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) *ІП-21 Сергієнко Сергій* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Головченко М.М.* _____
(прізвище, ім'я, по батькові)

Київ 2023

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ 3

2 ЗАВДАННЯ 4

3 ВИКОНАННЯ 10

3.1 Програмна реалізація алгоритму 10

3.1.1 Вихідний код 10

3.1.2 Приклади роботи 16

3.2 Тестування алгоритму 17

3.2.1 Значення цільової функції зі збільшенням кількості ітерацій . 17

3.2.2 Графіки залежності розв'язку від числа ітерацій 19

ВИСНОВОК 20

КРИТЕРІЇ ОЦІНЮВАННЯ 21

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

2 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**
1	Задача про рюкзак (місткість P=250, 100 предметів, цінність предметів від 2 до 20 (випадкова), вага від 1 до 10 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування одноточковий по 50 генів, мутація з ймовірністю 5% змінюємо тільки 1 випадковий ген). Розробити власний оператор локального покращення.
2	Задача комівояжера (100 вершин, відстань між вершинами випадкова від 5 до 50), мурашиний алгоритм ($\alpha = 2$, $\beta = 4$, $\rho = 0{,}4$, Lmin знайти жадібним алгоритмом, кількість мурах M = 30, починають маршрут в різних випадкових вершинах).
3	Задача розфарбовування графу (200 вершин, степінь вершини не більше 20, але не менше 1), бджолиний алгоритм ABC (число бджіл 30 із них 2 розвідники).
4	Задача про рюкзак (місткість P=200, 100 предметів, цінність предметів від 2 до 20 (випадкова), вага від 1 до 10 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування двоточковий порівну генів, мутація з ймовірністю 10% змінюємо тільки 1 випадковий ген). Розробити

	власний оператор локального покращення.
5	Задача комівояжера (150 вершин, відстань між вершинами випадкова від 5 до 50), мурашиний алгоритм (α = 2, β = 3, ρ = 0,4, Lmin знайти жадібним алгоритмом, кількість мурах M = 35, починають маршрут в різних випадкових вершинах).
6	Задача розфарбовування графу (250 вершин, степінь вершини не більше 25, але не менше 2), бджолиний алгоритм ABC (число бджіл 35 із них 3 розвідники).
7	Задача про рюкзак (місткість P=150, 100 предметів, цінність предметів від 2 до 10 (випадкова), вага від 1 до 5 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування рівномірний, мутація з ймовірністю 5% два випадкові гени міняються місцями). Розробити власний оператор локального покращення.
8	Задача комівояжера (200 вершин, відстань між вершинами випадкова від 0(перехід заборонено) до 50), мурашиний алгоритм (α = 3, β = 2, ρ = 0,3, Lmin знайти жадібним алгоритмом, кількість мурах M = 45, починають маршрут в різних випадкових вершинах).
9	Задача розфарбовування графу (150 вершин, степінь вершини не більше 30, але не менше 1), бджолиний алгоритм ABC (число бджіл 25 із них 3 розвідники).
10	Задача про рюкзак (місткість P=150, 100 предметів, цінність предметів від 2 до 10 (випадкова), вага від 1 до 5 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування рівномірний, мутація з ймовірністю 10% два випадкові гени міняються місцями). Розробити власний оператор локального покращення.
11	Задача комівояжера (250 вершин, відстань між вершинами випадкова від 0(перехід заборонено) до 50), мурашиний алгоритм (α = 2, β = 4, ρ =

	0,6, Lmin знайти жадібним алгоритмом, кількість мурах M = 45, починають маршрут в різних випадкових вершинах).
12	Задача розфарбовування графу (300 вершин, степінь вершини не більше 30, але не менше 1), бджолиний алгоритм ABC (число бджіл 60 із них 5 розвідники).
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.
14	Задача комівояжера (250 вершин, відстань між вершинами випадкова від 1 до 40), мурашиний алгоритм (α = 4, β = 2, ρ = 0,3, Lmin знайти жадібним алгоритмом, кількість мурах M = 45 (10 з них дикі, обирають випадкові напрямки), починають маршрут в різних випадкових вершинах).
15	Задача розфарбовування графу (100 вершин, степінь вершини не більше 20, але не менше 1), класичний бджолиний алгоритм (число бджіл 30 із них 3 розвідники).
16	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 двоточковий 30%, 40% і 30%, мутація з ймовірністю 10% два випадкові гени міняються місцями). Розробити власний оператор локального покращення.
17	Задача комівояжера (200 вершин, відстань між вершинами випадкова від 1 до 40), мурашиний алгоритм (α = 2, β = 4, ρ = 0,7, Lmin знайти жадібним алгоритмом, кількість мурах M = 45 (15 з них дикі, обирають випадкові напрямки), починають маршрут в різних випадкових

	вершинах).
18	Задача розфарбовування графу (300 вершин, степінь вершини не більше 50, але не менше 1), класичний бджолиний алгоритм (число бджіл 60 із них 5 розвідники).
19	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 триточковий 25%, мутація з ймовірністю 5% два випадкові гени міняються місцями). Розробити власний оператор локального покращення.
20	Задача комівояжера (200 вершин, відстань між вершинами випадкова від 1 до 40), мурашиний алгоритм ($\alpha = 3$, $\beta = 2$, $\rho = 0,7$, Lmin знайти жадібним алгоритмом, кількість мурах M = 45 (10 з них елітні, подвійний феромон), починають маршрут в різних випадкових вершинах).
21	Задача розфарбовування графу (200 вершин, степінь вершини не більше 30, але не менше 1), класичний бджолиний алгоритм (число бджіл 40 із них 2 розвідники).
22	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 триточковий 25%, мутація з ймовірністю 5% змінюємо тільки 1 випадковий ген). Розробити власний оператор локального покращення.
23	Задача комівояжера (300 вершин, відстань між вершинами випадкова від 1 до 60), мурашиний алгоритм ($\alpha = 3$, $\beta = 2$, $\rho = 0,6$, Lmin знайти жадібним алгоритмом, кількість мурах M = 45 (15 з них елітні, подвійний феромон), починають маршрут в різних випадкових вершинах).

24	Задача розфарбовування графу (400 вершин, степінь вершини не більше 50, але не менше 1), класичний бджолиний алгоритм (число бджіл 70 із них 10 розвідники).
25	Задача про рюкзак (місткість P=250, 100 предметів, цінність предметів від 2 до 20 (випадкова), вага від 1 до 10 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування одноточковий по 50 генів, мутація з ймовірністю 5% змінюємо тільки 1 випадковий ген). Розробити власний оператор локального покращення.
26	Задача комівояжера (100 вершин, відстань між вершинами випадкова від 5 до 50), мурашиний алгоритм ($\alpha = 2$, $\beta = 4$, $\rho = 0{,}4$, Lmin знайти жадібним алгоритмом, кількість мурах M = 30, починають маршрут в різних випадкових вершинах).
27	Задача розфарбовування графу (200 вершин, степінь вершини не більше 20, але не менше 1), бджолиний алгоритм ABC (число бджіл 30 із них 2 розвідники).
28	Задача про рюкзак (місткість P=200, 100 предметів, цінність предметів від 2 до 20 (випадкова), вага від 1 до 10 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування двоточковий порівну генів, мутація з ймовірністю 10% змінюємо тільки 1 випадковий ген). Розробити власний оператор локального покращення.
29	Задача комівояжера (150 вершин, відстань між вершинами випадкова від 5 до 50), мурашиний алгоритм ($\alpha = 2$, $\beta = 3$, $\rho = 0{,}4$, Lmin знайти жадібним алгоритмом, кількість мурах M = 35, починають маршрут в різних випадкових вершинах).
30	Задача розфарбовування графу (250 вершин, степінь вершини не більше 25, але не менше 2), бджолиний алгоритм ABC (число бджіл 35 із них 3 розвідники).

31	Задача про рюкзак (місткість P=250, 100 предметів, цінність предметів від 2 до 20 (випадкова), вага від 1 до 10 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування одноточковий по 50 генів, мутація з ймовірністю 5% змінюємо тільки 1 випадковий ген). Розробити власний оператор локального покращення.
32	Задача комівояжера (100 вершин, відстань між вершинами випадкова від 5 до 50), мурашиний алгоритм ($\alpha = 2$, $\beta = 4$, $\rho = 0,4$, Lmin знайти жадібним алгоритмом, кількість мурах M = 30, починають маршрут в різних випадкових вершинах).
33	Задача розфарбовування графу (200 вершин, степінь вершини не більше 20, але не менше 1), бджолиний алгоритм ABC (число бджіл 30 із них 2 розвідники).
34	Задача про рюкзак (місткість P=200, 100 предметів, цінність предметів від 2 до 20 (випадкова), вага від 1 до 10 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування двоточковий порівну генів, мутація з ймовірністю 10% змінюємо тільки 1 випадковий ген). Розробити власний оператор локального покращення.
35	Задача комівояжера (150 вершин, відстань між вершинами випадкова від 5 до 50), мурашиний алгоритм ($\alpha = 2$, $\beta = 3$, $\rho = 0,4$, Lmin знайти жадібним алгоритмом, кількість мурах M = 35, починають маршрут в різних випадкових вершинах).

3 ВИКОНАННЯ

3.1 Програмна реалізація алгоритму

3.1.1 Вихідний код

```
from knapsack import *
from helpers import *
import sys


def main():
    is_graph = False
    if len(sys.argv) > 1:
        if sys.argv[1] == 'gr':
            is_graph = True
        elif sys.argv[1] == 'ngr':
            is_graph = False
        else:
            print("Unexpected parameter")
            return
    if len(sys.argv) > 2:
        try:
            file_data = read_and_parse_file(sys.argv[2])
        except FileNotFoundError as e:
            print(f"Error: {e}")
            return
        except ValueError as e:
            print(f"Error: {e}")
            return
        except Exception as e:
            print(f"An unexpected error occurred: {e}")
            return
        knapsack = Knapsack(len(file_data), 100, (0, 0), (0, 0), 250,
0.05, 1000)
        knapsack.items = file_data
    else:
        knapsack = Knapsack(100, 100, (1, 25), (2, 30), 250, 0.05, 1000)
        knapsack.generate_items_data()
    knapsack.generate_initial_population()
    knapsack.record_counter()
    i_values, record_values = knapsack.genetic_algorithm()
```

```
        if is_graph:
            print("Число ітерацій Значення цільової функції")
            for i in range(len(i_values)):
                print(i_values[i], record_values[i])
            graph(i_values, record_values)
        knapsack.result()


    if __name__ == "__main__":
        main()


    import random


    class Knapsack:
        def __init__(self, num_items, population_size, weight_bounds,
value_bounds, max_weight, mutation_probability, iterations):
            self.num_items = num_items
            self.weight_bounds = weight_bounds
            self.value_bounds = value_bounds
            self.population_size = population_size
            self.max_weight = max_weight
            self.mutation_probability = mutation_probability
            self.items = []
            self.population = []
            self.record = 0
            self.record_chromosome = []
            self.iterations = iterations

        def generate_initial_population(self, difficulty_factor=2.3):
            self.population = [
                [random.choices([0, 1], weights=[difficulty_factor, 1],
k=1)[0]
                 for _ in range(self.num_items)]
                for _ in range(self.population_size)
            ]

        def generate_items_data(self):
            self.items = [[random.randint(*self.weight_bounds),
random.randint(*self.value_bounds)]
                          for _ in range(self.num_items)]
```

```
        def calculate_value(self, chromosome):
            total_weight = 0
            total_value = 0
            for i in range(len(chromosome)):
                if chromosome[i] == 1:
                    total_weight += self.items[i][0]
                    total_value += self.items[i][1]
            if total_weight > self.max_weight:
                return 0
            else:
                return total_value

        def record_counter(self):
            max_chromosome = max(self.population, key=lambda chromosome:
self.calculate_value(chromosome))
            self.record = self.calculate_value(max_chromosome)
            self.record_chromosome = max_chromosome

         def select_parents(self):
            # пропорційна селекція
            values = []
            for chromosome in self.population:
                values.append(self.calculate_value(chromosome))

            values_sum = sum(values)
            if values_sum:
                proportional_values = [float(i)/values_sum for i in values]
                while True:
                    parent1 = random.choices(self.population,
weights=proportional_values, k=1)[0]
                    parent1_index = self.population.index(parent1)
                    parent2 = random.choices(self.population,
weights=proportional_values, k=1)[0]
                    parent2_index = self.population.index(parent2)
                    if parent1_index == parent2_index:
                        while True:
                            parent2_index = random.randint(0,
self.population_size - 1)
                            if parent2_index != parent1_index:
                                break
                        parent2 = self.population[parent2_index]
                        break
```

```
            else:
                while True:
                    parent1_index,    parent2_index    =    (random.randint(0,
self.population_size - 1),
                                                          random.randint(0,
self.population_size - 1))
                    if parent1_index != parent2_index:
                        break
                parent1 = self.population[parent1_index]
                parent2 = self.population[parent2_index]
            return parent1, parent2

        def crossover(self, parent1, parent2):
            # триточковий оператор схрещування 25%
            size = self.num_items // 4
            parent1  =  [parent1[i:i+size]  for  i  in  range(0,  self.num_items,
size)]
            parent2  =  [parent2[i:i+size]  for  i  in  range(0,  self.num_items,
size)]
            child1 = parent1[0] + parent2[1] + parent1[2] + parent2[3]
            child2 = parent2[0] + parent1[1] + parent2[2] + parent1[3]
            return child1, child2

        def mutation(self, children):
            children_values  =  [self.calculate_value(child)  for  child  in
children]
            for i in range(len(children)):
                mutation_num = random.random()
                if mutation_num < self.mutation_probability:
                    mutation_index = random.randint(0, self.num_items - 1)
                    children[i][mutation_index]           =           int(not
children[i][mutation_index])
                    if          children_values[i]          and          not
self.calculate_value(children[i]):
                        children[i][mutation_index]         =         int(not
children[i][mutation_index])
            return children

        def local_improvement(self, children):
            # серед допустимих шукаємо предмет з найменшою вагою і додаємо
його
            children_values  =  [self.calculate_value(child)  for  child  in
children]
```

```
            for i in range(len(children)):
                available_items_indexes = [index for index, value in
enumerate(children[i]) if value == 1]
                available_items = [self.items[index] for index in
available_items_indexes]
                min_weight_item = min(available_items, key=lambda x: x[0])
                min_weight_item_index = self.items.index(min_weight_item)
                children[i][min_weight_item_index] = 1
                if children_values[i] and not
self.calculate_value(children[i]):
                    children[i][min_weight_item_index] = 0
            return children

        def iteration(self):
            parents = self.select_parents()
            children = self.crossover(parents[0], parents[1])
            children = self.mutation(children)
            children = self.local_improvement(children)

            population_values = [self.calculate_value(chromosome) for
chromosome in self.population]
            min_population_element = min(enumerate(population_values),
key=lambda x: x[1])
            for i, child in enumerate(children):
                child_value = self.calculate_value(child)
                if child_value and min(child_value, min_population_element[1])
== min_population_element[1]:
                    self.population[min_population_element[0]] = child
            self.record_counter()

        def genetic_algorithm(self):
            i_values = []
            record_values = []
            for i in range(self.iterations):
                self.iteration()
                if i % 20 == 0:
                    i_values.append(i)
                    record_values.append(self.record)
            return i_values, record_values

        def result(self):
            print("Items [Weight, Value]: ", self.items)
            total_weight = 0
```

```
            num_of_items = 0
            for i in range(len(self.record_chromosome)):
                if self.record_chromosome[i] == 1:
                    total_weight += self.items[i][0]
                    num_of_items += 1
            if total_weight > self.max_weight:
                print("Solution was not found")
                return
            print("Solution: ", self.record_chromosome)
            print("Knapsack value: ", self.record)
            print("Knapsack weight: ", total_weight)
            print("Num of items in knapsack: ", num_of_items)


    import matplotlib.pyplot as plt


    def read_and_parse_file(file_path):
        try:
            with open(file_path, 'r') as file:
                data = []
                for line_number, line in enumerate(file, start=1):
                    values = [value.strip() for value in line.split(',')]

                    if len(values) != 2:
                        raise ValueError(
                            f"Error in line {line_number}: Each line should
contain exactly two values separated by a comma and space.")

                    try:
                        values = [int(value) for value in values]
                    except ValueError:
                        raise ValueError(f"Error in line {line_number}: Values
must be valid integers.")

                    data.append(values)

                if len(data) % 4 != 0:
                    raise ValueError("Error: The number of lines in the file
must be a multiple of 4.")
            return data
        except FileNotFoundError:
            raise FileNotFoundError(f"The file '{file_path}' does not exist.")
```

```
            except Exception as e:
                raise e


        def graph(x_values, y_values):
            plt.plot(x_values, y_values, label='Графік залежності якості розв`язку
від числа ітерацій')
            plt.xlabel('Число ітерацій')
            plt.ylabel('Якість розв`язку')
            plt.xticks(x_values[::2])
            plt.yticks(list(set(y_values)))
            plt.show()
```

3.1.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

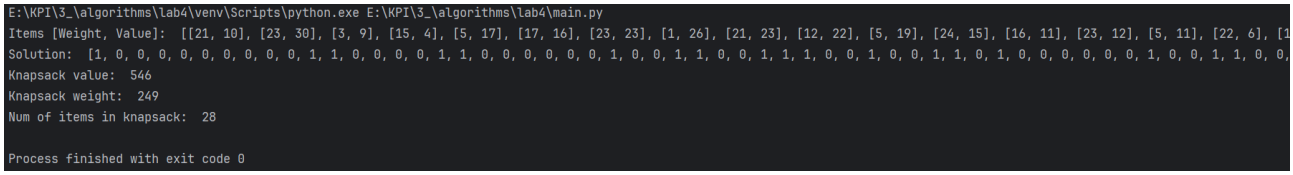

Рисунок 3.1 – Приклад роботи програми з генерацією випадкових предметів

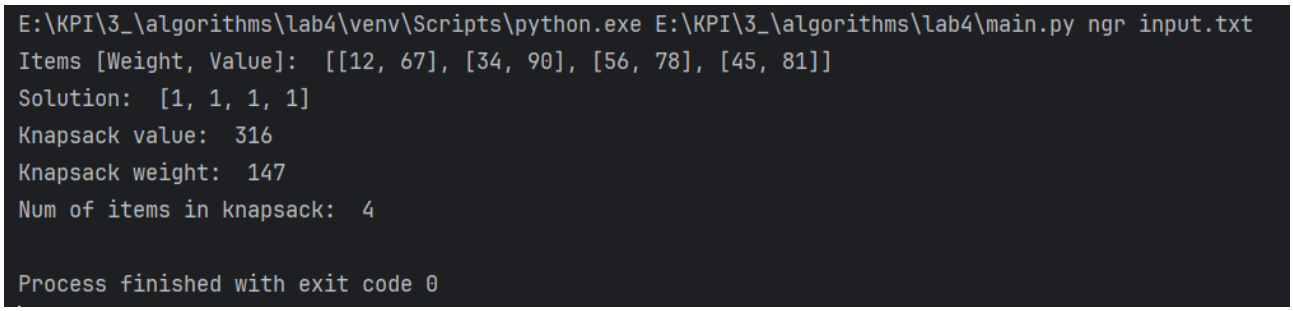

Рисунок 3.2 – Приклад роботи програми з читанням предметів з файлу

3.2 Тестування алгоритму

3.2.1 Значення цільової функції зі збільшенням кількості ітерацій

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

Кількість ітерацій	Значення цільової функції
0	409
20	422
40	422
60	422
80	422
100	422
120	422
140	422
160	422
180	422
200	424
220	424
240	424
260	424
280	433
300	433
320	433
340	433
360	433
380	433
400	433
420	433
440	433

460	452
480	452
500	452
520	452
540	452
560	452
580	452
600	452
620	452
640	452
660	452
680	452
700	452
720	452
740	452
760	452
780	452
800	452
820	452
840	452
860	452
880	475
900	475
920	475
940	475
960	475
980	475
1000	475

3.2.2 Графіки залежності розв'язку від числа ітерацій

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

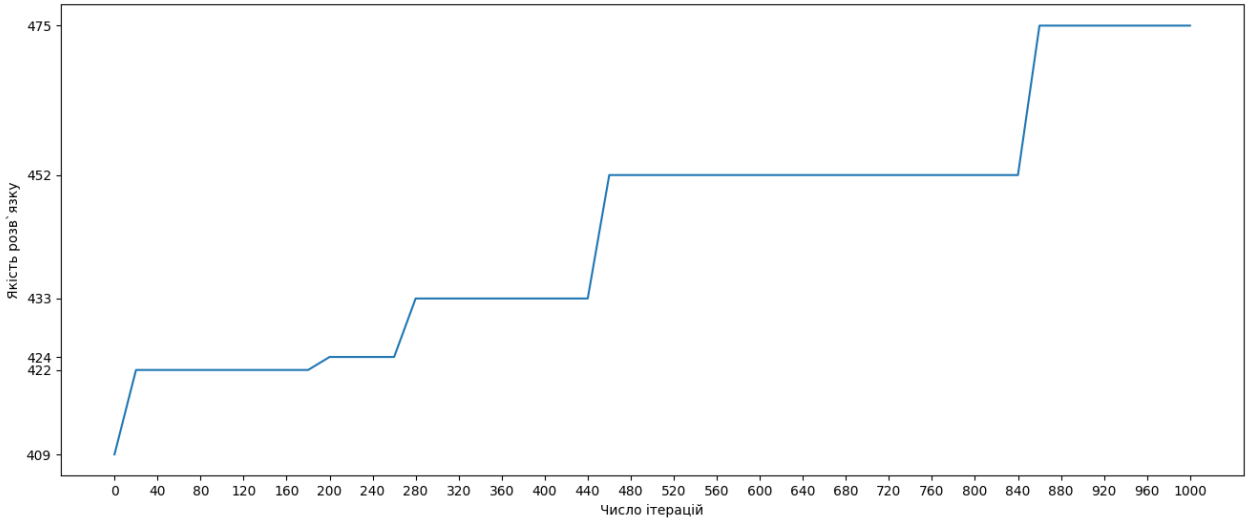

Рисунок 3.3 – Графіки залежності розв'язку від числа ітерацій

ВИСНОВОК

В рамках даної лабораторної роботи я вивчив основні підходи формалізації метаевристичних алгоритмів і вирішення типових задач з їхньою допомогою. Результатом моєї роботи стала програмна реалізація рішення задачі про рюкзак за допомогою генетичного алгоритму. Я розробив оператор вибору батьків для схрещування за допомогою пропорційної селекції. Оператор схрещування у моїй реалізації генетичного алгоритму є триточковим 25%. Мутація у моєму варіанті алгоритму відбувається з одним випадковим геном з ймовірністю 5%. За основу оператора локального покращення я взяв спробу серед ще невибраних предметів для хромосоми вибрати предмет з найменшою вагою та додати його в рюкзак.

Загалом програмна реалізація алгоритму показує себе добре. За 1000 ітерацій програма знаходить розв’язок, близький до оптимального, поступово покращуючи його. Тестування показало, що якщо ліміт ітерацій брати більше за 1000, то в рамках перших трьох – п’яти тисяч ітерацій алгоритму цінність рюкзака в рішенні може зрости ще на кілька десятків пунктів. Подальший ріст цінності поза межею цієї кількості ітерацій відбувається неохоче. Вага рюкзака в рішенні майже завжди досягає граничного значення.

КРИТЕРІЇ ОЦІНЮВАННЯ

При здачі лабораторної роботи до 10.12.2023 включно максимальний бал дорівнює – 5. Після 10.12.2023 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

- програмна реалізація алгоритму – 55%;
- робота з гіт – 20%;
- тестування алгоритму– 20%;
- висновок – 5%.

+1 додатковий бал можна отримати за виконання роботи до 3.12.2023