1. Книшоїд Софія
   Фольварочна Софія

2. Наша задача полягала в тому, щоб порівняти алгоритми Краскала та Прима. Для початку ми використали функцію gnp_random_connected_graph, щоб сформувати граф, а потім алгоритми, які будуть представлені нижче. Також щоб провести експеримент ми вирішили провести два порівняння. Перший раз ми вирішили порівняти швидкість виконання алгоритму з бібліотеки networkx, а другий раз наших алгоритмів.
3. Специфікація комп'ютера:
   - Кількість ядер: 6
   - Тактова частота: 2.38 GHz
   - Установлена фізична пам'ять: 8 ГБ(доступно для використання: 7,42 ГБ)
   - ОС: Майкрософт Windows 10 Pro
4. Програмний код алгоритмів
   Для алгоритму Прима для зручності запис даних перетворено в матрицю:

```
def turn_into_matrix(number, compl):

list_of_edges=list(gnp_random_connected_graph(number,compl).edges.data('we
ight'))
    adjacencyMatrix = [[0 for column in range(number)] for row in
range(number)]
    for i,j,k in list_of_edges:
        adjacencyMatrix[i][j]=k
        adjacencyMatrix[j][i]=k
    return adjacencyMatrix
```

```
def prim(adjmatrix):
    selected_node=[False]*len(adjmatrix)# we create a list of selected
nodes

    total=0
    counter=0#count nodes until we pick all of them
    selected_node[0]=True # we pick the first node
```

```
    while counter<len(adjmatrix)-1:#we create cycle which will go through
all the vertices tied to the selected until you find the min weight
        min=999999999 #we create a big number to compare with weight
        u=0
        v=0
        for i in range(len(adjmatrix)):
            if selected_node[i]:#pick selected node
                for j in range(len(adjmatrix)):
                    if ((not selected_node[j]) and adjmatrix[i][j]):
                        #pick not selected vertex, which creates an edge
with our vertex
                        if min > adjmatrix[i][j]:
                            min = adjmatrix[i][j]#we assign the value of
the least weight and compare everything else with it
                            u = i
                            v = j
        total+=adjmatrix[u][v]#add min weight of edge if it exists, else 0
        selected_node[v] = True#assign another vertex value True, so it is
picked
        counter+=1#add 1, as we pick one more node
    return total
```

Алгоритм Краскала:

```
def kruskal_alg(edges_ls:list):
    """
    the Kruskal algorithm
    """

    # since we take G = gnp_random_connected_graph(..,..,draw=True),
    # take edges_ls as G.edges(data=True)
    # edges_ls looks like: [(11, 12, {'weight': 7}), ...]
    # step 1: sorting the edges by weight
    sorted_edg_ls = sorted(edges_ls, key=lambda x: x[2]['weight'])

    # creating containers of joined nodes,
    # isolated, independent nodes groups
    # and final result edges of Kruskal algorithm
    joined, group, result_edges = set(), dict(), list()
```

```
    one_group = True # assume everything is in one connected group as
nodes are connected
    for edge in sorted_edg_ls:
        first_node = edge[0]
        second_node = edge[1]
        # join all the nodes to some joined group; in result, we get
several independent structures of connected nodes
        if first_node not in joined or second_node not in joined:  # if
both are already connected, there is a risk of creating a cycle
            if first_node not in joined and second_node not in joined:
                group[first_node] = group[second_node] = [first_node,
second_node]  # add both nodes to group dictionary
            else:
                if first_node not in group.keys():             # first
node is not in a group
                    group[second_node].append(first_node)        # put it
in the group with the second node
                    group[first_node] = group[second_node]
                else:
                    group[first_node].append(second_node)     # second
node is not in the group, put it in the group with node 1
                    group[second_node] = group[first_node]
            # since we have connected two nodes appropriately, add this
pair to the resulting graph
            result_edges.append(edge)
            joined.add(first_node)
            joined.add(second_node)

    # now join the groups of nodes we got
    possible_connection_ls = []
    for edge in sorted_edg_ls:
        first_node = edge[0]
        second_node = edge[1]
        if second_node not in group[first_node]:     # nodes are in
different groups
            one_group = False
            possible_connection_ls.append(edge)
    if one_group is False:
```

```
        final_join = sorted(possible_connection_ls, key=lambda x:
x[2]['weight'])
        result_edges.append(final_join[0])
    min_sum = 0
    for elem in result_edges: # count sum of the resulting graph
        min_sum += elem[2]['weight']
    return min_sum
```

5. Програмний код експерименту:
    - Для алгоритму з networkx

```
def counting_time(nodes, NUM_OF_ITERATIONS, alg):
    time_taken = 0
    for i in tqdm(range(NUM_OF_ITERATIONS)):

        # note that we should not measure time of graph creation
        G = gnp_random_connected_graph(nodes, 0.1, False)

        start = time.time()
        tree.minimum_spanning_tree(G, algorithm=alg)
        end = time.time()

        time_taken += end - start
    return time_taken

def draw_graph(dct1,dct2):
    row_1=list(dct1.keys())
    row_2=list(dct1.values())
    plt.plot(row_1, row_2, 'r', marker='o', label="Алгоритм Прима")
    row_1=list(dct2.keys())
    row_2=list(dct2.values())
    plt.plot(row_1, row_2, 'b', marker='o', label='Алгоритм Краскала')
    plt.title('Часова ефективність алгоритмів')
    plt.xlabel('Розмірність графу')
    plt.ylabel('Час роботи')
    plt.legend()

dict_of_results={}
dict_of_results[10]=counting_time(10,1000,'prim')
dict_of_results[20]=counting_time(20,1000,'prim')
```

```
dict_of_results[50]=counting_time(50,1000,'prim')
dict_of_results[100]=counting_time(100,1000,'prim')
dict_of_results[200]=counting_time(200,1000,'prim')
dict_of_results[250]=counting_time(250,1000,'prim')
dict_of_results[500]=counting_time(500,1000,'prim')
dict_of_results[1000]=counting_time(1000,1000,'prim')
second={}
second[10]=counting_time(10,1000,'kruskal')
second[20]=counting_time(20,1000,'kruskal')
second[50]=counting_time(50,1000,'kruskal')
second[100]=counting_time(100,1000,'kruskal')
second[200]=counting_time(200,1000,'kruskal')
second[250]=counting_time(250,1000,'kruskal')
second[500]=counting_time(500,1000,'kruskal')
second[1000]=counting_time(1000,1000,'kruskal')
draw_graph(dict_of_results,second)
```

- Для наших алгоритмів

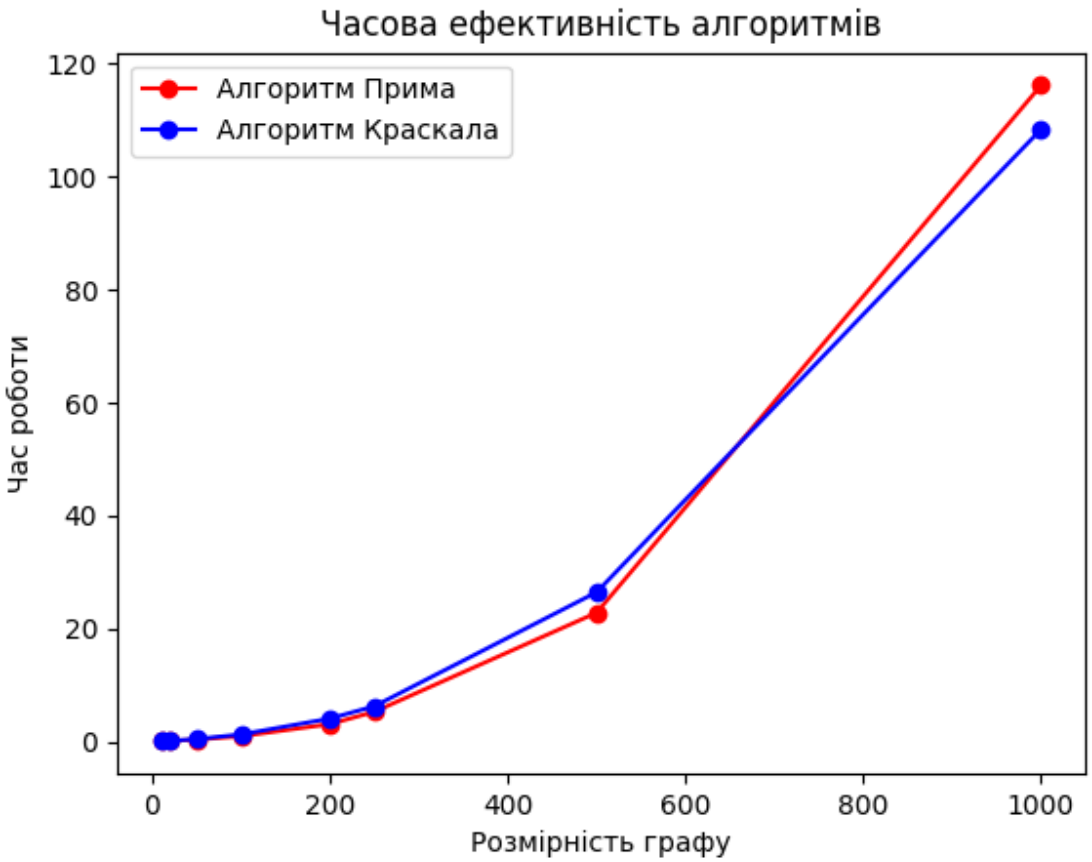
```
def counting_time(nodes, NUM_OF_ITERATIONS):
    time_taken = 0
    for i in tqdm(range(NUM_OF_ITERATIONS)):

        G = turn_into_matrix(nodes,0.1)

        start = time.time()
        prim(G)
        end = time.time()

        time_taken += end - start
    return time_taken
def counting_time_2(nodes,NUM_OF_ITERATIONS):
    time_taken = 0
    for i in tqdm(range(NUM_OF_ITERATIONS)):

        G=gnp_random_connected_graph(nodes,0.1)

        start = time.time()
        kruskal_alg(G.edges(data=True))
        end = time.time()
```

```
        time_taken += end - start
    return time_taken

def draw_graph(dct1,dct2):
    row_1=list(dct1.keys())
    row_2=list(dct1.values())
    plt.plot(row_1, row_2, 'r', marker='o', label="Алгоритм Прима")
    row_1=list(dct2.keys())
    row_2=list(dct2.values())
    plt.plot(row_1, row_2, 'b', marker='o', label='Алгоритм Краскала')
    plt.title('Часова ефективність алгоритмів')
    plt.xlabel('Розмірність графу')
    plt.ylabel('Час роботи')
    plt.legend()

    plt.show()


dict_of_results={}
dict_of_results[10]=counting_time(10,1000)
dict_of_results[20]=counting_time(20,1000)
dict_of_results[50]=counting_time(50,1000)
dict_of_results[100]=counting_time(100,1000)
dict_of_results[200]=counting_time(200,1000)
dict_of_results[250]=counting_time(250,1000)
dict_of_results[500]=counting_time(500,1000)

second={}
second[10]=counting_time_2(10,1000)
second[20]=counting_time_2(20,1000)
second[50]=counting_time_2(50,1000)
second[100]=counting_time_2(100,1000)
second[200]=counting_time_2(200,1000)
second[250]=counting_time_2(250,1000)
second[500]=counting_time_2(500,1000)

draw_graph(dict_of_results,second)
```

6. Це дані для алгоритму networkx при 1000 ітераціях:

Прима:
{10: 0.05402374267578125, 20: 0.10562992095947266, 50: 0.32158660888671875, 100: 0.9235556125640869, 200: 2.995626926422119, 250: 5.184518337249756, 500: 22.76223611831665, 1000: 115.60192441940308}

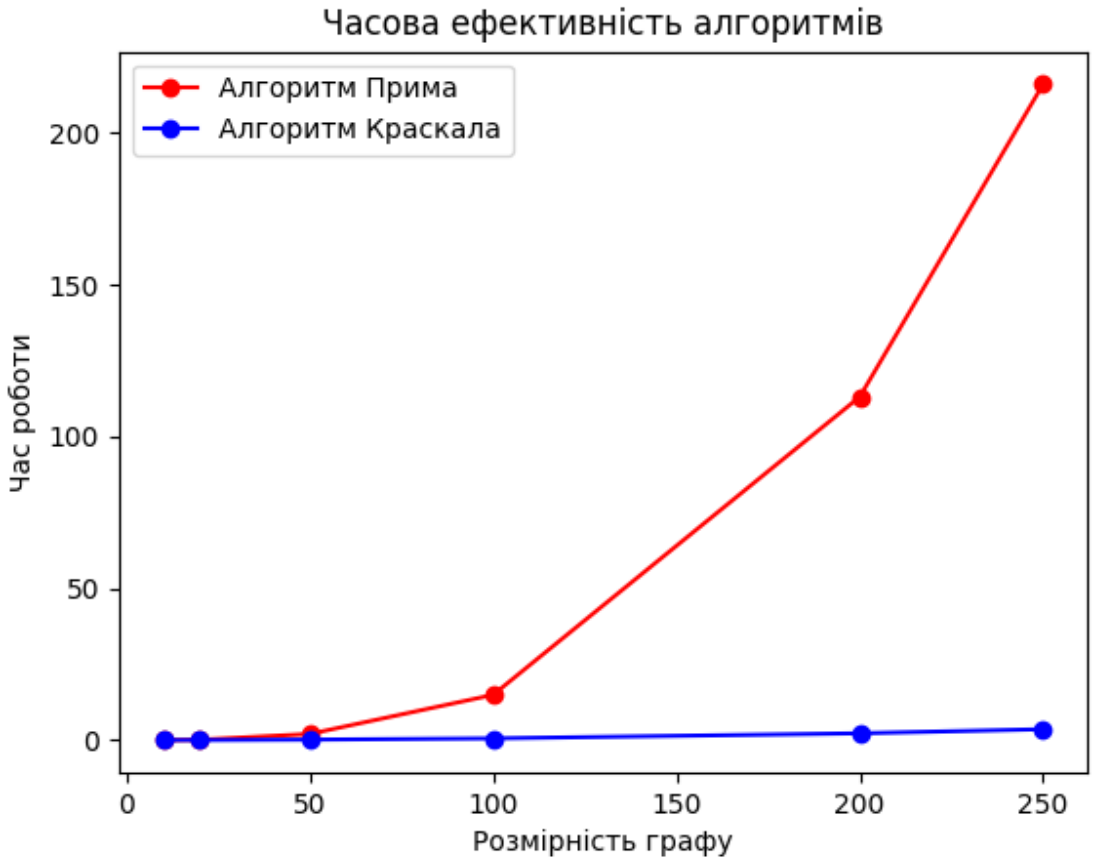
Краскала:
{10: 0.07300257682800293, 20: 0.12403130531311035, 50: 0.4360785484313965, 100: 1.2289187908172607, 200: 4.091622591018677, 250: 6.3779332637786865, 500: 25.425278425216675, 1000: 108.12357997894287}



На графіку ми можемо побачити, що спочатку різниця між часом виконання є мінімальною. При 500 вершинах Алгоритм Прима працює швидше, але вже при 1000 алгоритм Краскала виконується швидше.

Це дані для наших алгоритмів, але тут максимальна розмірність графа - 250 вершин, адже вони працюють набагато довше(особливо алгоритм Прима)

Прима:
{10: 0.026116609573364258, 20: 0.15837907791137695, 50: 2.0512471199035645, 100: 14.936888217926025, 200: 113.13581871986389, 250: 216.12461805343628}
Краскала:
{10: 0.026985645294189453, 20: 0.047014474868774414, 50: 0.18935418128967285, 100: 0.5981590747833252, 200: 2.2752010822296143, 250: 3.6132941246032715}



7. Загальний підсумок
   Як ми можемо побачити на графі при кількості вершин <600, варто використовувати алгоритм Прима, адже він є швидшим. Коли кількість вершин є більшою, варто використовувати алгоритм Краскала. Але якщо використовувати наші алгоритми, то Краскала працює набагато швидше. Отже, найкращим є алгоритм Краскала, адже якщо використовувати networkx, то спочатку різниця в часі є мінімальною, а потім при більшій кількості вершин Краскала працює набагато швидше. І якщо використовувати наші алгоритми, то вибір очевидний, адже Краскала в будь-якому випадку працює набагато швидше і навіть краще ніж алгоритм з networkx.

Примітки до другого завдання:

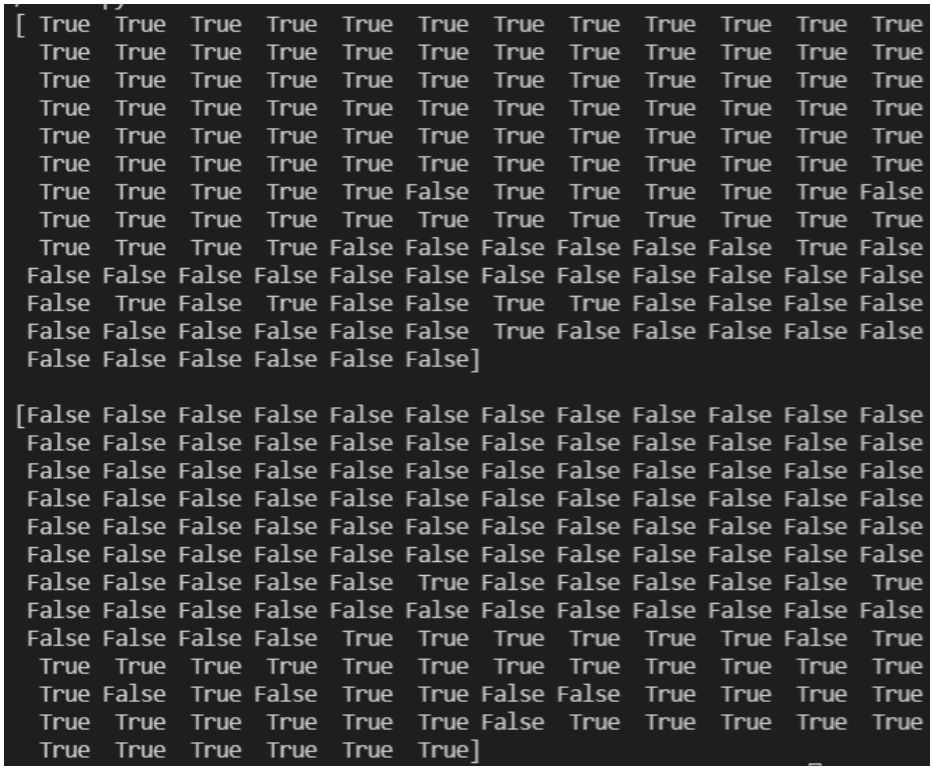
Використана техніка взаємодії з pandas dataframe (на прикладі iris.data):

```
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]]
```

1. iris.data[:10] - беремо набір перших 10 значень
2. iris.data[:10, 3] - беремо значення з третьої колонки

3. iris.data[:10, 3]<0.3
   - ділимо dataframe на 2 частини за булевим “центральним” значенням

4. Приклад розділу pandas dataframe з булевим значенням/ ~dataframe:



Таким чином було здійснено поділ pandas dataframe на дві групи (у контексті роботи - піддерева). Таке зручне представлення дозволило систематизувати обробку можливих поділів дерева рішень, після чого було обрано найбільш оптимальний варіант (критерієм відбору був gini impurity індекс).