Міністерство освіти і науки України

Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"

Факультет інформатики та обчислювальної техніки

Катедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 5 з дисципліни

«Проектування алгоритмів»

„Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.2"

Виконав(ла) *ІП-11 Лесів В.І.* __________

(шифр, прізвище, ім'я, по батькові)

Перевірив *Головченко М.Н.* __________

(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні підходи розробки метаевристичних алгоритмів для типових прикладних задач. Опрацювати методологію підбору прийнятних параметрів алгоритму.

2 ЗАВДАННЯ

Згідно варіанту, формалізувати алгоритм вирішення задачі відповідно загальної методології.

Записати розроблений алгоритм у покроковому вигляді. З достатнім степенем деталізації.

Виконати його програмну реалізацію на будь-якій мові програмування.

Перелік задач наведено у таблиці 2.1.

Перелік алгоритмів і досліджуваних параметрів у таблиці 2.2.

Задача і алгоритм наведені в таблиці 2.3.

Змінюючи параметри алгоритму, визначити кращі вхідні параметри алгоритму. Для цього необхідно:

– обрати критерій зупинки алгоритму (кількість ітерацій або значення ЦФ);

– зафіксувати усі параметри крім одного і змінювати цей параметр, поки не буде досягнуто пікової ефективності;

– після цього параметр фіксується і змінюються інші параметри;

– далі повторюємо процедуру спочатку, з першого зафіксованого параметру;

– зупиняємось коли будуть знайдені оптимальні параметри для даної задачі або встановлена залежність одних параметрів від інших.

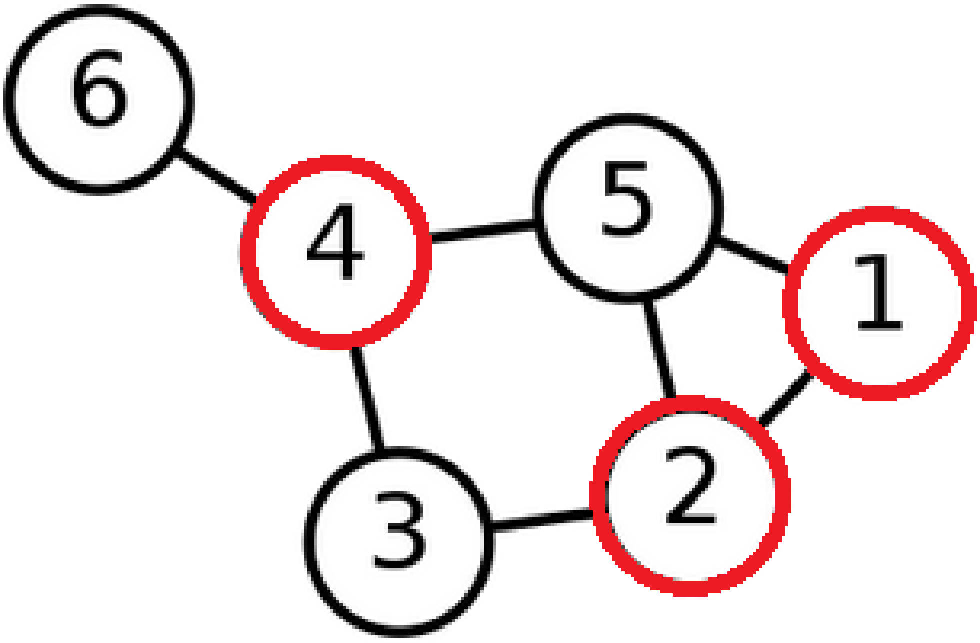
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

Таблиця 2.1 – Прикладні задачі

№	Задача
1	**Задача про рюкзак** (місткість P=500, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 20 (випадкова)). Для заданої множини предметів, кожен з яких має вагу і цінність, визначити яку кількість кожного з предметів слід взяти, так, щоб

	сумарна вага не перевищувала задану, а сумарна цінність була максимальною. Задача часто виникає при розподілі ресурсів, коли наявні фінансові обмеження, і вивчається в таких областях, як комбінаторика, інформатика, теорія складності, криптографія, прикладна математика.
2	**Задача комівояжера** (300 вершин, відстань між вершинами випадкова від 5 до 150) полягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста хоча б по одному разу. В умовах завдання вказуються критерій вигідності маршруту (найкоротший, найдешевший, сукупний критерій тощо) і відповідні матриці відстаней, вартості тощо. Зазвичай задано, що маршрут повинен проходити через кожне місто тільки один раз, в такому випадку розв'язок знаходиться серед гамільтонових циклів. **Розглядається симетричний, асиметричний та змішаний варіанти.** В загальному випадку, асиметрична задача комівояжера відрізняється тим, що ребра між вершинами можуть мати різну вагу в залежності від напряму, тобто, задача моделюється орієнтованим графом. Таким чином, окрім ваги ребер графа, слід також зважати і на те, в якому напрямку знаходяться ребра. У випадку симетричної задачі всі пари ребер між одними й тими самими вершинами мають однакову вагу. У випадку реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості або довжини маршрутів і напряму руху. Застосування: – доставка товарів (в цьому випадку може бути більш доречна постановка транспортної задачі - доставка в кілька магазинів з декількох складів); – доставка води;

	– моніторинг об'єктів; – поповнення банкоматів готівкою; – збір співробітників для доставки вахтовим методом.
3	**Розфарбовування графа** (300 вершин, степінь вершини не більше 30, але не менше 2) – називають таке приписування кольорів (або натуральних чисел) його вершинам, що ніякі дві суміжні вершини не набувають однакового кольору. Найменшу можливу кількість кольорів у розфарбуванні називають хроматичне число. Застосування: – розкладу для освітніх установ; – розкладу в спорті; – планування зустрічей, зборів, інтерв'ю; – розклади транспорту, в тому числі - авіатранспорту; – розкладу для комунальних служб;
4	**Задача вершинного покриття** (300 вершин, степінь вершини не більше 30, але не менше 2**.** Вершинне покриття для неорієнтованого графа G = (V, E) - це множина його вершин S, така, що, у кожного ребра графа хоча б один з кінців входить в вершину з S. Задача вершинного покриття полягає в пошуку вершинного покриття найменшого розміру для заданого графа (цей розмір називається числом вершинного покриття графа). На вході: Граф G = (V, E). Результат: множина $C \subseteq V$ - найменше вершинне покриття графа G.

	 Застосування: – розміщення пунктів обслуговування; – призначення екіпажів на транспорт; – проектування інтегральних схем і конвеєрних ліній.
5	**Задача про кліку** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Клікою в неорієнтованому графі називається підмножина вершин, кожні дві з яких з'єднані ребром графа. Іншими словами, це повний підграф первісного графа. Розмір кліки визначається як число вершин в ній. Задача про кліку існує у двох варіантах: у **задачі розпізнавання** потрібно визначити, чи існує в заданому графі G кліка розміру k, тоді як в **обчислювальному варіанті** потрібно знайти в заданому графі G кліку максимального розміру або всі максимальні кліки (такі, що не можна збільшити). Застосування: – біоінформатика; – електротехніка;
6	**Задача про найкоротший шлях** (300 вершин, відстань між вершинами випадкова від 5 до 150, степінь вершини не більше 10, але

	не менше 1) - задача пошуку найкоротшого шляху (ланцюга) між двома точками (вершинами) на графі, в якій мінімізується сума ваг ребер, що складають шлях. Важливість задачі визначається її різними практичними застосуваннями. Наприклад, в GPS-навігаторах здійснюється пошук найкоротшого шляху між точкою відправлення і точкою призначення. Як вершин виступають перехрестя, а дороги є ребрами, які лежать між ними. Якщо сума довжин доріг між перехрестями мінімальна, тоді знайдений шлях найкоротший.

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

№	**Алгоритми і досліджувані параметри**
1	**Генетичний алгоритм:** - оператор схрещування (мінімум 3); - мутація (мінімум 2); - оператор локального покращення (мінімум 2).
2	**Мурашиний алгоритм**: – α; – β; – ρ; – Lmin; – кількість мурах M і їх типи (елітні, тощо…); – маршрути з однієї чи різних вершин.
3	**Бджолиний алгоритм:** – кількість ділянок; – кількість бджіл (фуражирів і розвідників).

Таблиця 2.3 – Варіанти задач і алгоритмів

№	**Задачі і алгоритми**
1	Задача про рюкзак + Генетичний алгоритм
2	Задача про рюкзак + Бджолиний алгоритм
3	Задача комівояжера (асиметрична мережа) + Генетичний алгоритм
4	Задача комівояжера (симетрична мережа) + Генетичний алгоритм
5	Задача комівояжера (змішана мережа) + Генетичний алгоритм
6	Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм
7	Задача комівояжера (симетрична мережа) + Мурашиний алгоритм
8	Задача комівояжера (змішана мережа) + Мурашиний алгоритм
9	Задача вершинного покриття + Генетичний алгоритм
10	Задача вершинного покриття + Бджолиний алгоритм
11	Задача комівояжера (асиметрична мережа) + Бджолиний алгоритм
12	Задача комівояжера (симетрична мережа) + Бджолиний алгоритм
13	Задача комівояжера (змішана мережа) + Бджолиний алгоритм
14	Розфарбовування графа + Генетичний алгоритм
15	Розфарбовування графа + Бджолиний алгоритм
16	Задача про кліку (задача розпізнавання) + Генетичний алгоритм
17	Задача про кліку (задача розпізнавання) + Бджолиний алгоритм
18	Задача про кліку (обчислювальна задача) + Генетичний алгоритм
19	Задача про кліку (обчислювальна задача) + Бджолиний алгоритм
20	Задача про найкоротший шлях + Генетичний алгоритм
21	Задача про найкоротший шлях + Мурашиний алгоритм
22	Задача про найкоротший шлях + Бджолиний алгоритм
23	Задача про рюкзак + Генетичний алгоритм
24	Задача про рюкзак + Бджолиний алгоритм
25	Задача комівояжера (асиметрична мережа) + Генетичний алгоритм
26	Задача комівояжера (симетрична мережа) + Генетичний алгоритм
27	Задача комівояжера (змішана мережа) + Генетичний алгоритм

28	Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм
29	Задача комівояжера (симетрична мережа) + Мурашиний алгоритм
30	Задача комівояжера (змішана мережа) + Мурашиний алгоритм

3 ВИКОНАННЯ

3.1 Покроковий алгоритм

Обираємо початкову популяцію.

```
function generateRandomPopulation():
    population ← []
    flags ← [False for i ← 0 to nodesNumber]
    for i ← 0 to populNumber do
        rand ← randint(0, nodesNumber - 1)
        flags[rand] ← True
        clique ← Clique(rand)
        sortedList ← clique.computeSortedList()
        cnt ← 0
        while length(clique.pa) > 0 do
            node ← sortedList[cnt].node
            cnt ←  cnt + 1
            if clique.containsInPA(node) then
                clique.addVertex(node)
        end while
        population.append(clique)
    end for
    node ← graph.sortedNodes[0].value
    clique ← Clique(node)
    sortedList ← clique.computeSortedList()
    count ← 0
    while length(clique.pa) > 0 do
        node ← sortedList[count].node
        count ← 1
        if clique.containsInPA(node) then
            clique.addVertex(node)
    end while
    population.append(clique)
    return population
```

Застосовуємо оператори схрещування.

```
function greedyCrossover(c1, c2):
    vec ← []
    flags ← [False for i ← 0 to nodesNumber]
    for i ← 0 to length(c1.clique)) do
        vertex ← c1.clique[i]
        if not (flags[vertex]) then
```

```
        vec.append(vertex)
        flags[vertex] ← True
    end if
end for
for i ← 0 to length(c2.clique)) do
    vertex ← c2.clique[i]
    if not (flags[vertex]) then
        vec.append(vertex)
        flags[vertex] ← True
    end if
end for
sortedList ← []
for i ← 0 to length(vec) do
    node1 ← vec[i]
    reach ← 0
    for j ← 0 to length(vec) do
        if i = j then
            continue
        node2 ← vec[j]
        if graph.aMatrix[node1][node2] = 1 then
            reach ←  reach + 1
    end for
    sNode ← SortedListNode()
    sNode.reach ← reach
    sNode.node ← node1
    sortedList.append(sNode)
end for
sortedList.sort(key=lambda x: x.reach)
firstVertex ← sortedList[0].node
clique ← Clique(firstVertex)
count ← 1
while count < length(sortedList) do
    node ← sortedList[count].node
    if clique.containsInPA(node) then
        clique.addVertex(node)
    count ←  count + 1
end while
while length(clique.pa) > 0 do
    node ← clique.pa[0]
    clique.addVertex(node)
end while
return clique
```

```
function intersectionCrossover(c1, c2):
    intersect ← []
    flags ← [False for i ← 0 to nodesNumber]
    for i ← 0 to length(c2.clique)) do
        vertex ← c2.clique[i]
        flags[vertex] ← True
    end for
    for i ← 0 to length(c1.clique)) do
        ver1 ← c1.clique[i]
        if flags[ver1] then
            intersect.append(ver1)
    end for
    if length(intersect) = 0 then
        return greedyCrossover(c1, c2)
    vertex ← intersect[0]
    clique ← Clique(vertex)
    for i ← 1 to length(intersect) do
        vertex ← intersect[i]
        if clique.containsInPA(vertex) then
            clique.addVertex(vertex)
    end for
    if length(clique.pa) > 0 then
        sortedList ← clique.computeSortedList()
        cnt ← 0
        while length(clique.pa) > 0 do
            node ← sortedList[cnt].node
            cnt ←  cnt + 1
            if clique.containsInPA(node) then
                clique.addVertex(node)
        end while
    end if
    return clique
```

Застосовуємо оператор мутації.

```
function mutate(clique):
    flags ← [False for i ← 0 to nodesNumber]
    for i ← 0 to mutationNum do  # MUTATIONS
        rand ← randint(0, length(clique.clique) - 1)
        count ← 0
        while flags[rand] do
            rand ← randint(0, length(clique.clique) - 1)
            count ←  count + 1
```

```
            if count > 100 then
                break
        end while
        flags[rand] ← True
        vertex ← clique.clique[rand]
        clique.removeVertex(vertex)
    end for
    rand ← random()
    if rand < 0.5 then
        sortedList ← clique.computeSortedList()
        cnt ← 0
        while length(clique.pa) > 0 do
            node ← sortedList[cnt].node
            cnt ←  cnt + 1
            if clique.containsInPA(node) then
                clique.addVertex(node)
        end while
    else
        while length(clique.pa) > 0 do
            rand ← randint(0, length(clique.pa) - 1)
            vertex ← clique.pa[rand]
            clique.addVertex(vertex)
        end while
    end if
```

Застосовуємо оператор локального покращення.

```
function localImprovement(clique):
    gBest ← clique.clone()
    for i ← 0 to lINum:  # LOCAL IMPROVEMENT
        rand1 ← randint(0, length(clique.clique) - 1)
        rand2 ← randint(0, length(clique.clique) - 1)
        countt ← 0
        while rand1 = rand2 do
            countt ←  countt + 1
            if countt > 100 then  # UNIQUE OPERATIONS
                break
            rand1 ← randint(0, length(clique.clique) - 1)
            rand2 ← randint(0, length(clique.clique) - 1)
        end while
        vertex1 ← clique.clique[rand1]
        vertex2 ← clique.clique[rand2]
        clique.removeVertex(vertex1)
```

```
        clique.removeVertex(vertex2)
        sortedList ← clique.computeSortedList()
        count ← 0
        while length(clique.pa) > 0 do
            node ← sortedList[count].node
            count ←  count + 1
            if node >= nodesNumber then
                sys.exit("Node greater", node)
            if clique.containsInPA(node) then
                clique.addVertex(node)
        end while
        if length(gBest.clique) < length(clique.clique) then
            gBest ← clique.clone()
    clique ← gBest
    return clique
```

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

Main.py

```python
from clique import *
from graphColor import *


def main():
    for i in range(nodesNumber):
        degree = randint(2, 30)
        if graph.powers[i] < degree:
            for j in range(graph.powers[i], degree):
                try:
                    connect = randint(i + 1, nodesNumber - 1)
                except ValueError:
                    continue
                while graph.powers[connect] == 200 or connect == i:
                    connect = randint(0, nodesNumber - 1)
                graph.addEdge(i, connect)
                graph.powers[i] += 1
                graph.powers[connect] += 1

    graph.sortList()
    population = generateRandomPopulation()
    population.sort(key=lambda x: len(x.clique), reverse=True)
    gBest = population[0].clone()
    prevBest = len(gBest.clique)
    cnt = 0
    for n in range(1000):
        if prevBest == len(gBest.clique):
            cnt += 1
            if cnt > 10:
                pop = generateRandomPopulation()
                pop, population = population, pop
```

```
                cnt = 0
        else:
            prevBest = len(gBest.clique)
            cnt = 0
        newPopulation = []
        population.sort(key=lambda x: len(x.clique), reverse=True)
        localBest = population[0]
        if len(gBest.clique) < len(localBest.clique):
            gBest = localBest.clone()
        gBest=localImprovement(gBest)
        newPopulation.append(gBest)
        print(n, ":", len(gBest.clique))
        for i in range(populNumber):
            parents = randomSelection(population)
            offspring = intersectionCrossover(parents[0], parents[1])
            offspring=localImprovement(offspring)
            if len(offspring.clique) <= len(parents[0].clique) or
len(offspring.clique) <= len(parents[1].clique):
                mutate(offspring)
            newPopulation.append(offspring)
        population, newPopulation = newPopulation, population
        print("Vertices in the Clique:", len(gBest.clique))
        print([i + 1 for i in gBest.clique])

    g = graphVisualisation()
    g.addEdges(graph.aMatrix, nodesNumber)
    color = ["blue" for _ in range(nodesNumber)]
    for i in gBest.clique:
        color[i] = "red"
    g.visualize(color, nodesNumber)


if __name__ == "__main__":
    main()
```

clique.py

```
from random import randint, random
import sys

global nodesNumber
nodesNumber = 300
global populNumber
populNumber=13
global mutationNum
global lINum
mutationNum=2
lINum=2



class Node:
    def __init__(self, value=0):
        self.value = value
        self.degree = 0
        self.edges = []

    def addEdge(self, v):
        self.edges.append(v)


class SortedListNode:
```

```
    def __init__(self):
        self.node = -1
        self.reach = 0


class Graph:
    def __init__(self):
        self.aMatrix = [[0 for j in range(nodesNumber)] for i in
range(nodesNumber)]
        self.nodes = [None for i in range(nodesNumber)]
        self.powers = [0 for i in range(nodesNumber)]
        self.sortedNodes = []

    def addEdge(self, sv, ev):
        self.aMatrix[sv][ev] = 1
        self.aMatrix[ev][sv] = 1
        node = self.nodes[sv]
        if node is None:
            node = Node(sv)
            self.nodes[sv] = node
            node.addEdge(ev)
            self.sortedNodes.append(node)
            node.degree += 1
        else:
            node.addEdge(ev)
            node.degree += 1
        node = self.nodes[ev]
        if node is None:
            node = Node(ev)
            self.nodes[ev] = node
            node.addEdge(sv)
            self.sortedNodes.append(node)
            node.degree += 1
        else:
            node.addEdge(sv)
            node.degree += 1

    def sortList(self):
        self.sortedNodes.sort(key=lambda x: x.degree, reverse=True)

global graph
graph = Graph()


class Clique:
    def __init__(self, firstVertex="def"):
        self.clique = []
        self.pa = []
        self.mapPA = dict()
        self.mapClique = dict()
        if firstVertex != "def":
            self.clique.append(firstVertex)
            self.mapClique[firstVertex] = True
            for i in range(nodesNumber):
                if i == firstVertex:
                    continue
                elif graph.aMatrix[i][firstVertex] == 1:
                    self.pa.append(i)
                    self.mapPA[i] = True

    def addVertex(self, vertex):
        if self.containsInClique(vertex):
            return
        self.clique.append(vertex)
```

```
        self.mapClique[vertex] = True
        self.eraseFromPA(vertex)
        erasedNodes = []
        for i in range(len(self.pa)):
            pavertex = self.pa[i]
            if graph.aMatrix[pavertex][vertex] == 0:
                erasedNodes.append(pavertex)
        for i in range(len(erasedNodes)):
            self.eraseFromPA(erasedNodes[i])

    def removeVertex(self, vertex):
        if not (self.containsInClique(vertex)):
            return
        self.eraseFromClique(vertex)
        for i in range(nodesNumber):
            if self.containsInClique(i):
                continue
            else:
                flag = True
                for n in range(len(self.clique)):
                    ver = self.clique[n]
                    if graph.aMatrix[i][ver] == 0:
                        flag = False
                        break
                if flag:
                    if not i in self.mapPA:
                        self.pa.append(i)
                        self.mapPA[i] = True

    def eraseFromPA(self, vertex):
        self.mapPA.pop(vertex)
        if vertex in self.pa:
            self.pa.remove(vertex)

    def containsInPA(self, vertex):
        return (vertex in self.mapPA)

    def eraseFromClique(self, vertex):
        self.mapClique.pop(vertex)
        if vertex in self.clique:
            self.clique.remove(vertex)

    def containsInClique(self, vertex):
        for i in self.mapClique:
            if self.mapClique[i] == vertex:
                return True
        return False

    def computeSortedList(self):
        sortedList = []
        for i in range(len(self.pa)):
            node1 = self.pa[i]
            reach = 0
            for j in range(len(self.pa)):
                if i == j:
                    continue
                node2 = self.pa[j]
                if graph.aMatrix[node1][node2] == 1:
                    reach += 1
            n = SortedListNode()
            n.reach = reach
            n.node = node1
            sortedList.append(n)
        sortedList.sort(key=lambda x: x.reach)
```

```
        return sortedList

    def clone(self):
        cpa = []
        cclique = []
        for i in range(len(self.pa)):
            cpa.append(self.pa[i])
        for i in range(len(self.clique)):
            cclique.append(self.clique[i])
        cMapPa = dict(self.mapPA)
        cMapClique = dict(self.mapClique)
        clone = Clique()
        clone.clique = cclique
        clone.pa = cpa
        clone.mapPA = cMapPa
        clone.mapClique = cMapClique
        return clone


def generateRandomPopulation():
    population = []
    flags = [False for i in range(nodesNumber)]
    for i in range(populNumber):   # POPULATION
        rand = randint(0, nodesNumber - 1)
        cntt = 0
        while flags[rand]:
            cntt += 1
            if cntt > nodesNumber:
                break
            rand = randint(0, nodesNumber - 1)
        flags[rand] = True
        clique = Clique(rand)
        sortedList = clique.computeSortedList()
        cnt = 0
        while len(clique.pa) > 0:
            if cnt == len(sortedList):
                break
            node = sortedList[cnt].node
            cnt += 1
            if clique.containsInPA(node):
                clique.addVertex(node)
        population.append(clique)
    node = graph.sortedNodes[0].value
    clique = Clique(node)
    sortedList = clique.computeSortedList()
    count = 0
    while len(clique.pa) > 0:
        node = sortedList[count].node
        count += 1
        if clique.containsInPA(node):
            clique.addVertex(node)
    population.append(clique)
    return population


def greedyCrossover(c1, c2):
    vec = []
    flags = [False for i in range(nodesNumber)]
    for i in range(len(c1.clique)):
        vertex = c1.clique[i]
        if not (flags[vertex]):
            vec.append(vertex)
            flags[vertex] = True
```

```python
    for i in range(len(c2.clique)):
        vertex = c2.clique[i]
        if not (flags[vertex]):
            vec.append(vertex)
            flags[vertex] = True

    sortedList = []
    for i in range(len(vec)):
        node1 = vec[i]
        reach = 0
        for j in range(len(vec)):
            if i == j:
                continue
            node2 = vec[j]
            if graph.aMatrix[node1][node2] == 1:
                reach += 1
        sNode = SortedListNode()
        sNode.reach = reach
        sNode.node = node1
        sortedList.append(sNode)
    sortedList.sort(key=lambda x: x.reach)
    firstVertex = sortedList[0].node
    clique = Clique(firstVertex)
    count = 1
    while count < len(sortedList):
        node = sortedList[count].node
        if clique.containsInPA(node):
            clique.addVertex(node)
        count += 1
    while len(clique.pa) > 0:
        node = clique.pa[0]
        clique.addVertex(node)
    return clique


def intersectionCrossover(c1, c2):
    intersect = []
    flags = [False for i in range(nodesNumber)]
    for i in range(len(c2.clique)):
        vertex = c2.clique[i]
        flags[vertex] = True
    for i in range(len(c1.clique)):
        ver1 = c1.clique[i]
        if flags[ver1]:
            intersect.append(ver1)
    if len(intersect) == 0:
        return greedyCrossover(c1, c2)
    vertex = intersect[0]
    clique = Clique(vertex)
    for i in range(1, len(intersect)):
        vertex = intersect[i]
        if clique.containsInPA(vertex):
            clique.addVertex(vertex)
    if len(clique.pa) > 0:
        sortedList = clique.computeSortedList()
        cnt = 0
        while len(clique.pa) > 0:
            node = sortedList[cnt].node
            cnt += 1
            if clique.containsInPA(node):
                clique.addVertex(node)
    return clique
```

```python
def randomSelection(population):
    parents = []
    rand1 = randint(0, populNumber)
    rand2 = randint(0, populNumber)
    while rand1 == rand2:
        rand1 = randint(0, populNumber)
        rand2 = randint(0, populNumber)
    p1 = population[rand1]
    p2 = population[rand2]
    parents.append(p1)
    parents.append(p2)
    return parents


def localImprovement(clique):
    gBest = clique.clone()
    for i in range(lINum):  # LOCAL IMPROVEMENT
        rand1 = randint(0, len(clique.clique) - 1)
        rand2 = randint(0, len(clique.clique) - 1)
        countt = 0
        while rand1 == rand2:
            countt += 1
            if countt > 100:  # UNIQUE OPERATIONS
                break
            rand1 = randint(0, len(clique.clique) - 1)
            rand2 = randint(0, len(clique.clique) - 1)
        vertex1 = clique.clique[rand1]
        vertex2 = clique.clique[rand2]
        clique.removeVertex(vertex1)
        clique.removeVertex(vertex2)
        sortedList = clique.computeSortedList()
        count = 0
        while len(clique.pa) > 0:
            node = sortedList[count].node
            count += 1
            if node >= nodesNumber:
                sys.exit("Node greater", node)
            if clique.containsInPA(node):
                clique.addVertex(node)
        if len(gBest.clique) < len(clique.clique):
            gBest = clique.clone()
    clique = gBest
    return clique


def mutate(clique):
    flags = [False for i in range(nodesNumber)]
    for i in range(mutationNum):  # MUTATIONS
        rand = randint(0, len(clique.clique) - 1)
        count = 0
        while flags[rand]:
            rand = randint(0, len(clique.clique) - 1)
            count += 1
            if count > 100:
                break
        flags[rand] = True
        vertex = clique.clique[rand]
        clique.removeVertex(vertex)
    rand = random()
    if rand < 0.5:
        sortedList = clique.computeSortedList()
        cnt = 0
        while len(clique.pa) > 0:
            node = sortedList[cnt].node
```

```python
            cnt += 1
            if clique.containsInPA(node):
                clique.addVertex(node)
    else:
        while len(clique.pa) > 0:
            rand = randint(0, len(clique.pa) - 1)
            vertex = clique.pa[rand]
            clique.addVertex(vertex)
```

graphColor.py

```python
import networkx as nx
import matplotlib.pyplot as plt


class graphVisualisation:
    def __init__(self):
        self.visual = []

    def addEdges(self, aMatrix,s):
        for i in range(s):
            for j in range(s):
                if aMatrix[i][j]==1:
                    self.visual.append([i, j])

    def visualize(self, color,k):
        g = nx.Graph()
        g.add_nodes_from([i for i in range(k)])
        g.add_edges_from(self.visual)
        nx.draw_networkx(g, node_color=color,with_labels=True)
        plt.show()
```

3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

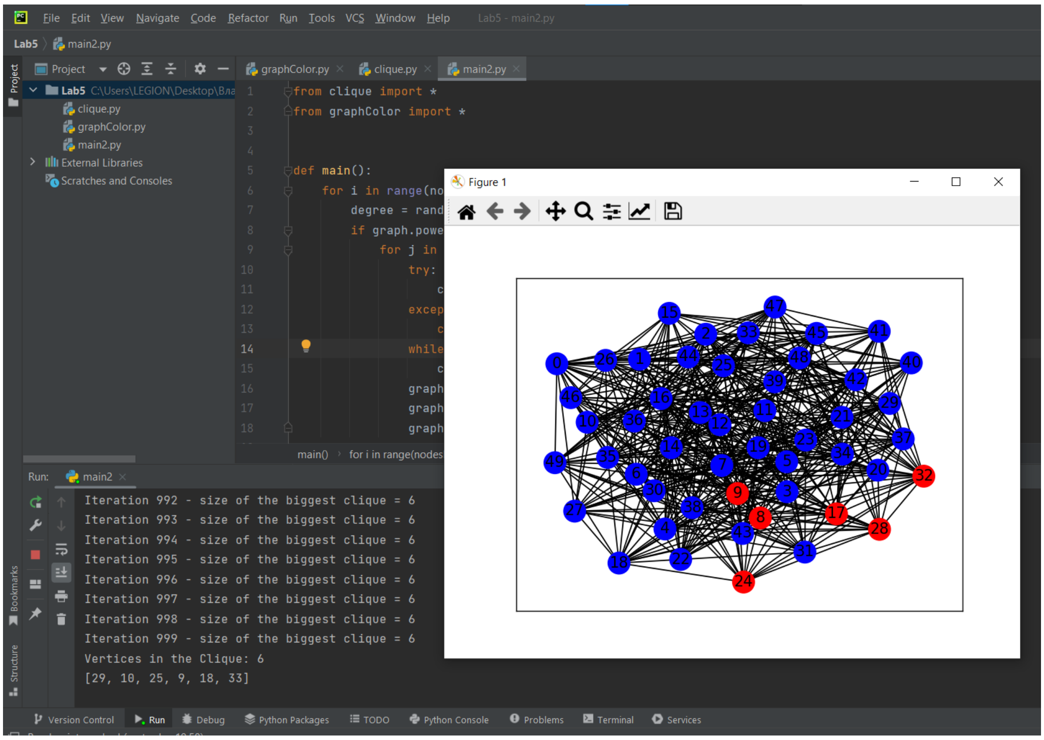


Рисунок 3.1 – Пошук найбільшої кліки в графі розміром 50 і вершинами зі степенями від 20 до 30.

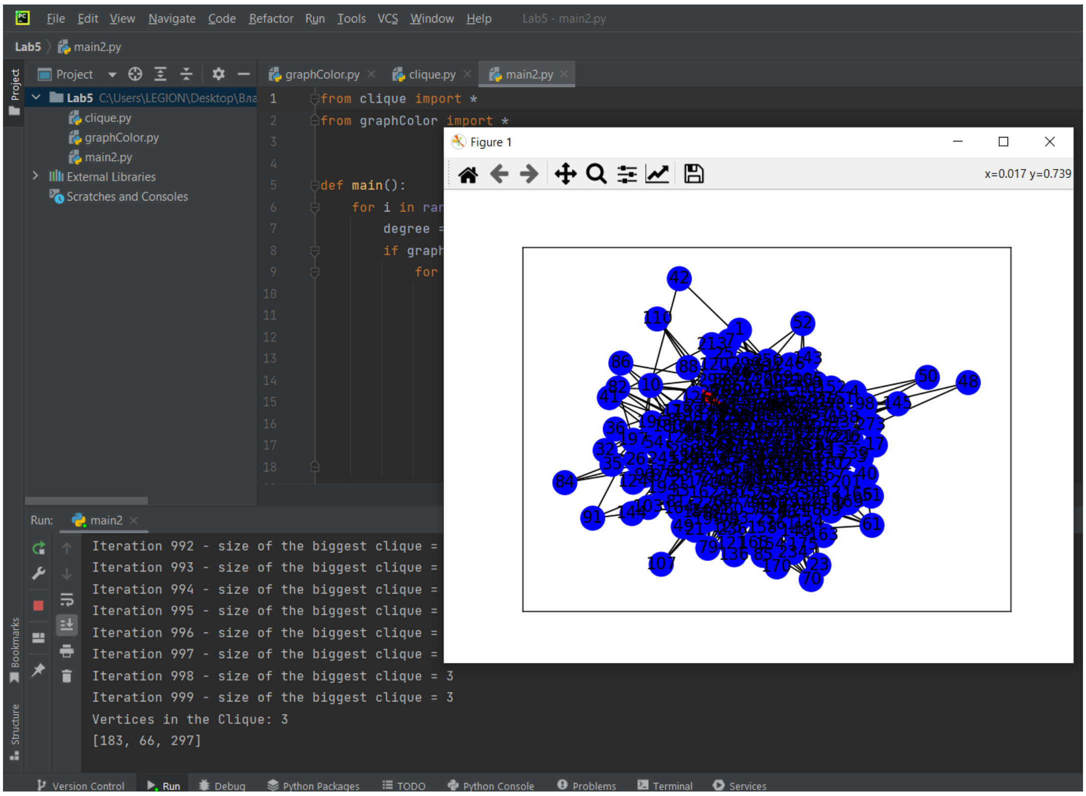


Рисунок 3.2 – Пошук найбільшої кліки в графі розміром 300 й вершинами зі степенями від 2 до 30 (за умовою задачі)

3.3 Тестування алгоритму

Таблиця 3.1 – Залежність рішення від кількості операторів схрещування, операторів мутації, операторів локального покращення – шукаємо оптимальну кількість схрещувань.

Розв’язок шукається для графа у 300 вершин, зі степенями вершин від 40 до 200, адже з варіантом від 2 до 30 за умовою покращення ефективности прослідкувати важко.

К-сть схрещування	К-сть мутацій	К-сть локальних покращень	Ітерація, на якій знайдено кінцевий розв’язок
3	2	2	484 (6->7)
4			385
5			350
6			306
7			296
8			138
9			483

Таблиця 3.2 – Залежність рішення від кількості операторів схрещування, операторів мутації, операторів локального покращення – шукаємо оптимальну кількість мутацій.

К-сть схрещування	К-сть мутацій	К-сть локальних покращень	Ітерація, на якій знайдено кінцевий розв’язок
8	2	2	230
	3		45
	4		140
	5		306
	6		295

Таблиця 3.3 – Залежність рішення від кількості операторів схрещування, операторів мутації, операторів локального покращення – шукаємо оптимальну кількість локальних покращень.

К-сть схрещування	К-сть мутацій	К-сть локальних покращень	Ітерація, на якій знайдено кінцевий розв’язок
8	3	2	286
		3	418
		4	64
		5	229
		6	738

Рисунок 3.3 – Графік залежности ітерацій найкращого рішення від кількости операторів схрещувань.

Рисунок 3.4 – Графік залежности ітерацій найкращого рішення від кількости операторів мутації.

Рисунок 3.5 – Графік залежности ітерацій найкращого рішення від кількости операторів локального покращення.

ВИСНОВОК

В рамках даної лабораторної роботи було вивчено основні підходи розробки метаевристичного генетичного алгоритму для задачі про кліку й опрацьовано методологію підбору прийнятних параметрів алгоритму: операторів схрещення, мутацій і локального покращення. Було спроєктовано та розроблено програмне забезпечення згідно з варіантом і детально описано кроки роботи алгоритму. За допомогою алгоритму було знайдено кліку найбільшого розміру у згенерованому графі. За допомогою тестування було з’ясовано, що на даному прикладі графа найоптимальнішим рішенням є 8 операторів схрещування, 3 мутації та 4 операторів локального покращення.

КРИТЕРІЇ ОЦІНЮВАННЯ

При здачі лабораторної роботи до 11.12.2022 включно максимальний бал дорівнює – 5. Після 11.12.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- покроковий алгоритм – 15%;
- програмна реалізація алгоритму – 50%;
- тестування алгоритму– 30%;
- висновок – 5%.