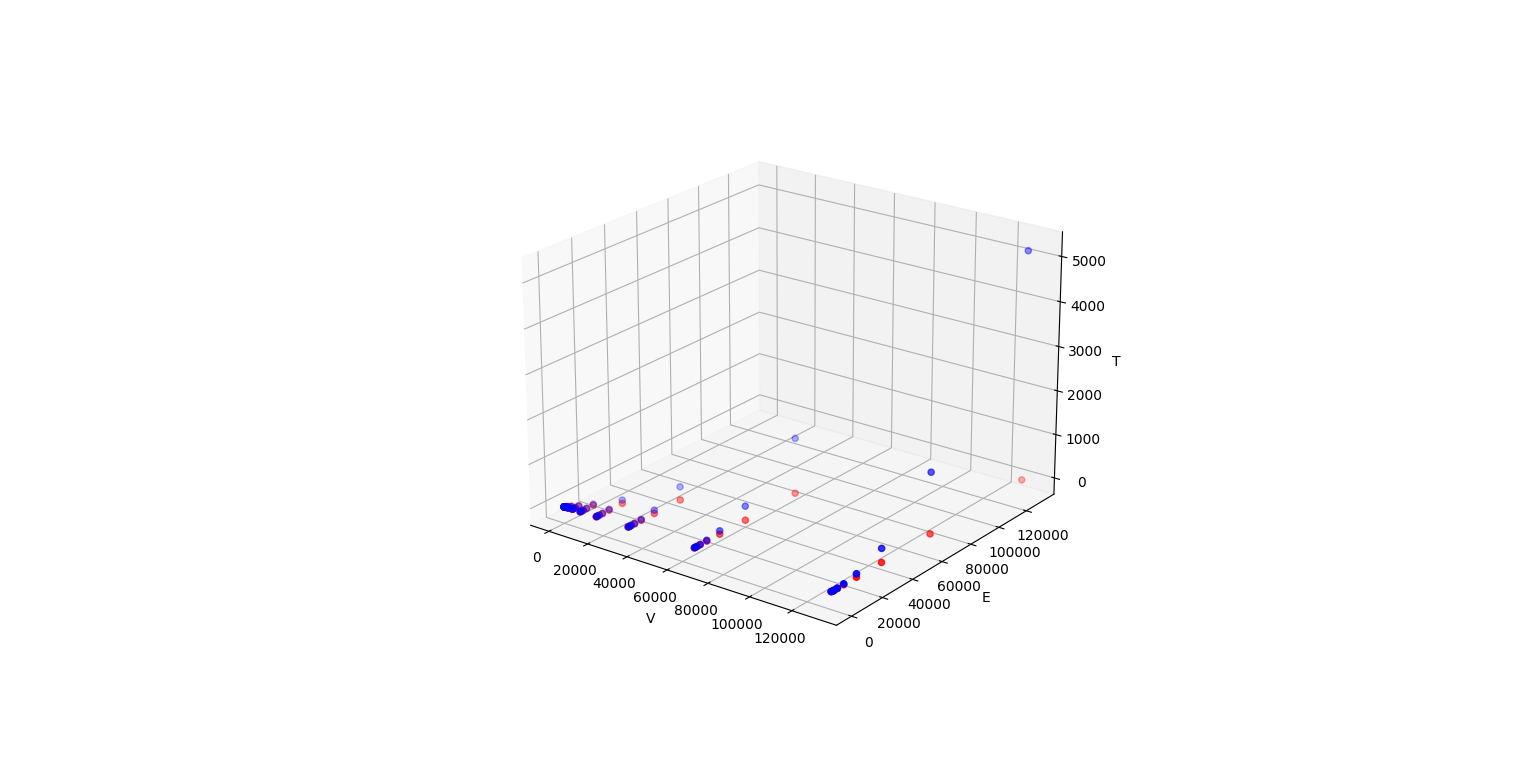
W celu zbadania złożoności obliczeniowej algorytmów, wygenerowano grafy nieskierowane o wierzchołkach i krawędziach przyrastających kwadratowo w celu uzyskania dużej liczby wierzchołków dla stosunkowo nie dużej ilości punktów. Ze względu na problemy związane z uruchomieniem algorytmów Boruvki i Kruksala 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)  
  
 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  
 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  
  
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.*