**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 № 4 з дисципліни

«Проектування алгоритмів»

„**Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.1**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-14 Прокопенко Олексій*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc51260917)

[2 Завдання 4](#_Toc51260918)

[3 Виконання 10](#_Toc51260919)

[3.1 Програмна реалізація алгоритму 10](#_Toc51260920)

[3.1.1 Вихідний код 10](#_Toc51260921)

[3.1.2 Приклади роботи 10](#_Toc51260922)

[3.2 Тестування алгоритму 11](#_Toc51260923)

[3.2.1 Значення цільової функції зі збільшенням кількості ітерацій 11](#_Toc51260924)

[3.2.2 Графіки залежності розв'язку від числа ітерацій 11](#_Toc51260925)

[Висновок 12](#_Toc51260926)

[Критерії оцінювання 13](#_Toc51260927)

# Мета лабораторної роботи

Мета роботи – вивчити основні підходи формалізації метаеврестичних алгоритмів і вирішення типових задач з їхньою допомогою.

# Завдання

Згідно варіанту, розробити алгоритм вирішення задачі і виконати його програмну реалізацію на будь-якій мові програмування.

Задача, алгоритм і його параметри наведені в таблиці 2.1.

Зафіксувати якість отриманого розв'язку (значення цільової функції) після кожних 20 ітерацій до 1000 і побудувати графік залежності якості розв'язку від числа ітерацій.

Зробити узагальнений висновок.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |
| --- | --- |
| **№** | **Задача і алгоритм** |
| 13 | Задача про рюкзак (місткість P=250, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 25 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування одноточковий 30% і 70%, мутація з ймовірністю 5% два випадкові гени міняються місцями). Розробити власний оператор локального покращення. |

# Виконання

## Програмна реалізація алгоритму

### Вихідний код

import random

from typing import List

from unittest import TestCase, main

class Item:

def \_\_init\_\_(self, name, weight, value):

self.name = name

self.weight = weight

self.value = value

class Individual:

def \_\_init\_\_(self, bits: List[int]):

self.bits = bits

def \_\_str\_\_(self):

return repr(self.bits)

def \_\_hash\_\_(self):

return hash(str(self.bits))

def fitness(self) -> float:

total\_value = sum([

bit \* item.value

for item, bit in zip(items, self.bits)

])

total\_weight = sum([

bit \* item.weight

for item, bit in zip(items, self.bits)

])

if total\_weight <= MAX\_KNAPSACK\_WEIGHT:

return total\_value

return 0

MAX\_KNAPSACK\_WEIGHT = 250

CROSSOVER\_RATE = 0.70

MUTATION\_RATE = 0.05

REPRODUCTION\_RATE = 0.30

items = []

for i in range(100):

it = Item("Item" + str(i),random.randint(1, 25), random.randint(2, 30))

print(it.name + " " + str(it.weight) + " " + str(it.value))

items.append(it)

def generate\_initial\_population(count=100) -> List[Individual]:

population = set()

while len(population) != count:

bits = [

random.choice([0, 0, 0, 0, 1])

for \_ in items

]

population.add(Individual(bits))

return list(population)

def selection(population: List[Individual]) -> List[Individual]:

parents = []

random.shuffle(population)

if population[0].fitness() > population[1].fitness():

parents.append(population[0])

else:

parents.append(population[1])

if population[2].fitness() > population[3].fitness():

parents.append(population[2])

else:

parents.append(population[3])

return parents

def crossover(parents: List[Individual]) -> List[Individual]:

N = len(items)

child1 = parents[0].bits[:N//2] + parents[1].bits[N//2:]

child2 = parents[0].bits[N//2:] + parents[1].bits[:N//2]

return [Individual(child1), Individual(child2)]

def mutate(individuals: List[Individual]) -> List[Individual]:

for individual in individuals:

for i in range(len(individual.bits)):

if random.random() < MUTATION\_RATE:

individual.bits[i] = ~individual.bits[i]

def next\_generation(population: List[Individual]) -> List[Individual]:

next\_gen = []

while len(next\_gen) < len(population):

children = []

parents = selection(population)

if random.random() < REPRODUCTION\_RATE:

children = parents

else:

if random.random() < CROSSOVER\_RATE:

children = crossover(parents)

if random.random() < MUTATION\_RATE:

mutate(children)

next\_gen.extend(children)

return next\_gen[:len(population)]

def print\_generation(population: List[Individual]):

for individual in population:

print(individual.bits, individual.fitness())

print()

print("Average fitness", sum([x.fitness() for x in population])/len(population))

print("-" \* 32)

def average\_fitness(population: List[Individual]) -> float:

return sum([i.fitness() for i in population]) / len(population)

def solve\_knapsack() -> Individual:

population = generate\_initial\_population()

avg\_fitnesses = []

for \_ in range(500):

avg\_fitnesses.append(average\_fitness(population))

population = next\_generation(population)

population = sorted(population, key=lambda i: i.fitness(), reverse=True)

return population[0]

def get\_total\_weigth(solution):

weigth = 0

for i in range(100):

if(solution.bits[i] == 1):

it = items[i]

weigth += it.weight

return weigth

def print\_solution(solution):

print("Pack: ")

for i in range(100):

if(solution.bits[i] == 1):

it = items[i]

print(it.name + " " + str(it.weight) + " " + str(it.value))

print("Total price:" + str(solution.fitness()))

print("Total weigth:" + str(get\_total\_weigth(solution)))

def solve\_problem():

for \_ in range(10):

solution = solve\_knapsack()

if get\_total\_weigth(solution) < 250 and get\_total\_weigth(solution) > 235:

print\_solution(solution)

return

if \_\_name\_\_ == '\_\_main\_\_':

solve\_problem()

#tests

class HeneticTests(TestCase):

def test\_generate\_100\_people\_in\_population(self):

count = 100

length = len(generate\_initial\_population(count))

self.assertEqual(length, count)

def test\_selection\_return\_2\_persons(self):

population = generate\_initial\_population(100)

self.assertEqual(len(selection(population)), 2)

def test\_next\_generation\_return\_100\_persons(self):

population = generate\_initial\_population(100)

self.assertEqual(len(next\_generation(population)), 100)

if \_\_name\_\_ == '\_\_main\_\_':

main()

### Приклади роботи

На рисунку 3.1 показаний приклади роботи програми.

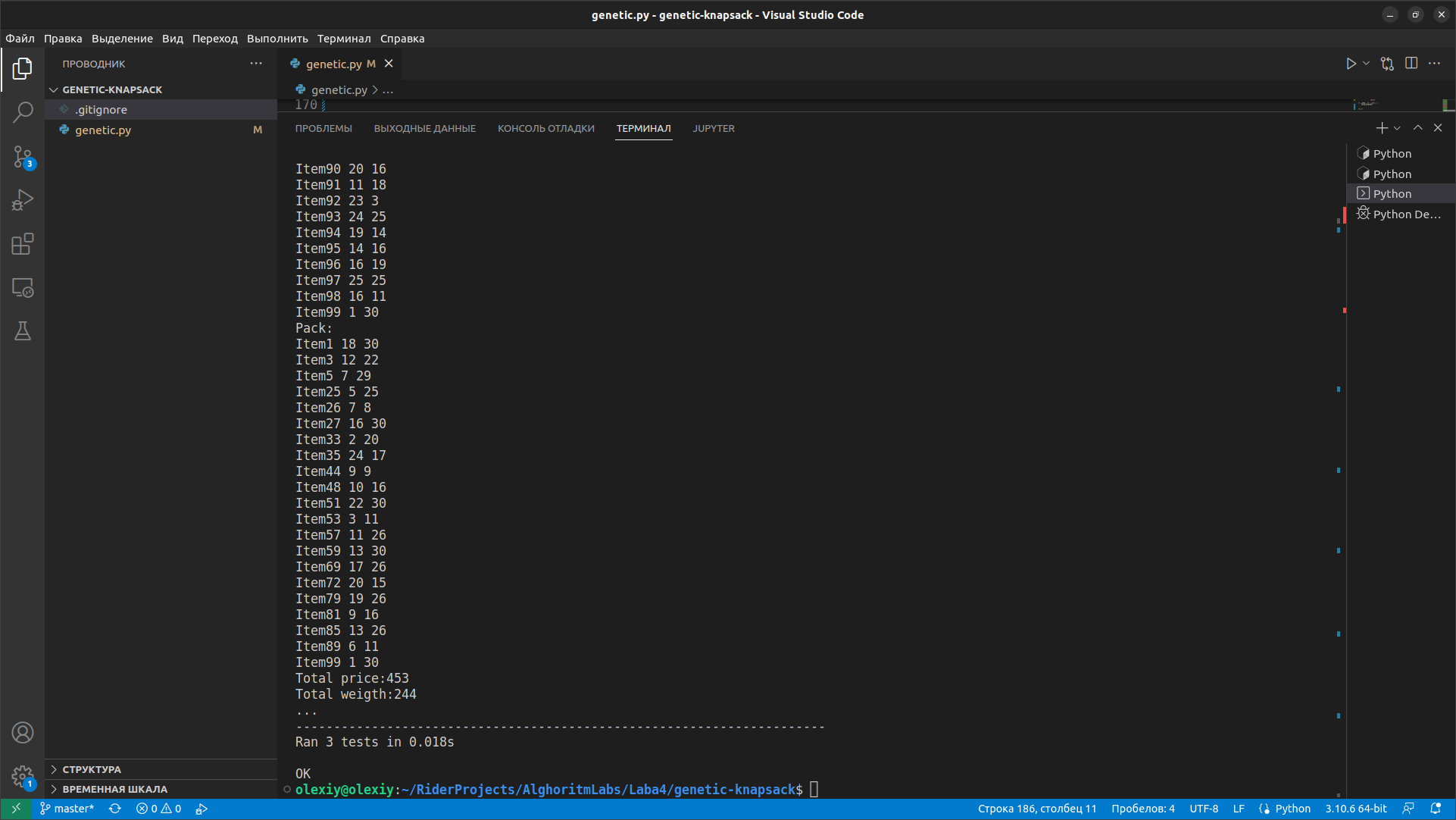
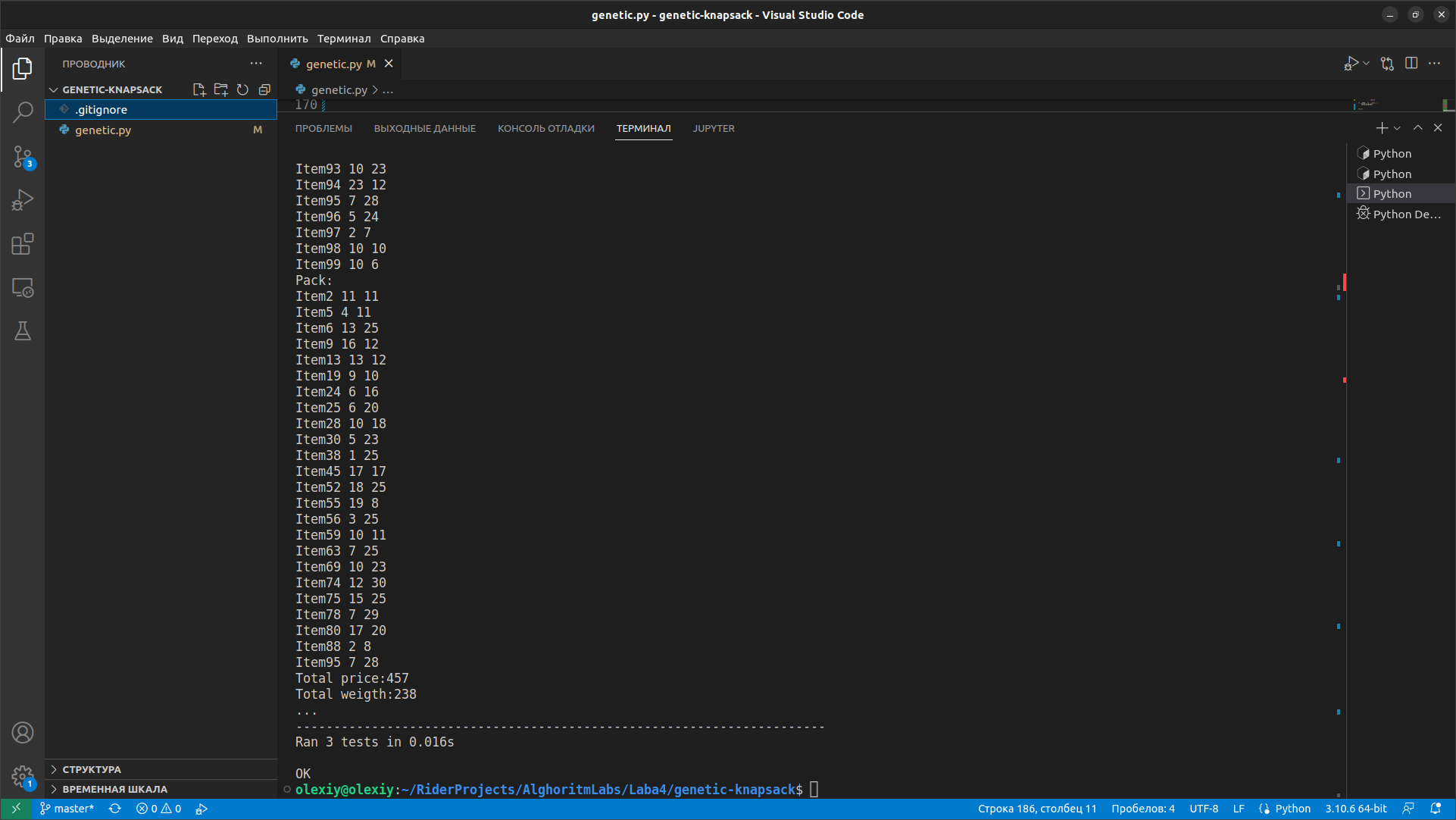


Рисунок 3.1

## Тестування алгоритму

### Значення цільової функції зі збільшенням кількості ітерацій

У таблиці 3.1 наведено значення цільової функції зі збільшенням кількості ітерацій.

|  |  |
| --- | --- |
| Вага всього рюкзака | Загальна ціна |
| 235 | 338 |
| 243 | 346 |
| 236 | 356 |
| 246 | 362 |
| 246 | 376 |
| 237 | 356 |
| 236 | 356 |
| 249 | 369 |
| 249 | 369 |

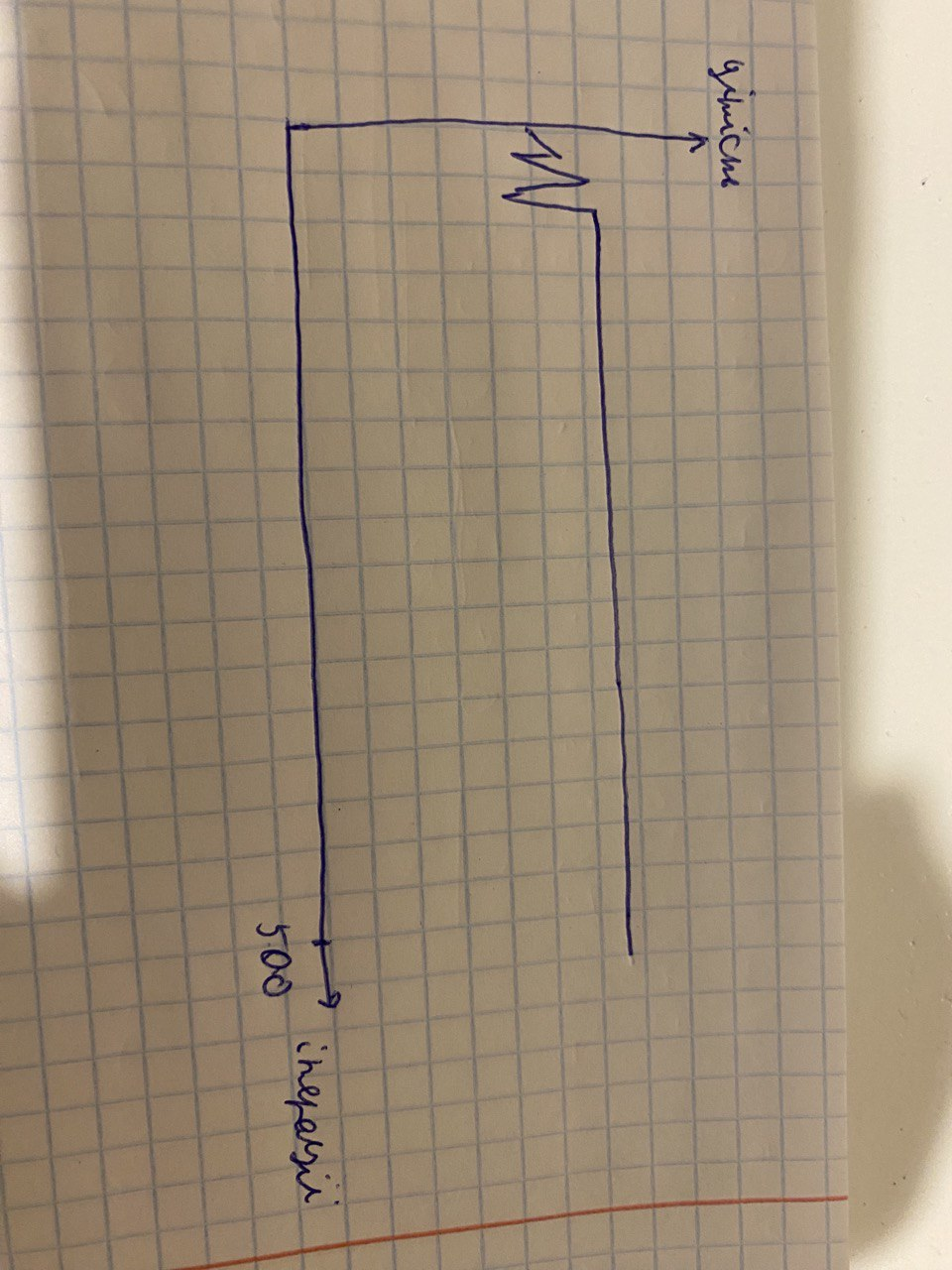
В цьому випадку алгоритм не зміг знайти найефективніше рішення, бо з вагою рюкзака в 246 його цінність буде більшою ніж з вагою 249, яку алгоритм визначив

|  |  |
| --- | --- |
| Вага всього рюкзака | Загальна ціна |
| 239 | 323 |
| 249 | 323 |
| 250 | 383 |
| 239 | 401 |
| 250 | 383 |
| 250 | 383 |
| 249 | 410 |
| 249 | 410 |

### Графіки залежності розв'язку від числа ітерацій

На рисунку 3.3 наведений графік, який показує якість отриманого розв'язку.

Рисунок 3.3 – Графіки залежності розв'язку від числа ітерацій



Висновок

В рамках даної лабораторної роботи було реалізовано генетичний алгоритм, за допомогою якого було вирішено проблему з рюкзаком. Було згереновано 100 предметів відповідно до умови. Для генетичного алгоритму було створено 100 особистостей, яким випадковим чином “покладено в рюкзак” випадкові предмети. Далі, під час роботи алгоритму, особистості з найціннішим вмістом рюкзака (найбільше значення фітнес-функції) вибиралися і випадково зхрещувалась, або мутували, потім створювалося нове покоління на основі них, це відбувалося поки не була знайдена особа з найвищим значенням фітнес функції.

Критерії оцінювання

При здачі лабораторної роботи до 27.11.2021 включно максимальний бал дорівнює – 5. Після 27.11.2021 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* програмна реалізація алгоритму – 75%;
* тестування алгоритму– 20%;
* висновок – 5%.