**Прізвище:** Пархут

**Ім'я:** Адам

**Група:** КН-406

**Варіант:** 24

**Кафедра:** САПР

**Дисципліна:** Дискретні моделі в САПР

**Перевірив:** Кривий Р.З.

**GitHub:**

**ЗВІТ**

до лабораторної роботи №1

на тему "Побудова мінімального остового дерева"

**Мета роботи:** метою даної лабораторної роботи є вивчення алгоритмів рішення задач побудови остових дерев.

**Короткі теоретичні відомості:**

Графом G називають скінчену множину V з нерефлексивним симетричним відношенням R на V. Визначим E як множину симетричних пар в R. Кожний елемент V називають вершиною. Кожний елемент Е називають ребром, а E множиною ребер G.

Граф називається зв’язним, якщо в ньому для будь-якої пари вершин знайдеться ланцюг, який їх з’єднує, тобто, якщо по ребрах (дугах) можна попасти з будь-якої вершини в іншу.

Цикл - це ланцюг, в якого початкова і кінцева точки співпадають. Дерево - це зв’язний граф без циклів.

Покриваючим деревом графа називається любе дерево, що утворене сукупністю його ребер(дуг), які включають всі вершини графа.

Лісом називається будь-яка сукупність дуг (ребер) інцидентних до вершин, які не містять циклів. Таким чином, ліс складається з одного або більше дерев. Орієнтований ліс визначається як звичайний, тільки складається не з простих дерев, а орієнтованих.

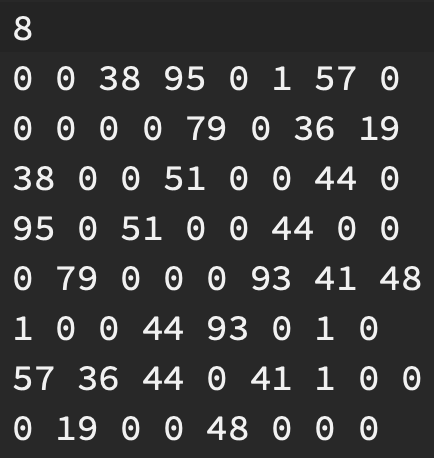
Згідно завдання потрібно виконати побудову за алгоритмом Борувки. Це алгоритм знаходження мінімального остового дерева в графі. Робота алгоритму складається з декількох ітерацій, кожна з яких полягає в послідовному додаванні ребер до остового лісу графа, до тих пір, поки ліс не перетвориться на дерево, тобто, ліс, що складається з однієї компоненти зв'язності.

У псевдокоді, алгоритм можна описати так:

* Спочатку, нехай T - порожня множина ребер (представляє собою остовий ліс, до якого кожна вершина входить в якості окремого дерева).
* Поки T не є деревом (поки число ребер у T менше, ніж V-1, де V - кількість вершин у графі):
  + Для кожної компоненти зв'язності (тобто, дерева в остовому лісі) в підпункті з ребрами T, знайдемо ребро найменшої ваги, що зв'язує цю компоненту з деякої іншої компонентою зв'язності. (Передбачається, що ваги ребер різні, або як-то додатково впорядковані так, щоб завжди можна було знайти єдине ребро з мінімальною вагою).
  + Додамо всі знайдені ребра в множину T.
  + Отримана множина ребер T є мінімальним остовим деревом вхідного графа.

**Індивідуальне завдання:**

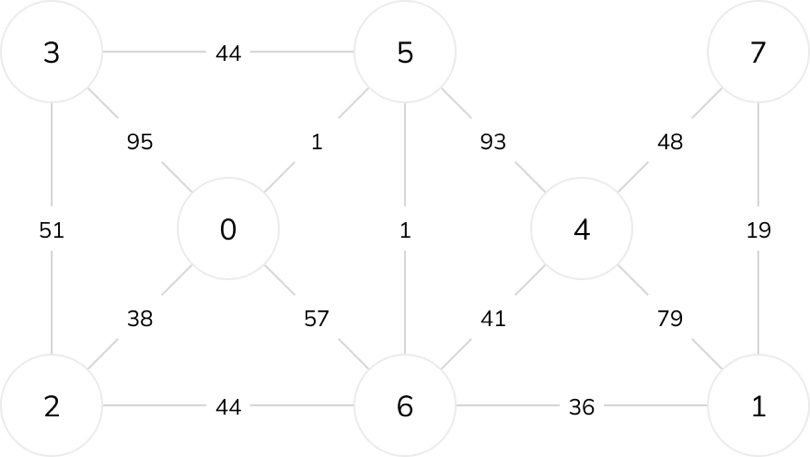
1. Отримати у викладача індивідуальне завдання.
2. Підготувати програму для вирішення виданого завдання.
3. Запустити на покрокове виконання програму побудови мінімального покриваючого дерева і максимального покриваючого дерева.
4. Здійснити перевірки роботи програм з результатами розрахунків проведених вручну.
5. Зафіксувати результати роботи.
6. Оформити і захистити звіт.



*Рис. 1. Матриця ваг*

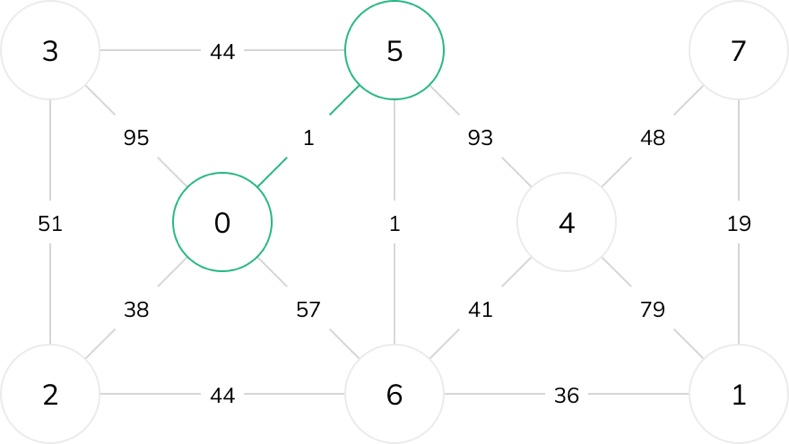
**Хід роботи:**

Спочатку побудовано візуалізований граф, на основі заданої матриці ваг.

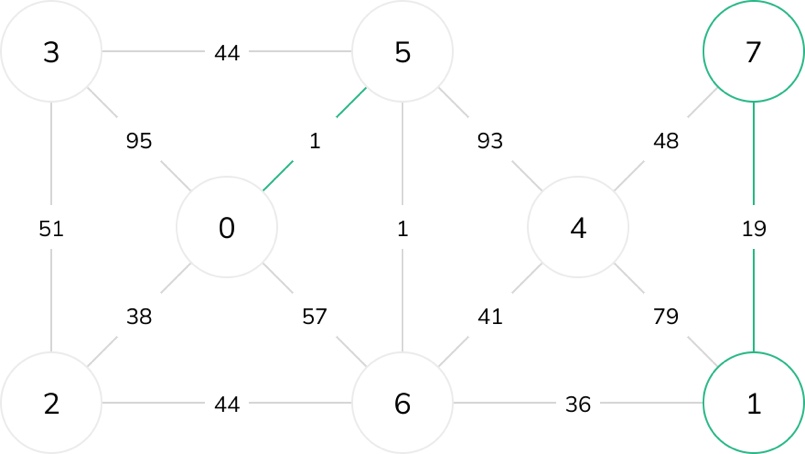


*Рис. 2. Візуалізований граф*

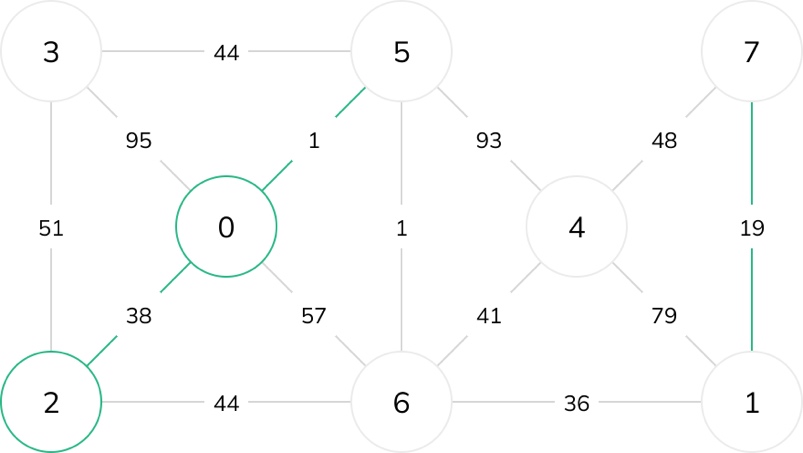
Далі у множину, записано вершину, кожне ребро, якого має найменшу вагу.



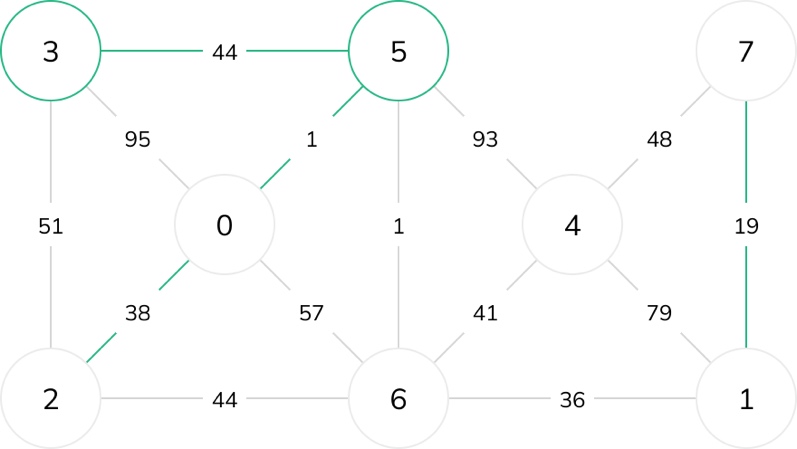
*Рис. 3. Вершини 0 та 5*



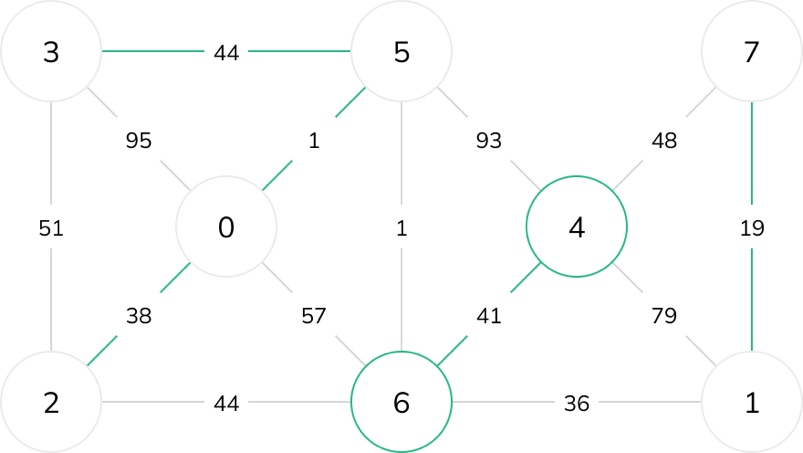
*Рис. 4. Вершини 1 та 7*



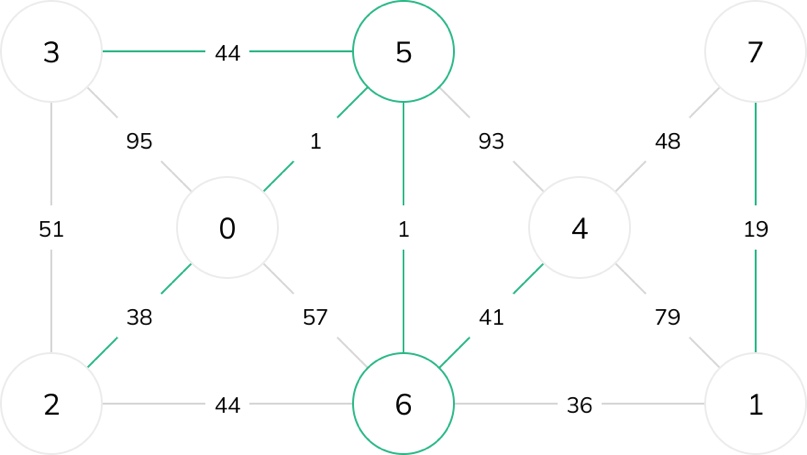
*Рис. 5. Вершини 2 та 0*



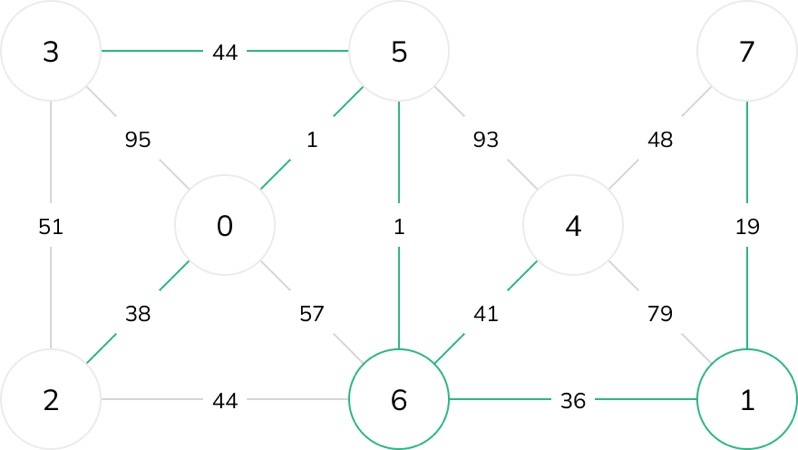
*Рис. 6. Вершини 3 та 5*



*Рис. 7. Вершини 4 та 6*



*Рис. 6. Вершини 5 та 6*



*Рис. 7. Вершини 6 та 1*

Закінчено ітерації, через те, що виконана умова, де написано, що виконувати потрібно до тих пір, поки число ребер у T менше, ніж V-1, де V - кількість вершин у графі.

Далі обчислено вагу дерева. Для цього потрібно додати вагу усіх позначених ребер.

**Фрагмент коду програми:**

def boruvka(self):

arr = np.empty((0, 3), int)

result = 0

selector = self.vertices

subsets = [None] \* (self.vertices)

cheapest = [None] \* (self.vertices)

v = 0

while (v < self.vertices):

subsets[v] = State(v, 0)

v += 1

while (selector > 1):

v = 0

while (v < self.vertices):

cheapest[v] = None

v += 1

k = 0

while (k < self.vertices):

i = 0

while (i < len(self.graphEdge[k])):

set1 = self.find(subsets, self.graphEdge[k][i].src)

set2 = self.find(subsets, self.graphEdge[k][i].dest)

if (set1 != set2):

if (cheapest[k] == None):

cheapest[k] = self.graphEdge[k][i]

elif (cheapest[k].weight > self.graphEdge[k][i].weight):

cheapest[k] = self.graphEdge[k][i]

i += 1

k += 1

i = 0

while (i < self.vertices):

if (cheapest[i] != None):

set1 = self.find(subsets, cheapest[i].src)

set2 = self.find(subsets, cheapest[i].dest)

if (set1 != set2):

selector -= 1

self.findUnion(subsets, set1, set2)

print("\n Включити ребро (", cheapest[i].src, " - ", cheapest[i].dest, ") вагою",

cheapest[i].weight, end="")

result += cheapest[i].weight

newRow = [cheapest[i].src,

cheapest[i].dest, cheapest[i].weight]

arr = np.vstack([arr, newRow])

i += 1

arrayOld = getFile()

arrayOld = [[int(j) if '.' not in j else float(j)

for j in i] for i in arrayOld]

numberOfNodes = arrayOld[0][0]

arrayNumbers = np.empty((0, numberOfNodes), int)

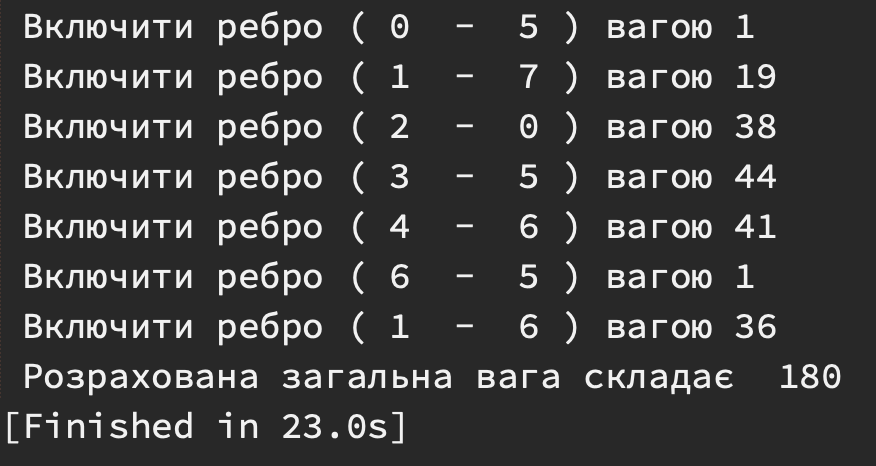
newRow = arrayNumbers[i[ for i in arrayOld[1:]:

arrayNumbers = np.vstack([arrayNumbers, i])

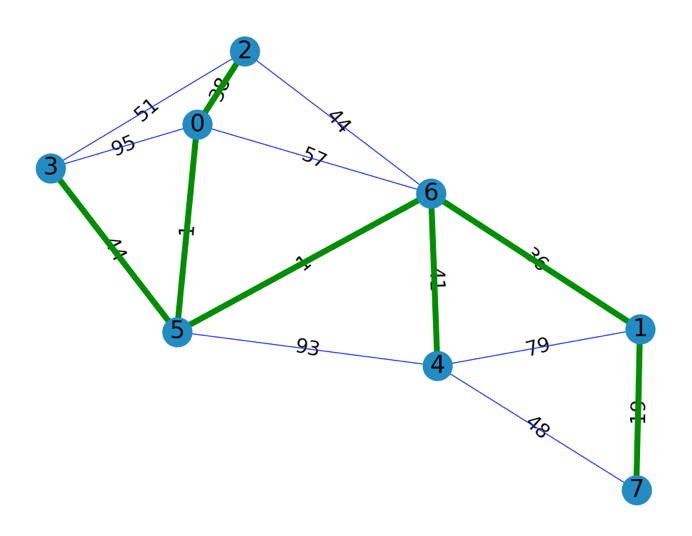
res = np.where(arrayNumbers != 0)

oneArray = np.column\_stack((res[0], res[1]))

**Результати виконання програми:**

****

*Рис. 8. Результат виконання програми*

****

*Рис. 9. Результат виконання програми*

**Висновок:**

Під час виконання лабораторної роботи я ознайомився з бінарними відношеннями. Для виконання цієї роботи було задана матриця розмірністю 5 на 5, та спочатку було опрацьована теоретично, залежно від того як вона має поводитись залежно від того, яку відношення ми перевіряємо відносно неї. Загалом були опрацюванні такі відношення як: рефлексивність, антирефлексивність, симетричність, антисиметричність, асиметричність, транзитивність. Програмний код був розроблений на мові програмування Python. В результаті ми отримали результати, котрі й очікували і вони збіглись з тими, котрі були отримані під час ручних обчислень. Також програма може опрацьовувати матриці різних розмірів, а не лише матрицю 5 на 5.