

**Міністерство освіти і науки України**  
**Національний технічний університет України «Київський політехнічний**  
**інститут імені Ігоря Сікорського»**  
**Факультет інформатики та обчислювальної техніки**  
  
**Кафедра 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-14 Чорній Владислав  
(шифр, прізвище, ім'я, по батькові)

**Перевірив**

Сопов О.О.  
(прізвище, ім'я, по батькові)

Київ 2022

## ЗМІСТ

<b>1</b>	<b>МЕТА ЛАБОРАТОРНОЇ РОБОТИ .....</b>	<b>3</b>
<b>2</b>	<b>ЗАВДАННЯ .....</b>	<b>4</b>
<b>3</b>	<b>ВИКОНАННЯ.....</b>	<b>6</b>
3.1	ПОКРОКОВИЙ АЛГОРИТМ .....	6
3.2	ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ .....	10
3.2.1	<i>Вихідний код.....</i>	<i>10</i>
3.2.2	<i>Приклади роботи .....</i>	<i>15</i>
3.3	ТЕСТУВАННЯ АЛГОРИТМУ .....	17
	<b>ВИСНОВОК .....</b>	<b>18</b>
	<b>КРИТЕРІЇ ОЦІНЮВАННЯ .....</b>	<b>22</b>

## 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 ступенем 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
5	<b>Задача про кліку</b> (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.

	<p>Задача про кліку існує у двох варіантах: у <b>задачі розпізнавання</b> потрібно визначити, чи існує в заданому графі <math>G</math> кліка розміру <math>k</math>, тоді як в <b>обчислювальному варіанті</b> потрібно знайти в заданому графі <math>G</math> кліку максимального розміру або всі максимальні кліки (такі, що не можна збільшити).</p> <p>Застосування:</p> <ul style="list-style-type: none"> <li>– біоінформатика;</li> <li>– електротехніка;</li> </ul>
--	---

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

№	Алгоритми і досліджувані параметри
1	<p><b>Генетичний алгоритм:</b></p> <ul style="list-style-type: none"> <li>- оператор схрещування (мінімум 3);</li> <li>- мутація (мінімум 2);</li> <li>- оператор локального покращення (мінімум 2).</li> </ul>

Таблиця 2.3 – Варіанти задач і алгоритмів

№	Задачі і алгоритми
18	Задача про кліку (обчислювальна задача) + Генетичний алгоритм

### 3 ВИКОНАННЯ

#### 3.1 Покроковий алгоритм

##### Основний алгоритм

1. Створити популяцію зі шляхів створення кліку.
2. Запам'ятати максимальне значення функції.
3. **for**  $i$  **in** range(0, 100000):
  - a. Обрати двох батьків з популяції: турнірний шлях – найкраща особина та випадкова.
  - b. Шляхом кросинговеру генів створити дитину.
  - c. При ймовірності MUTATION виконати мутацію гену.
  - d. **if not** ЦФ дитини **then**:
    - i. **continue**
  - e. **end if**
  - f. Виконати локальне покращення значення функції.
  - g. **if** ЦФ дитини > рекорд **then**:
    - i. Присвоїти рекорду значення функції.
  - h. **end if**
  - i. **if** ідентичного генотипу дитини в популяції нема **then**:
    - i. Додати дитину до популяції.
    - ii. Виділити з популяції найгіршу особину – з найменшою ЦФ.
  - j. **end if**
4. **end for**
5. Кінець

##### Алгоритм обчислювальною задачі про кліку – пошук ЦФ

1. Створити множину вершин у кліку.
2. Почати нумерацію генів.
3. **for** гену **in** хромосомі:
  - a. Запам'ятати номер.

- b. **if** ген існує **then**:
      - i. Додати номер у масив
    - c. **end if**
  - 4. **end for**
  - 5. Перевірити зв'язки між вершинами у кліку.
    - a. **for** вершині **in** кліці:
      - i. **for** суміжній вершині **in** кліці:
        - 1. **if** вершина одна й та ж **then**:
          - a. **continue**
        - 2. **else then**:
          - a. **if** вершини не суміжні **then**:
            - i. Повернути 0
          - b. **end if**
        - 3. **end if**
      - ii. **end for**
    - b. **end for**
    - c. Повернути розмір кліку
  - 6. Кінець.

Оператори схрещування:

Одноточкове (a,b):

1. Обирається випадкова точка
2. Повернути геном a до точки плюс b від точки до кінця.
3. Кінець

Двоточкове (a,b):

1. Обирається випадкова перша точка від 0 до передостаннього числа
2. Обирається випадкова друга точка від першої точки до кінця
3. Повернути геном a від початку до точки b, b від першої точки до другої точки, a від другої точки до кінця.
4. Кінець

Рівномірне (a,b):

1. **For** генів x,y **in** a,b:
  - a. Вибрати випадкове x або y та додати до нової хромосоми
2. **End for**
3. Повернути нову хромосому
4. Кінець

Оператори мутації

Мутація гена(c):

1. Обрати випадковий ген
2. Змінити на протилежний
3. Оновити індивіда
4. Кінець

Мутація проміжку(c):

1. Обирається випадкова перша точка від 0 до передостаннього числа
2. Обирається випадкова друга точка від першої точки до кінця
3. **For** гену **in** range(перша точка, друга точка):
  - a. Змінити ген на протилежний
4. **End for**
5. Оновити індивіда
6. Кінець

Оператори локального покращення

Додавання випадкової вершини

1. Визначити вершини у множині
2. Почати нумерацію генів.
3. **for** гену **in** хромосомі:
  - a. Запам'ятати номер.
  - b. **if** ген існує **then**:
    - i. Додати номер у масив
  - c. **end if**



4. **end for**
5. Перевірити суміжність вершин
  - a. **for** вершині **in** кліці:
    - i. **for** суміжній вершині **in** суміжних вершин:
      1. **if** усі вершини у кліці **in** суміжних вершинах сусіда:
        - a. Додати сусіда до кліку
      2. **End if**
    - ii. **End for**
6. Кінець

Додавання випадкової вершини з евристикою

1. Визначити вершини у кліці
2. Визначити усіх сусідів
3. Відсортувати за зростанням степеня сусідів
4. Знайти сумісного
  - a. **For** сусідів **in** сусідах:
    - i. **if** усі вершини у кліці **in** суміжних вершинах сусіда:
      1. Додати сусіда до кліку
    - ii. **End if**
  - b. **End for**
5. Кінець

## 3.2 Програмна реалізація алгоритму

### 3.2.1 Вихідний код

main.py

```
from graph import *
import random as rand
from individual import Individual
import crossover, mutation, local

def createPopulation(population, graph):
    nodeN = len(graph)
    for i in range(nodeN):
        chromosome = [0 for x in range(nodeN)]
        chromosome[i] = 1
        population.append(Individual(chromosome, graph))
    return 1

def Parents(population):
    a = max(population)
    b = rand.choice(population)
    while a == b:
        b = rand.choice((population))
    return a, b

def removeWorstSol(population):
    minimum = []
    m = population[0].f
    for individ in population:
        if individ.f < m:
            minimum.clear()
            m = individ.f
            minimum.append(individ)
        elif individ.f == m:
            minimum.append(individ)
    population.remove(rand.choice(minimum))

def clique(crossingOverF, mutationF, localeF):
    a = b = c = 100000
    MUTATION = 0.25
    graph = generateGraph(300, 0.9, 30, 2)
    population = []
    record = createPopulation(population, graph)

    for i in range(100000):
        if not i%10000:
            print(f"Iteration: {i}")

        parents = Parents(population)
        kid = crossingOverF(*parents)
```

```

    if rand.random() <= MUTATION:
        mutationF(kid)

    if not kid.f:
        continue

    localeF(kid, graph)

    if kid.f > record:
        record = kid.f
        print(f"Iteration: {i}, f:{record}")
        if record == 15:
            a = i
        if record == 16:
            b = i
        if record >= 17:
            c = i
            break

    if kid not in population:
        population += kid,
        removeWorstSol(population)
    print(f"Best solution: {record}")
    return a, b, c

if __name__ == '__main__':
    clique(crossingover.twoPoint, mutation.interval, local.improvementHeuristic)

```

## individual.py

```

class Individual:
    def __init__(self, chromosome, graph):
        self.chromosome = chromosome
        self.graph = graph
        self.f = self.maxClique(chromosome)

    def maxClique(self, chromosome):
        nodes = []
        i = 0
        for gene in chromosome:
            i += 1
            if gene:
                nodes.append(i)
        for node in nodes:
            for adjacentNode in nodes:
                if node == adjacentNode:
                    continue
            else:
                if adjacentNode not in self.graph[node]:
                    return 0
        return len(nodes)

    def update(self, chromosome):

```

```

self.chromosome = chromosome
self.f = self.maxClique(chromosome)

def __lt__(self, other):
    return self.f.__lt__(other.f)

def __gt__(self, other):
    return self.f.__gt__(other.f)

def __le__(self, other):
    return self.f.__le__(other.f)

def __ge__(self, other):
    return self.f.__ge__(other.f)

def __repr__(self):
    return f"{self.f}"

def __eq__(self, other):
    return self.chromosome == other.chromosome

```

#### crossingover.py

```

import random as rand
from individual import Individual

def onePoint(a, b):
    graph = a.graph
    a, b = a.chromosome, b.chromosome
    point = rand.randint(0, len(a)-1)
    newChromosome = a[:point+1] + b[point+1:]
    return Individual(newChromosome, graph)

def twoPoint(a, b):
    graph = a.graph
    a, b = a.chromosome, b.chromosome
    point1 = rand.randint(0, len(a)//2)
    point2 = rand.randint(point1, len(b) - 1)
    newChromosome = a[:point1 + 1] + b[point1 + 1:point2+1] + a[point2+1:]
    return Individual(newChromosome, graph)

def uniform(a, b):
    chromosome = []
    graph = a.graph
    a, b = a.chromosome, b.chromosome
    for x, y in zip(a, b):
        chromosome += rand.choice([x, y]),
    return Individual(chromosome, graph)

```

#### mutation.py

```

import random as rand

```

```

def onePoint(c):
    i = rand.randint(0, len(c.chromosome)-1)
    chromosome = list(c.chromosome)
    chromosome[i] = 0 if chromosome[i] else 1
    c.update(chromosome)

def interval(c):
    chromosome = list(c.chromosome)
    point1 = rand.randint(0, len(chromosome)-2)
    point2 = rand.randint(point1, len(chromosome))
    for i in range(point1, point2):
        chromosome[i] = 0 if chromosome[i] else 1
    c.update(chromosome)

```

local.py

```
import random as rand
```

```

def improvementRand(c, graph):
    nodes = []
    i = 0
    for gene in c.chromosome:
        i+=1
        if gene:
            nodes.append(i)

    rand.shuffle(nodes)
    for node in nodes:
        adjacentNodes = graph[node]
        rand.shuffle(adjacentNodes)
        for adjacentNode in adjacentNodes:
            if adjacentNode in nodes:
                continue

            if set(nodes) <= set(graph[adjacentNode]):
                chromosome = list(c.chromosome)
                chromosome[adjacentNode-1] = 1
                c.update(chromosome)
    return

```

```

def improvementHeuristic(c, graph):
    nodes = []
    for i, gene in enumerate(c.chromosome):
        if gene:
            nodes.append(i + 1)

    rand.shuffle(nodes)
    adjacentNodes = []
    for node in nodes:

```

```

    adjacentNodes += graph[node]

adjacentNodes = list(set(adjacentNodes))
rand.shuffle(adjacentNodes)

for adjacentNode in sorted(adjacentNodes, key=lambda x: len(graph[x])):
    if adjacentNode in nodes:
        continue
    if set(nodes) <= set(graph[adjacentNode]):
        chromosome = list(c.chromosome)
        chromosome[adjacentNode-1] = 1
        c.update(chromosome)
        return

tests.py
from main import clique
import crossover, mutation, local

def testCrossover():
    funcs = { crossover.onePoint: (0,0,0,0), crossover.twoPoint:
(0,0,0,0), crossover.uniform: (0,0,0,0) }
    for func in funcs:
        print(f"Function: Crossing over - {func.__name__}")
        a,b,c,stuck = 0,0,0,0
        for i in range(10):
            print(f"\nTest number {i+1}")
            result = clique(func, mutation.onePoint, local.improvementRand)
            a += result[0]
            b += result[1]
            c += result[2]
            if result[2] == 100000:
                stuck += 1

        funcs[func] = (a / 10, b / 10, c / 10, stuck)

    for func, results in funcs.items():
        print(func.__name__, ":", results[3], ":", results[0], results[1], results[2])

def testMutation():
    funcs = { mutation.onePoint: (0,0,0,0), mutation.interval: (0,0,0,0) }
    for func in funcs:
        print(f"Function: Mutation - {func.__name__}")
        a,b,c,stuck = 0,0,0,0
        for i in range(10):
            print(f"\nTest number {i+1}")
            result = clique(crossover.twoPoint, func, local.improvementRand)
            a += result[0]
            b += result[1]
            c += result[2]
            if result[2] == 100000:
                stuck += 1

```

```

funcs[func] = (a / 10, b / 10, c / 10, stuck)

for func, results in funcs.items():
    print(func.__name__, ":", results[3], ":", results[0], results[1], results[2])

def testLocal():
    funcs = {local.improvementRand: (0,0,0,0), local.improvementHeuristic: (0,0,0,0)}
    for func in funcs:
        print(f"Function: Local - {func.__name__}")
        a,b,c,stuck = 0,0,0,0
        for i in range(10):
            print(f"\nTest number {i+1}")
            result = clique(crossingover.twoPoint, mutation.interval, func)
            a += result[0]
            b += result[1]
            c += result[2]
            if result[2] == 100000:
                stuck += 1

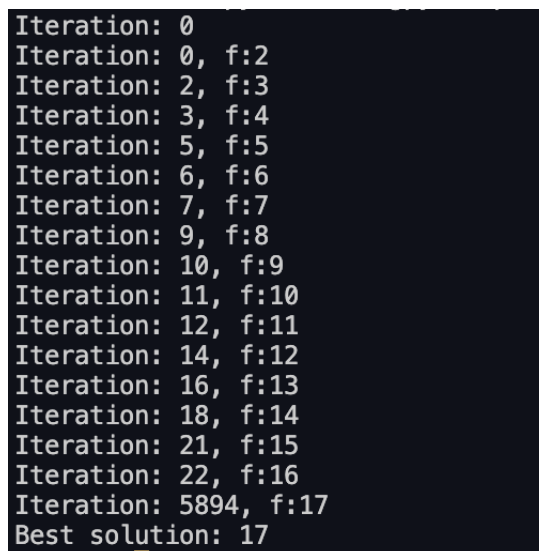
    funcs[func] = (a / 10, b / 10, c / 10, stuck)

for func, results in funcs.items():
    print(func.__name__, ":", results[3], ":", results[0], results[1], results[2])
if __name__ == "__main__":
    # testCrossingOver()
    # testMutation()
    testLocal()

```

### 3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.



```

Iteration: 0
Iteration: 0, f:2
Iteration: 2, f:3
Iteration: 3, f:4
Iteration: 5, f:5
Iteration: 6, f:6
Iteration: 7, f:7
Iteration: 9, f:8
Iteration: 10, f:9
Iteration: 11, f:10
Iteration: 12, f:11
Iteration: 14, f:12
Iteration: 16, f:13
Iteration: 18, f:14
Iteration: 21, f:15
Iteration: 22, f:16
Iteration: 5894, f:17
Best solution: 17

```

Рисунок 3.1 – Приклад роботи графа для випадкової програми

```
Iteration: 0  
Iteration: 2, f:2  
Iteration: 3, f:3  
Iteration: 4, f:4  
Iteration: 11, f:5  
Iteration: 12, f:6  
Iteration: 18, f:7  
Iteration: 21, f:8  
Iteration: 22, f:9  
Iteration: 23, f:10  
Iteration: 29, f:11  
Iteration: 32, f:12  
Iteration: 33, f:13  
Iteration: 34, f:14  
Iteration: 35, f:15  
Iteration: 10000  
Iteration: 11068, f:16  
Iteration: 12373, f:17  
Best solution: 17
```

Рисунок 3.2 – Приклад роботи графа для випадкової програми



### 3.3 Тестування алгоритму

Було проведено тестування кожної з трьох груп операторів, кожного оператора в усіх групах.

1. Оператори схрещування:

- а. Одноточкове
- б. Двоточкове
- с. Рівномірне

2. Мутації:

- а. Мутація випадкового гену
- б. Мутація випадкового проміжку

3. Оператори локального покращення:

- а. Додавання випадкової вершини
- б. Додавання за допомогою евристики

Критерії зупинки алгоритму: ЦФ = 17 – оптимально-максимальне число, 100000 ітерацій.

Перевіряючи оператори схрещування, зафіксуємо оператор мутації випадкового гену та оператор локального покращення додаванням випадкової вершини. Заповнимо таблицю незнаходжень оптимального числа та середніх кількостей ітерацій знаходження ЦФ=15,16,17.

Таблиця 3.1 – Показники тестування операторів схрещення

Назва оператора	Кількість незнаходжень оптимального числа	Iter(15)	Iter(16)	Iter(17)
Одноточкове	6	34809	61321	68009
Двоточкове	7	17246	37841	73749
Рівномірне	7	41471	55243	72342

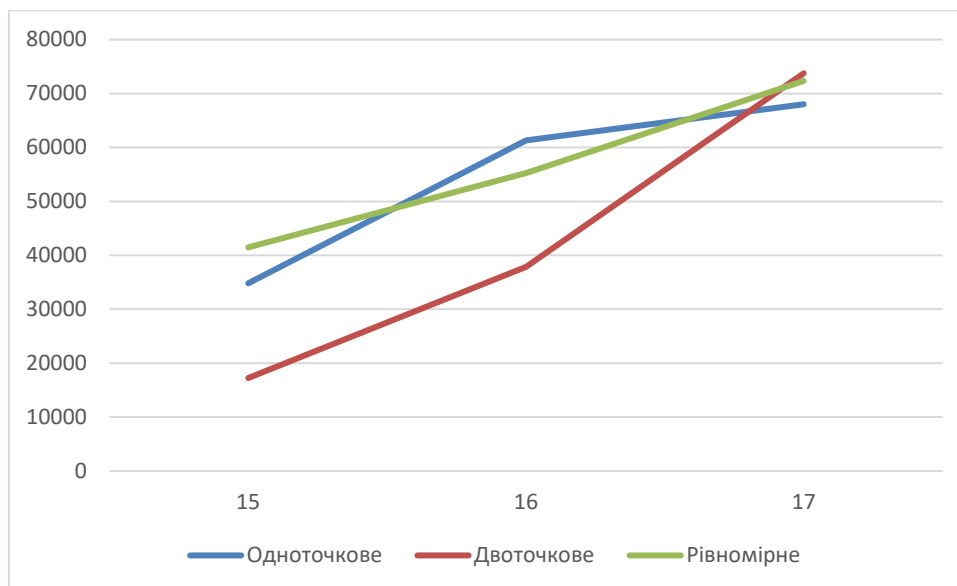


Рисунок 3.3 – Показники тестування операторів схрещення

За меншою кількістю незнаходжень та найменшою кількістю ітерацій для знаходження оптимально-максимально числа функції, найефективніший оператор – одноточкове. Зафіксуємо цю функцію з оператором випадкового локального покращення. Заповнимо таблицю незнаходжень оптимального числа та середніх кількостей ітерацій знаходження ЦФ=15,16,17.

Таблиця 3.2 – Показники тестування операторів мутації

Назва оператора	Кількість незнаходжень оптимального числа	Iter(15)	Iter(16)	Iter(17)
Гена	4	11902	27270	53415
Проміжку	7	22384	70014	70015

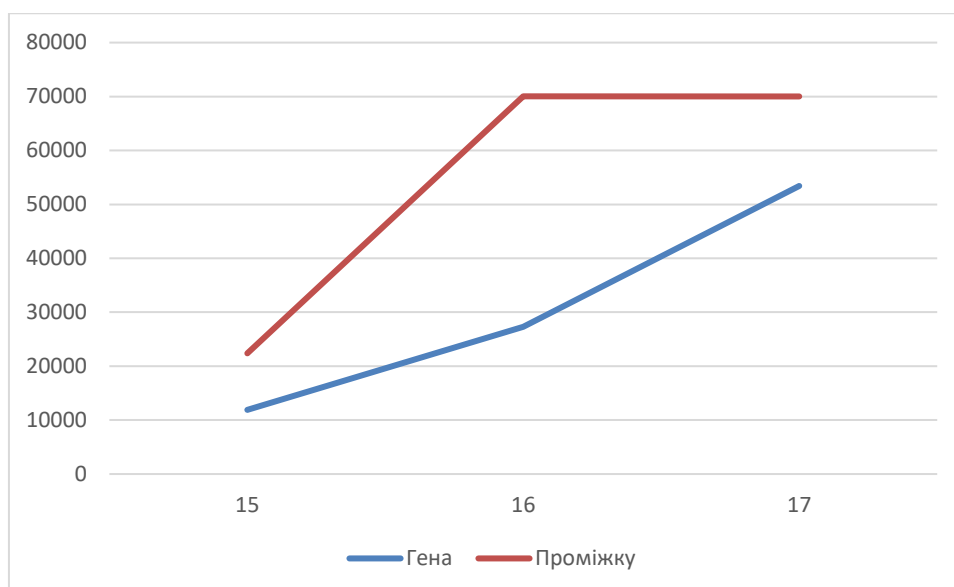


Рисунок 3.4 – Показники тестування операторів мутації

За меншою кількістю незнаходжень та найменшою кількістю ітерацій для знаходження оптимально-максимально числа функції, найефективніший оператор – мутація гена. Зафіксуємо цю функцію з оператором одноточкового скрещення. Заповнимо таблицю незнаходжень оптимального числа та середніх кількостей ітерацій знаходження ЦФ=15,16,17.

Таблиця 3.3 – Показники тестування операторів локального покращення

Назва оператора	Кількість незнаходжень оптимального числа	Iter(15)	Iter(16)	Iter(17)
Гена	9	28485	55547	92298
Проміжку	5	22460	47105	60587

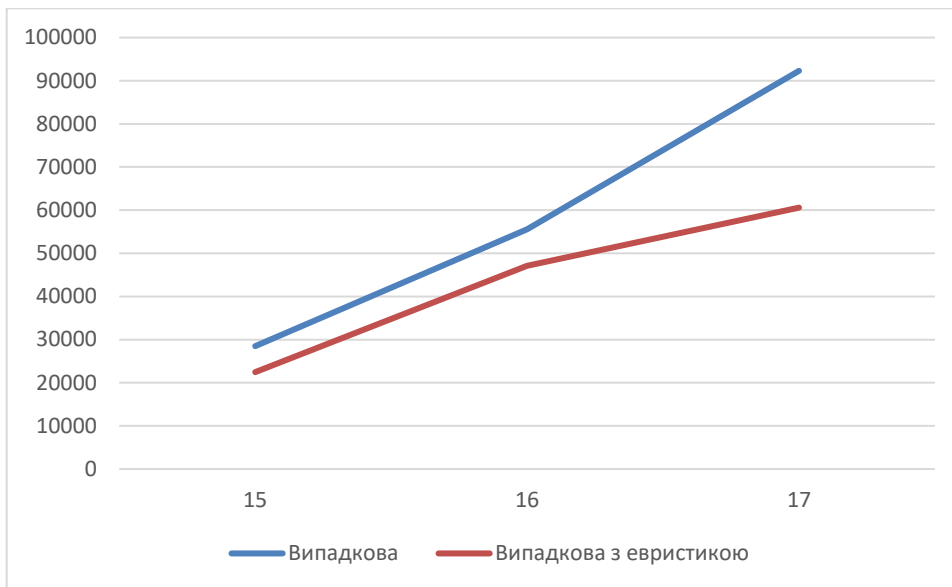


Рисунок 3.4 – Показники тестування операторів мутації

За меншою кількістю незнаходжень та найменшою кількістю ітерацій для знаходження оптимально-максимально числа функції, найефективніший оператор – покращення з евристикою.

За проведеними тестуваннями отримали найоптимальнішу комбінацію алгоритмів: одноточкове скрещення, мутація випадкового гену, оператор локального покращення, де до кліку додається вершина з найбільшим степенем.

## ВИСНОВОК

В рамках даної лабораторної роботи було розроблено алгоритм вирішення генетичним алгоритмом обчислювальної задачі про кліку. Було сформульовано покровий вигляд та виконано його програмну реалізацію на мові програмування Python.

Було проведено тестування різних операторів кожною з груп: оператори схрещування, мутації та локального покращення. Найоптимальнішою комбінацією виявилися наступна: одноточкове схрещення, мутація випадкового гену, локальне покращення з додаванням суміжної вершини з найбільшим степенем. Під час досліджень була зроблений висновок, що рівномірне схрещування та мутація випадкового проміжку генів – є не оптимальними.

Отже, генетичний алгоритм є доволі ефективним метаевристичним алгоритмом розв'язування задач.

## 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%.