МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”
Кафедра САПР

Комплексна лабораторна робота

з курсу: “Дискретні моделі в системному проектуванні”

Варіант 19

Виконала:
ст. гр. КН-416
Нестор Христина
Прийняв:
к.т.н., доцент каф. САПР
Кривий Р.З.

ЛЬВІВ 2020

ЛАБОРАТОРНА РОБОТА 1
на тему: «Алгоритм побудови дерев»

Мета роботи:

Вивчення алгоритмів рішення задач побудови остових дерев.

Короткі теоретичні відомості:

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.
Ліс - будь-яка сукупність дуг (ребер) інцидентних до вершин, які не містять циклів. Таким чином, ліс складається з одного або більше дерев.
Остове дерево графа - будь-яке дерево, яке утворене сукупністю дуг, які включають всі вершини графа.
Корінь орієнтованого дерева (прадерева) - його вершина, в яку не входить жодна з дуг.
Орієнтований ліс - визначається як звичайний, тільки складається не з простих дерев, а орієнтованих.
Остове орієнтоване дерево - орієнтоване дерево, яке одночасно є і остовим деревом.
Остовий орієнтований ліс - орієнтований ліс, який включає всі вершини відповідного графа.
Вага дерева - це сума ваг його ребер.
Максимальний орієнтований ліс графа G - орієнтований ліс графа G з максимально можливою вагою.
Максимальне орієнтоване дерево графа G - орієнтоване дерево графа G з максимально можливою вагою. Мінімальні орієнтовані ліс і дерево визначаються аналогічним чином.
Кущ(букет) - зв’язний фрагмент графа.
Алгоритм Борувки - 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.
Алгоритм Крускала - на кожному кроці до споруджуваного остового дерева додається найлегше ребро, яке з'єднує вершини з різних компонент. Таким чином на кожному кроці побудована множина складається з однієї або більше нетривіальних компонент, кожна з яких є підграфом деякого мінімального кістяка.
Алгоритм Прима - спочатку остов покладається складається з єдиної вершини (її можна вибрати довільно). Потім вибирається ребро максимальної ваги, що виходить з цієї вершини, і додається в максимальне остове дерево. Після цього остов містить уже дві вершини, і тепер шукається і додається ребро максимальної ваги, що має один кінець в одній з двох обраних вершин, а інший - навпаки, у всіх інших, крім цих двох. І так далі.

Текст програми:

```
with open('lab1.txt', 'r') as f:
   adjMatrix = [[int(num) for num in line.split(',')] for line in f]
   for row in adjMatrix:
      print (row)
   print ()
   setMatrix = []
for i in range(0,len(adjMatrix)):
   setMatrix.append([i])

def combine(e):
   e0 = -1
   e1 = -1
   for i in range(0,len(setMatrix)):
      if e[0] in setMatrix[i]:
         e0 = i
      if e[1] in setMatrix[i]:
         e1 = i
   setMatrix[e0] += setMatrix[e1]
   del setMatrix[e1]

while (len(setMatrix) > 1):
   edges = []
   for component in setMatrix:
      m = [999,[0,0]]
      for vertex in component:
         for i in range(0,len(adjMatrix[0])):
            if i not in component and adjMatrix[vertex][i] != 0:
               if (m[0] > adjMatrix[vertex][i]):
```

```
                m[0] = adjMatrix[vertex][i]
                m[1] = [vertex,i]
      if (m[1][0] > m[1][1]):
         m[1][0], m[1][1] =  m[1][1],m[1][0]
      if (m[1] not in edges):
         edges.append(m[1])
   for e in edges:
      combine(e)
   print("Зв'язки: " + str(edges) + " Групи: " + str(setMatrix))
```

Інструкція користувача:

Програму написано на мові програмування Python без застосування сторонніх модулів. Програма зчитує матрицю суміжності з файлу lab1.txt (кожен елемент у рядку відділений від іншого комою та пробілом) та записує дану матрицю у змінну adjMatrix. Змінна setMatrix слугує для збереження інформації про кожен вузол матриці. Функція combine() слугує для поєднання ребер у остовому лісі графа, до тих пір, поки ліс не перетвориться на дерево.

Вхідний файл:

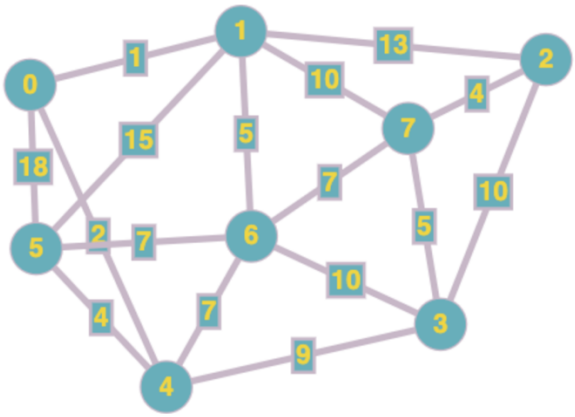
```
0, 1, 0, 0, 2, 18, 0, 0
1, 0, 13, 0, 0, 15, 5, 10
0, 13, 0, 10, 0, 0, 0, 4
0, 0, 10, 0, 9, 0, 10, 5
2, 0, 0, 9, 0, 4, 7, 0
18, 15, 0, 0, 4, 0, 7, 0
0, 5, 0, 10, 7, 7, 0, 7
0, 10, 4, 5, 0, 0, 7, 0
```

Результати виконання програми:

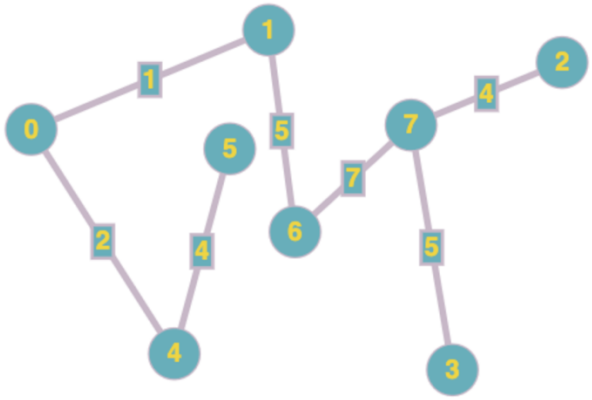
```
[0, 1, 0, 0, 2, 18, 0, 0]
[1, 0, 13, 0, 0, 15, 5, 10]
[0, 13, 0, 10, 0, 0, 0, 4]
[0, 0, 10, 0, 9, 0, 10, 5]
[2, 0, 0, 9, 0, 4, 7, 0]
[18, 15, 0, 0, 4, 0, 7, 0]
[0, 5, 0, 10, 7, 7, 0, 7]
[0, 10, 4, 5, 0, 0, 7, 0]


Зв'язки: [[0, 1], [2, 7], [3, 7], [0, 4], [4, 5], [1, 6]] Групи: [[0, 1, 4, 5, 6], [3,
2, 7]]
Зв'язки: [[6, 7]] Групи: [[0, 1, 4, 5, 6, 3, 2, 7]]
```

Розрахунки вручну:

Заданий граф:



Мінімальне остове дерево:



Висновки:

Під час даної лабораторної роботи було розглянуто алгоритми рішення задач побудови остових дерев. Зокрема було досліджено алгоритм Борувки та написано програму побудови мінімального покриваючого дерева. Результати виконання програми співпадають з результатами обрахунків вручну.

ЛАБОРАТОРНА РОБОТА 2
на тему: «Алгоритм рішення задачі листоноші»

Мета роботи:

Вивчення алгоритмів рішення задачі листоноші.

Короткі теоретичні відомості:

Ейлеревим циклом в графі називається шлях, який починається і закінчується в тій самій вершині, при чому всі ребра графа проходяться тільки один раз. *Ейлеревим шляхом* називається шлях, який починається в вершині А, а закінчується в вершині Б, і всі ребра проходяться лише по одному разу. Граф, який включає в себе ейлерів цикл називається ейлеревим. Будь-який листоноша перед тим, як відправитись в дорогу повинен підібрати на пошті листи, що відносяться до його дільниці, потім він повинен рознести їх адресатам, що розмістились вздовж маршрута його проходження, і повернутись на пошту. Кожен листоноша, бажаючи втратити якомога менше сил, хотів би подолати свій маршрут найкоротшим шляхом. Загалом, задача листоноші полягає в тому, щоб пройти всі вулиці маршрута і повернутися в його початкову точку, мінімізуючи при цьому довжину пройденого шляху. Перша публікація, присвячена рішенню подібної задачі, появилась в одному з китайських журналів, де вона й була названа задачею листоноші. Очевидно, що така задача стоїть не тільки перед листоношею. Наприклад, міліціонер хотів би знати найбільш ефективний шлях патрулювання вулиць свого району, ремонтна бригада зацікавлена у виборі найкоротшого шляху переміщення по всіх дорогах. Задача листоноші може бути сформульована в термінах теорії графів. Для цього побудуємо граф G = (X , E), в якому кожна дуга відповідає вулиці в маршруті руху листоноші, а кожна вершина - стик двох вулиць. Ця задача являє собою задачу пошуку найкоротшого маршруту, який включає кожне ребро хоча б один раз і закінчується у початковій вершині руху.

Текст програми:

Алгоритм Фльорі і подальший пошук Ейлерового циклу/шляху:

```
def fleury_walk(graph, start=None, circuit=False):
    visited = set()
    node = start if start else random.choice(graph.node_keys)

    route = [node]
    while len(visited) < len(graph):
        reduced_graph = copy.deepcopy(graph)
        reduced_graph.remove_edges(visited)
        options = reduced_graph.edge_options(node)
        bridges = [k for k in options.keys() if reduced_graph.is_bridge(k)]
        non_bridges = [k for k in options.keys() if k not in bridges]
        if non_bridges:
            chosen_path = random.choice(non_bridges)
        elif bridges:
            chosen_path = random.choice(bridges)
        else:
            break
        next_node = reduced_graph.edges[chosen_path].end(node)  # Other end

        visited.add(chosen_path)

        route.append(next_node)
        node = next_node

    return route

def eularian_path(graph, start=None, circuit=False):

    for i in range(1, 1001):
        route = fleury_walk(graph, start, circuit)
        if len(route) == len(graph) + 1:
            return route, i
    return [], i
```

Знаходимо ноди з непарним степенем, паруємо їх, відкидаємо дублікати:

```
def find_dead_ends(graph):

    single_nodes = [k for k, order in graph.node_orders.items() if order == 1]
    return set([x for k in single_nodes for x in graph.edges.values() \
            if k in (x.head, x.tail)])


def build_node_pairs(graph):
    odd_nodes = graph.odd_nodes
    return [x for x in itertools.combinations(odd_nodes, 2)]


def build_path_sets(node_pairs, set_size):
    return (x for x in itertools.combinations(node_pairs, set_size) \
            if all_unique(sum(x, ())))


def unique_pairs(items):
    for item in items[1:]:
        pair = items[0], item
        leftovers = [a for a in items if a not in pair]
```

```
        if leftovers:
            for tail in unique_pairs(leftovers):
                yield [pair] + tail
        else:
            yield [pair]
```

За допомогою алгоритма Дейкстри шукаємо ваги шляхів між вузлами з непарним степенем, знаходим найкоротші шляхи:

```
def summarize_path(end, previous_nodes):
    route = []
    prev = end
    while prev:
        route.insert(0, prev)
        prev = previous_nodes[prev]
    return route


def find_cost(path, graph):
    start, end = path

    all_nodes = graph.node_keys
    unvisited = set(all_nodes)
    total_cost = graph.total_cost
    node_costs = {node: total_cost for node in all_nodes}
    node_costs[start] = 0

    previous_nodes = {node: None for node in all_nodes}

    node = start
    while unvisited:
        for option in graph.edge_options(node).values():
            next_node = option.end(node)
            if next_node not in unvisited:
                continue
            if node_costs[next_node] > node_costs[node] + option.weight:
                node_costs[next_node] = node_costs[node] + option.weight
                previous_nodes[next_node] = node
        unvisited.remove(node)
        options = {k:v for k, v in node_costs.items() if k in unvisited}
        try:
            node = min(options, key=options.get)
        except ValueError:
            break
        if node == end:
            break

    cost = node_costs[end]
    shortest_path = summarize_path(end, previous_nodes)

    return cost, shortest_path
```

Інструкція користувача:

Прогоама написана на мові Python.

Використано наступні модулі: random (вибір початкової вершини у графі), argparse (парсинг аргументів з командної стрічки).

Вхідні дані знаходяться у файлі data.py, кожен граф представдений списком кортежів. Кожен з кортежів представляє ребро графа (початкова вершина, кінцева вершина, вага).
Запуск програми відбувається через консоль, в якості аргумента прописується назва змінної з необхідним графом, наприклад,

```
$ python main.py test1
```

Вхідні дані:

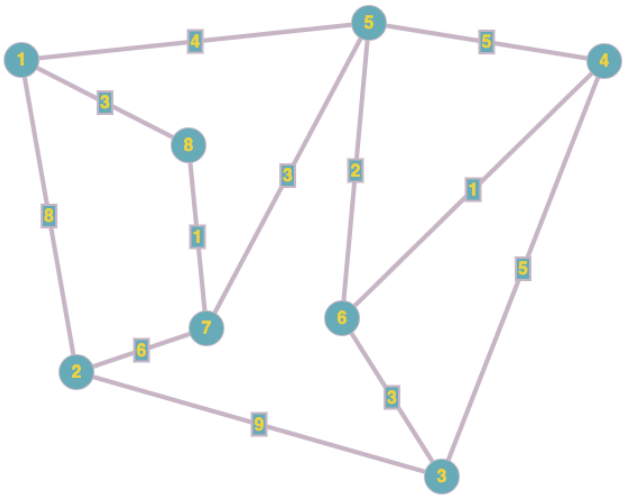
```
test1 = [
    (1,2,8), (1,5,4), (1,8,3), (2,3,9), (2,7,6), (3,4,5),
    (3,6,3), (4,5,5), (4,6,1), (5,6,2), (5,7,3), (7,8,1),
]
```

Результати виконання програми:

```
cristinas-MacBook-Pro:lab2 noirmeansblack$ python main.py test1
<12> edges
        Added 4 edges
        Total cost is 64
Solution: (<16> edges)
        [1, 5, 7, 8, 7, 2, 3, 6, 4, 6, 5, 4, 3, 2, 1, 8, 1]
cristinas-MacBook-Pro:lab2 noirmeansblack$
```

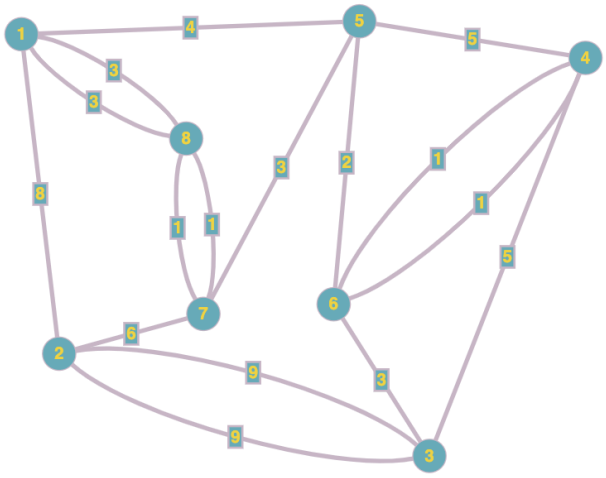
Розрахунки вручну:

Заданий граф:



Непарні вузли: 1, 2, 3, 4, 6, 7

Видозмінений граф за теоремою 1 є Ейлерівим графом, на відміну від початково заданого графа, у цього всі вузли мають парну кількість сполучень:



Прохід по графу у даному випадку може виконатись шляхом, описаним у результатах виконання програми:

1–>5–>7–>8–>7–>2–>3–>6–>4–>6–>5–>4–>3–>2–>1–>8–>1

Висновки:

Під час виконання даної лабораторної роботи я освоїла алгоритм розв’язку задачі за допомогою ейлерового циклу та ейлерового шляху. Написала програму на Python для перетворення початкового графу в Ейлерів граф (при потребі) і для вирішення задачі листоноші.

ЛАБОРАТОРНА РОБОТА 3
на тему: «Потокові алгоритми»

Мета роботи:

Вивчення потокових алгоритмів.

Короткі теоретичні відомості:

Потік-визначає спосіб пересилання деяких об’єктів з одного пункту вінший.

Розв’язання задачі потоку зводиться до таких основних підзадач:

- Максимізація сумарного обсягу перевезень
- Мінімізація вартості пересилань предметів з одного пункту в інший
- Мінімізація часу перевезень в заданій системі

Поняття потокові алгоритми включає в себе ряд алгоритмів.

Алгоритм пошуку збільшую чого ланцюга. Основна ідея алгоритму: побудова дерева, що росте з вершини “s” і складається з розфарбованих дуг, по яких з вершини “s” можуть пересилатись додаткові одиниці потоку.

В процесі виконання алгоритму можуть виникнути дві різні ситуації:

а) стік “t” є розфарбованим, тоді в побудованому з розфарбованих дуг дереві єдиний ланцюг з “s” в “t” є збільшуючим потік ланцюгом;

б) стік “t” не вдається розфарбувати, що означає: що у вихідній сітці не існує збільшуючого ланцюга між “s” і “t” .

Алгоритм пошуку потоку мінімальної вартості.

Дана задача полягає в організації пересилання з мінімальними витратами заданої кількості v одиниць потоку з витоку в стік в графі з заданими на дугах пропускними здатностями і вартостями проходження одної одиниці потоку.

Алгоритм дефекту.

Основна ідея алгоритму: рішення задачі про потік мінімальної вартості, але

на відміну від попереднього алгоритму алгоритм дефекту вирішує задачу про потік мінімальної вартості у випадку, коли найменша кількість одиниць потоку, яка повинна протікати по дузі, більша або рівна 0 для всіх дуг.

Потоки з підсиленням.

Попередні розгляди потоку показували, що потік не змінювався: одиниця ввійшла - одиниця вийшла. При проходженні потоку через дугу нові одиниці не створювались, але і старі не щезали. Потоки з підсиленням усувають припущення, згідно якого при проходженні по дугам потік залишається незмінним. Висувається припущення, що кількість одиниць в потоці, що проходить по дузі, може збільшуватись або зменшуватись. Точніше, вважають, що якщо в будь-яку дугу (x,y) в вершині “x” входить f(x,y) одиниць потоку, то з цієї дуги в вершині “y” вийде k(x,y)f(x,y) одиниць потоку. Можна вважають, що кожна одиниця потоку, що проходить по дузі (x,y), помножується на величину k(x,y) (яка називається підсиленням дуги (x,y).

Текст програми:

```
n, k = 5, 4
g = [[1], [1], [1, 2], [2, 3], [3, 4]]
mt = [-1] * k
used = list()

def try_kuhn(v):
    if used[v]:
        return False
    used[v] = True
    i = 0
    while i < len(g[v]):
        to = g[v][i]-1
        if mt[to] == -1 or try_kuhn(mt[to]):
            mt[to] = v
            return True
        i += 1
    return False

for v in range(n):
    used = [False] * n
    try_kuhn(v)
```

```
for i in range(k):
    if mt[i] != -1:
        print(str(mt[i]+1), str(i+1))
```

Інструкція користувача:

Програма написана мовою програмування Python, її запуск може відбуватись через консоль:

```
$ python lab3.py
```

Для задання кількості вершин зліва і справа ми використовуємо змінні n і k відповідно. Змінна g використовуються для зберігання списку правих вершин, з якими має зв’язок кожна ліва вершина.

Результати виконання програми:

```
1 1
3 2
4 3
5 4
```

Висновки:

При виконанні даної лабораторної роботи я вивчила, дослідила, а також описала мовою програмування Python потоковий алгоритм.

ЛАБОРАТОРНА РОБОТА 4
на тему: «Алгоритм рішення задачі комівояжера»

Мета роботи:

Вивчення і дослідження алгоритмів рішення задачі комівояжера.

Короткі теоретичні відомості:

Рішенням задачі комівояжера є оптимальний гамільтоновий контур. Нажаль, не всі графи містять гамільтоновий контур. Отже перед тим, ніж перейти до пошуку оптимального гамільтонового контура потрібно довести факт його існування в даному графі. Введемо ряд визначень, які в подальшому нам знадобляться. Граф називається сильно зв`язаним, якщо в ньому для будь-яких двох вершин “x” і “y” існує шлях від “x” до “y”. Підмножина вершин X деякого графа називається сильно зв`язаною, якщо для будь-яких пар вершин x □ X і y □ X існує шлях з “x” в “y” і X не є підмножиною ніякої іншої множини вершин, які володіють тими ж властивостями.

В кінці кінців один граф, в якому знайдений гамільтоновий контур мінімальної довжини, виключить з подальшого перегляду всі графи, що залишилися. Цей гамільтоновий контур повинен бути гамільтоновим контуром мінімальної довжину в вихідному графі G.

Метод поступового покращення рішення задачі комівояжера використовується для знаходження в графі G близького до оптимального (а інколи і оптимального) гамільтонового контура і полягає в наступному. Спочатку береться деякий гамільтоновий контур. Нехай x1, x2, ..., xn визначає послідовність , в якій в цьому контурі обходяться вершини графа G. Процес попарних перестановок повинен бути скінченним, так як:

- існує тільки скінченне число різних методів обходу n вершин графа (різних перестановок);
- кожен раз формується нова послідовність обходу, відмінна від попередніх.

При цьому може бути отриманий новий гамільтоновий контур, який має строго меншу довжину, ніж попередній. Потрібно зауважити, що при використанні метода покращення рішень результуючий гамільтоновий контур залежить від того, яким був вибраний початковий гамільтоновий

контур. Вдосконалення метода послідовного покращення рішення можна знайти в роботах Стекхана, а також Лина і Кернінга.

Текст програми:

```
#!/usr/bin/env python
#coding=utf8

import matplotlib.pyplot as plt
import numpy as np
from numpy import exp,sqrt
n=10;m=100;ib=3;way=[];a=0
#X=np.random.uniform(a,m,n)
#Y=np.random.uniform(a,m,n)

with open('data.txt', 'r') as f:
    XY = [[int(num) for num in line.split(',')] for line in f]
X = XY[0]
Y = XY[1]

#X=[10, 10, 100,100 ,30, 20, 20, 50, 50, 85, 85, 75, 35, 25, 30, 47, 50]
#Y=[5, 85, 0,90,50, 55,50,75 ,25,50,20,80,25,70,10,50,100]
#n=len(X)
M = np.zeros([n,n])
for i in np.arange(0,n,1):
         for j in np.arange(0,n,1):
                  if i!=j:
                           M[i,j]=sqrt((X[i]-X[j])**2+(Y[i]-Y[j])**2)
                  else:
                           M[i,j]=float('inf')
way.append(ib)
for i in np.arange(1,n,1):
         s=[]
         for j in np.arange(0,n,1):
                  s.append(M[way[i-1],j])
         way.append(s.index(min(s)))
         for j in np.arange(0,i,1):
                  M[way[i],way[j]]=float('inf')
                  M[way[i],way[j]]=float('inf')
S=sum([sqrt((X[way[i]]-X[way[i+1]])**2+(Y[way[i]]-Y[way[i+1]])**2) for i in
np.arange(0,n-1,1)])+ sqrt((X[way[n-1]]-X[way[0]])**2+(Y[way[n-1]]-Y[way[0]])**2)
plt.title('загальний шлях: %s.номер міста: %i.всього міст: %i.'%(round(S,3),ib,n),
size=14)
X1=[X[way[i]] for i in np.arange(0,n,1)]
Y1=[Y[way[i]] for i in np.arange(0,n,1)]
plt.plot(X1, Y1, color='g', linestyle=' ', marker='o')
plt.plot(X1, Y1, color='b', linewidth=1)
X2=[X[way[n-1]],X[way[0]]]
Y2=[Y[way[n-1]],Y[way[0]]]
plt.plot(X2, Y2, color='r', linewidth=2,  linestyle='-', label='шлях між першим і \n
останнім містами')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```

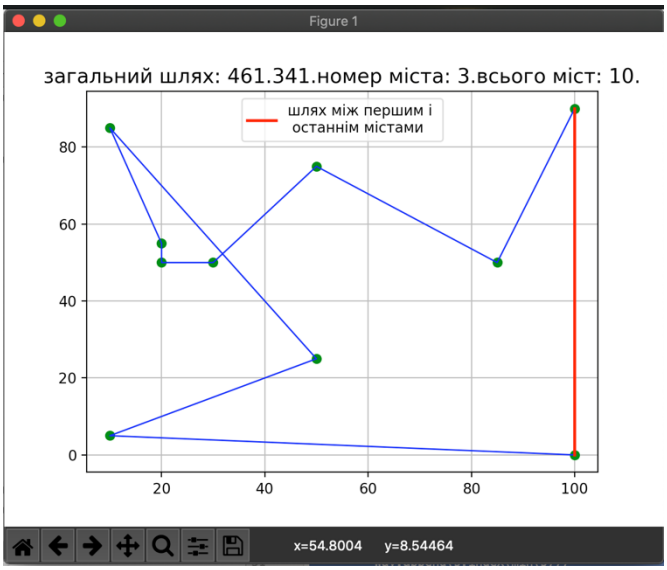
Інструкція користувача:

Програма написана мовою програмування Python та може запускатись з консолі як скрипт:

```
$ python3 lab4.py
```

Початкові дані можуть бути зчитані з файла `data.txt` або згенеровані випадково та записуються у змінні X та Y. У даній програмі застосовуються модулі numpy (математичні вирази) та matplotlib (побудова графіків).

Результати виконання програми:



Висновки:

У даній лабораторній роботі мною було вивчено і досліджено практично способи рішення задачі комівояжера. Було написано програму на мові Python, котра методом найближчого сусіда вирішує задачу комівояжера.

ЛАБОРАТОРНА РОБОТА 5
на тему: «Ізоморфізм графів»

Мета роботи:

Вивчення і дослідження основних підходів до встановлення ізоморфізму графів.

Короткі теоретичні відомості:

Теорія графів дає простий, доступний і потужній інструмент побудови моделей і рішення задач впорядкування взаємозвязаних обєктів. Нині є багато проблем де необхідно дослідити деякі складні системи з допомогою впорядкування їх елементів. До таких проблем відносяться і задачі ідентифікації в електричних схемах, в авіації, в органічній хімії і т.д. Вирішення таких проблем досягається з допомогою встановлення ізоморфізму графів. Два графа G=(X,U,P) і G'=(X',U',P') називаються ізоморфними, якщо між їх вершинами, а також між їхніми ребрами можна встановити взаємно однозначне співвідношення X <-> X', U <-> U', що зберігає інцидентність, тобто таке, що для всякої пари (x,y)єX ребра u є U, що з'єднує їх, обов'язково існує пара (x',y') є X' і ребро u' є U', що з'єднує їх, і навпаки. Тут P - предикат, інцидентор графа G. Зауважимо, що відношення ізоморфізму графів рефлексивне, симетричне і транзитивне, тобто представляє собою еквівалентність. На даний час існує досить детальна класифікація розроблених методів рішення такого типу задач /1/. Розглядаючи комбінаторно-логічну природу вказаної задачі можна всі роботи в цьому напрямку розділити на дві групи: рішення теоретичної задачі встановлення ізоморфізму простих графів; розробка наближених методів, які найбільш повно враховують обмеження і специфіку задачі з застосуванням характерних ознак об’єкту дослідження. До першої групи відносяться алгоритми: повного перебору і почергового “підвішування” графів за вершини.

а) Одним з найпростіших з точки зору програмної реалізації, є алгоритм перевірки ізоморфізму графів повним перебором(можливої перенумерації вершин), але складність цього алгоритму є факторіальною.

б) Почергове “підвішування” графів за вершини (всі ребра зрівноважені). Суть цього

алгоритму полягає в знаходженні однакових “підвішаних” графів (за довільні вершини), ізоморфність яких визначаємо. При чому водному з графів почергово змінюється вершина за яку він “підвішується”. Ізоморфізм графів визначається по їх матрицях суміжності, які формуються по однотипних правилах.

Текст програми:

```
from collections import defaultdict
from itertools import permutations

with open('dmsp_lab_data(5)1.data') as file:
    graph1 = file.read()
with open('dmsp_lab_data(5)2.data') as file:
    graph2 = file.read()
graph1 = [string.split(' ') for string in graph1.split('\n')]
graph2 = [string.split(' ') for string in graph2.split('\n')]
nodes1 = len(graph1)
nodes2 = len(graph2)

class Node:
    def __init__(self, number):
        self.number = number
    def __repr__(self):
        return str(self.number)

def equal(defdictL, defdictR):
    notEqual = False
    for keyL in defdictL:
        for keyR in defdictR:
            if keyL.number == keyR.number and len(defdictL[keyL]) ==
len(defdictR[keyR]):
                tempList1 = [n.number for n in defdictL[keyL]]
                tempList2 = [n.number for n in defdictR[keyR]]
                tempList1.sort()
                tempList2.sort()
                if tempList1 != tempList2:
                    notEqual = True
                    break
        if notEqual:
            break
    return not notEqual

if nodes1 != nodes2:
    print("Кількість вершин не співпадає! Графи не ізоморфні.")
else:
    nodesNum = nodes1
    edges1 = list()
    edges2 = list()
    for i in range(nodesNum):
        for j in range(nodesNum):
            graph1[i][j] = int(graph1[i][j])
            graph2[i][j] = int(graph2[i][j])
        edges1.append(sum(graph1[i]))
        edges2.append(sum(graph2[i]))
    edges1.sort()
    edges2.sort()
    if edges1 != edges2:
```

```
        print("Степені вершин не співпадають! Графи не ізоморфні.")
    else:
        isomorphic = False
        neighbors1 = defaultdict(list)
        neighbors2 = defaultdict(list)
        nodes1 = [Node(number) for number in list(range(nodesNum))]
        nodes2 = [Node(number) for number in list(range(nodesNum))]
        for y in range(nodesNum):
            for x in range(nodesNum):
                if graph1[y][x] != 0:
                    neighbors1[nodes1[y]].append(nodes1[x])
                if graph2[y][x] != 0:
                    neighbors2[nodes2[y]].append(nodes2[x])
        counter = 0
        for variant in permutations(list(range(nodesNum))):
            if equal(neighbors1, neighbors2):
                print(counter, "ітерацій! Графи ізоморфні")
                isomorphic = True
                break
            else:
                index = 0
                for number in variant:
                    nodes2[index].number = number
                    index += 1
            counter += 1
        if not isomorphic:
            print("Зв'язки між вершинами не ідентичні! Графи не ізоморфні.")
```

Інструкція користувача:

Програма написана мовою програмування Python та може запускатись з консолі як скрипт:

```
$ python3 lab5.py
```

Початкові дані про обидва графи беруться з файлів `dmsp_lab_data(5)1.data` та `dmsp_lab_data(5)2.data` - у них записуються матриці досяжностей графів.

Результати виконання програми:

23 ітерацій! Графи ізоморфні

Висновки:

При виконанні даної лабораторної роботи мною було досліджено та вивчено основні підходи до встановлення ізоморфізму графів.