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 *ІП-12 Басараб Олег Андрійович* _____
(шифр, прізвище, ім'я, по батькові)

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

Київ 2023

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.

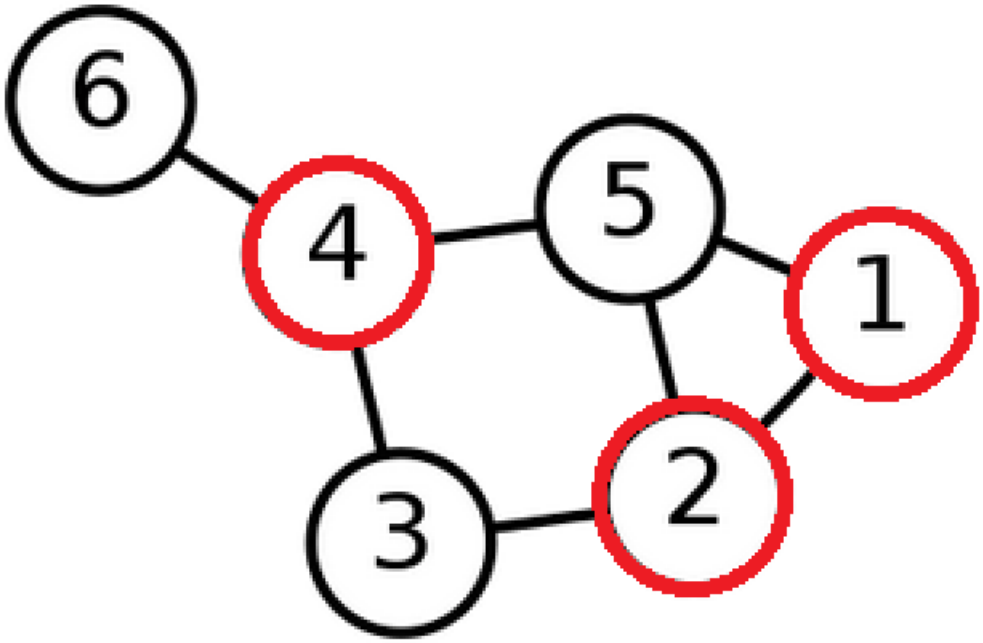
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

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 ⊆ 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

Варіант 2 – Задача про рюкзак + Бджолиний алгоритм

3.1 Покроковий алгоритм

1. Встановлення та генерація параметрів задачі про рюкзак (місткості рюкзака, цінності та маси кожного з предметів, відносної цінності предметів (відношення цінності до маси) кожного з предметів).
2. Відсортовування маси, цінності та відносної цінності допустимих предметів у задачі про рюкзак за відносною цінністю в порядку спадання (це значно спростить виправлення початкових розв’язків бджолами-робітниками на кроці 4.1)
3. Встановлення та генерація параметрів для бджолиного алгоритму (кількості ітерацій, розміру бджолиної колонії, кількості бджіл-розвідників, початкових ділянок з їжею (тобто початкових розв’язків)).
4. Запуск циклу for на задану на кроці 3 кількість ітерацій.
   4.1. Діяльність бджіл-робітників (окрім стандартних для алгоритму бджолиної колонії кроків, включає виправлення початкових розв’язків, якщо вони цього потребують (відбувається, коли початковий розв’язок має більшу масу за місткість рюкзака; передбачає видалення предметів з найменшою відносною цінністю)).
   4.2. Діяльність бджіл-спостерігачів.
   4.3. Запам’ятовування найкращого розв’язку.
   4.4. Діяльність бджіл-розвідників.
5. Повернення розв’язку, що відповідає найкращій ділянці з їжею.

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

Файл KnapsackProblem.py:

```
import random
import math
```

```
from sys import float_info

class KnapsackProblem:

    def __init__(self,
                 capacity = 500,
                 number_of_unique_items = 100,
                 min_item_profit = 2,
                 max_item_profit = 30,
                 min_item_weight = 1,
                 max_item_weight = 20,
                 seed = None):

        self.capacity = capacity
        self.number_of_unique_items = number_of_unique_items
        self.min_item_profit = min_item_profit
        self.max_item_profit = max_item_profit
        self.min_item_weight = min_item_weight
        self.max_item_weight = max_item_weight

        if (seed == None):
            self.seed = random.randint(0, 1000)
        else:
            self.seed = seed
        random.seed(self.seed)

        self._random()
        self.lower = [0] * number_of_unique_items
        self.upper = [math.floor(capacity / self.wpr[i][1]) for i in
range(number_of_unique_items)]
        # self.upper = [1] * number_of_unique_items

    def _random(self):
        weights = []
        profits = []
        relation_values = []

        for _ in range(self.number_of_unique_items):
            weights.append(random.randint(self.min_item_weight,
self.max_item_weight))
            profits.append(random.randint(self.min_item_profit,
self.max_item_profit))
            relation_values.append(profits[-1] / weights[-1])

        zipped_wpr = zip(weights, profits, relation_values)
        wpr = list(zipped_wpr)
        wpr.sort(key = lambda i: i[2], reverse = True)

        self.wpr = wpr

    def _calculate_total_weight(self, x: list):
        total_weight = 0
        for i in range(self.number_of_unique_items):
            total_weight += self.wpr[i][0] * round(x[i])

        return total_weight

    def is_solution_acceptable(self, x: list):
        solution_is_acceptable = True
        total_weight = self._calculate_total_weight(x)
        if total_weight > self.capacity:
            solution_is_acceptable = False

        return solution_is_acceptable

    def calculate_total_profit(self, x: list):
        total_profit = 0
```

```
        for i in range(self.number_of_unique_items):
            total_profit += self.wpr[i][1] * round(x[i])

        if not self.is_solution_acceptable(x):
            total_profit = float_info.min

        return total_profit
```

Файл main.py:

```
import math
import numpy as np
from KnapsackProblem import *
from KnapsackArtificialBeeColony import *
import matplotlib.pyplot as plt

def main():
    currKnapsackProblem = KnapsackProblem(seed = 123)
    abc_obj = KnapsackArtificialBeeColony(currKnapsackProblem, 50, iterations = 200,
min_max = 'max', seed = 123, scouts = 10)
    abc_obj.fit()
    solution = abc_obj.get_solution()

    print("Solution:")
    print(solution)
    print("Solution Total Profit: " +
str(currKnapsackProblem.calculate_total_profit(solution)))
    print("Solution Total Weight: " +
str(currKnapsackProblem._calculate_total_weight(solution)))
    #x1 = []
    #y1 = []
    #for i in range(100):
    #    abc_obj = KnapsackArtificialBeeColony(currKnapsackProblem, 4 + i * 4,
iterations = 100, min_max = 'max', seed = 123, scouts = 5)
    #    abc_obj.fit()
    #    x1.append(4 + i * 4)

    #    solution = abc_obj.get_solution()
    #    y1.append(currKnapsackProblem.calculate_total_profit(solution))
    #    print(str(currKnapsackProblem.calculate_total_profit(solution)) + ' ' +
str(currKnapsackProblem._calculate_total_weight(solution)) + " " + str(4 + i * 4))

    #plt.xlabel("Colony Size")
    #plt.ylabel("Max Total Profit")
    #plt.plot(x1, y1)
    #plt.savefig('ColonySize_vs_MaxTotalProfit.pdf', format = 'pdf')
    #abc_obj = KnapsackArtificialBeeColony(currKnapsackProblem, 148, iterations =
100, min_max = 'max', seed = 123, scouts = 5)
    #abc_obj.fit()
    #solution = abc_obj.get_solution()
    #print(str(currKnapsackProblem.calculate_total_profit(solution)) + ' ' +
str(currKnapsackProblem._calculate_total_weight(solution)))
    #x2 = []
    #y2 = []
    #for i in range(30):
    #    abc_obj = KnapsackArtificialBeeColony(currKnapsackProblem, 148, iterations
= 100, min_max = 'max', seed = 123, scouts = 5 + 5 * i)
    #    abc_obj.fit()
    #    x2.append(4 + i * 4)

    #    solution = abc_obj.get_solution()
    #    y2.append(currKnapsackProblem.calculate_total_profit(solution))
    #    print(str(currKnapsackProblem.calculate_total_profit(solution)) + ' ' +
str(currKnapsackProblem._calculate_total_weight(solution)) + " " + str(5 + 5 * i))
```

```
    #plt.xlabel("Scouts Per Iteration")
    #plt.ylabel("Max Total Profit")
    #plt.plot(x2, y2)
    #plt.savefig('ScoutsPerIteration_vs_MaxTotalProfit.pdf', format = 'pdf')

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

Файл KnapsackArtificialBeeColony.py:

```
import numpy as np
import random as rng
import warnings as wrn
import KnapsackProblem as kp
import math

class KnapsackArtificialBeeColony:

    def __init__(self,
                 knapsack_problem: kp.KnapsackProblem,
                 colony_size: int=40,
                 scouts: float=0.5,
                 iterations: int=50,
                 min_max: str='min',
                 nan_protection: bool=True,
                 seed: int=None):

        boundaries = list(zip(knapsack_problem.lower, knapsack_problem.upper))
        self.wpr = knapsack_problem.wpr
        self.constraint_function = knapsack_problem.is_solution_acceptable
        self.boundaries = boundaries
        self.min_max_selector = min_max
        self.cost_function = knapsack_problem.calculate_total_profit
        self.nan_protection = nan_protection
        self.seed = seed

        self.max_iterations = int(max([iterations, 1]))
        if (iterations < 1):
            warn_message = 'Using the minimun value of iterations = 1'
            wrn.warn(warn_message, RuntimeWarning)

        self.employed_onlookers_count = int(max([(colony_size/2), 2]))
        if (colony_size < 4):
            warn_message = 'Using the minimun value of colony_size = 4'
            wrn.warn(warn_message, RuntimeWarning)

        if (scouts <= 0):
            self.scout_limit = int(self.employed_onlookers_count *
len(self.boundaries))
            if (scouts < 0):
                warn_message = 'Negative scout count given, using default scout ' \
                    'count: colony_size * dimension = ' + str(self.scout_limit)
                wrn.warn(warn_message, RuntimeWarning)
        elif (scouts < 1):
            self.scout_limit = int(self.employed_onlookers_count *
len(self.boundaries) * scouts)
        else:
            self.scout_limit = int(scouts)

        self.scout_status = 0
        self.iteration_status = 0
        self.nan_status = 0

        if (self.seed is not None):
            rng.seed(self.seed)

        self.foods = [None] * self.employed_onlookers_count
```

```
        for i in range(len(self.foods)):
            _ABC_engine(self).generate_food_source(i)

        try:
            self.best_food_source = self.foods[np.nanargmax([food.fit for food in
self.foods])]
        except:
            self.best_food_source = self.foods[0]
            warn_message = 'All food sources\'s fit resulted in NaN and beecolpy can
got stuck ' \
                           'in an infinite loop during fit(). Enable nan_protection to
prevent this.'
            wrn.warn(warn_message, RuntimeWarning)


    def fit(self):
        if (self.seed is not None):
            rng.seed(self.seed)

        for _ in range(self.max_iterations):

            _ABC_engine(self).employer_bee_phase()

            _ABC_engine(self).onlooker_bee_phase()

            _ABC_engine(self).memorize_best_solution()

            _ABC_engine(self).scout_bee_phase()

            self.iteration_status += 1

        return self.best_food_source.position


    def get_solution(self):
        assert (self.iteration_status > 0), 'fit() not executed yet!'
        return self.best_food_source.position


    def get_status(self):
        return self.iteration_status, self.scout_status, self.nan_status


class _FoodSource:
    def __init__(self, abc, engine):

        self.abc = abc
        self.engine = engine
        self.trial_counter = 0
        self.position = [rng.uniform(*self.abc.boundaries[i]) for i in
range(len(self.abc.boundaries))]
        self.fit = self.engine.calculate_fit(self.position)


    def evaluate_neighbor(self, partner_position):

        j = rng.randrange(0, len(self.abc.boundaries))

        xj_new = self.position[j] + rng.uniform(-1, 1)*(self.position[j] -
partner_position[j])

        xj_new = self.abc.boundaries[j][0] if (xj_new < self.abc.boundaries[j][0])
else \
            self.abc.boundaries[j][1] if (xj_new > self.abc.boundaries[j][1]) else
xj_new
```

```
        neighbor_position = [(self.position[i] if (i != j) else xj_new) for i in
range(len(self.abc.boundaries))]
        j = len(neighbor_position) - 1
        while not self.abc.constraint_function(neighbor_position):
            while j >= 0:
                if round(neighbor_position[j]) != 0:
                    neighbor_position[j] = 0
                    break
                else:
                    j -= 1

        neighbor_fit = self.engine.calculate_fit(neighbor_position)

        if (neighbor_fit > self.fit):
            self.position = neighbor_position
            self.fit = neighbor_fit
            self.trial_counter = 0
        else:
            self.trial_counter += 1



class _ABC_engine:
    def __init__(self, abc):
        self.abc = abc


    def check_nan_lock(self):
        if not(self.abc.nan_protection):
            if np.all([np.isnan(food.fit) for food in self.abc.foods]):
                raise Exception('All food sources\'s fit resulted in NaN and
beecolpy got ' \
                                'stuck in an infinite loop. Enable nan_protection to
prevent this.')


    def execute_nan_protection(self, food_index):
        while (np.isnan(self.abc.foods[food_index].fit) and
self.abc.nan_protection):
            self.abc.nan_status += 1
            self.abc.foods[food_index] = _FoodSource(self.abc, self)


    def generate_food_source(self, index):
        self.abc.foods[index] = _FoodSource(self.abc, self)
        self.execute_nan_protection(index)


    def prob_i(self, actual_fit, max_fit):
        return 0.9*(actual_fit/max_fit) + 0.1


    def calculate_fit(self, evaluated_position):
        cost = self.abc.cost_function(evaluated_position)
        if (self.abc.min_max_selector == 'min'):
            fit_value = (1 + np.abs(cost)) if (cost < 0) else (1/(1 + cost))
        else:
            fit_value = (1 + cost) if (cost > 0) else (1/(1 + np.abs(cost)))
        return fit_value


    def food_source_dance(self, index):
        while True:
            d = int(rng.randrange(0, self.abc.employed_onlookers_count))
            if (d != index):
                break
        self.abc.foods[index].evaluate_neighbor(self.abc.foods[d].position)
```

```
    def employer_bee_phase(self):
        for i in range(len(self.abc.foods)):
            j = len(self.abc.foods[i].position) - 1
            while not self.abc.constraint_function(self.abc.foods[i].position):
                while j >= 0:
                    if round(self.abc.foods[i].position[j]) != 0:
                        self.abc.foods[i].position[j] = 0
                        break
                    else:
                        j -= 1

            self.abc.foods[i].fit = self.calculate_fit(self.abc.foods[i].position)

            self.food_source_dance(i)


    def onlooker_bee_phase(self):
        self.check_nan_lock()
        max_fit = np.nanmax([food.fit for food in self.abc.foods])
        onlooker_probability = [self.prob_i(food.fit, max_fit) for food in
self.abc.foods]
        p = 0
        i = 0
        while (p < self.abc.employed_onlookers_count):
            if (rng.uniform(0, 1) <= onlooker_probability[i]):
                p += 1
                self.food_source_dance(i)
                self.check_nan_lock()
                max_fit = np.nanmax([food.fit for food in self.abc.foods])
                if (self.abc.foods[i].fit != max_fit):
                    onlooker_probability[i] = self.prob_i(self.abc.foods[i].fit,
max_fit)
                else:
                    onlooker_probability = [self.prob_i(food.fit, max_fit) for food
in self.abc.foods]
            i = (i+1) if (i < (len(self.abc.foods)-1)) else 0


    def scout_bee_phase(self):
        trial_counters = [food.trial_counter for food in self.abc.foods]
        if (max(trial_counters) > self.abc.scout_limit):
            trial_counters = np.where(np.array(trial_counters) ==
max(trial_counters))[0].tolist()
            i = trial_counters[rng.randrange(0, len(trial_counters))]
            self.generate_food_source(i)
            self.abc.scout_status += 1


    def memorize_best_solution(self):
        best_food_index = np.nanargmax([food.fit for food in self.abc.foods])
        if (self.abc.foods[best_food_index].fit >= self.abc.best_food_source.fit):
            self.abc.best_food_source = self.abc.foods[best_food_index]
```

3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

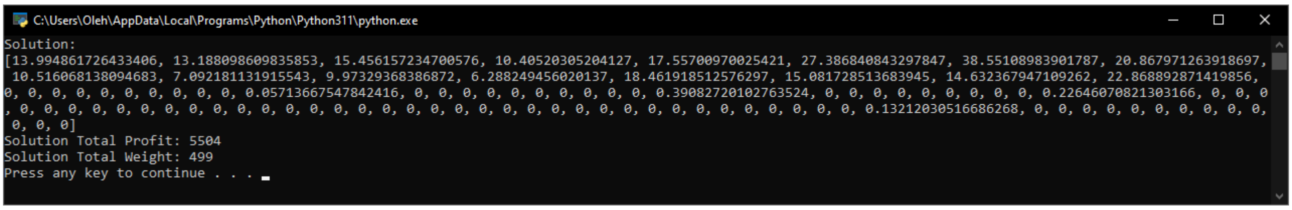


Рисунок 3.1 – Приклад роботи програми при розмірі колонії – 20, кількості ітерацій – 100, кількості бджіл-розвідників – 5

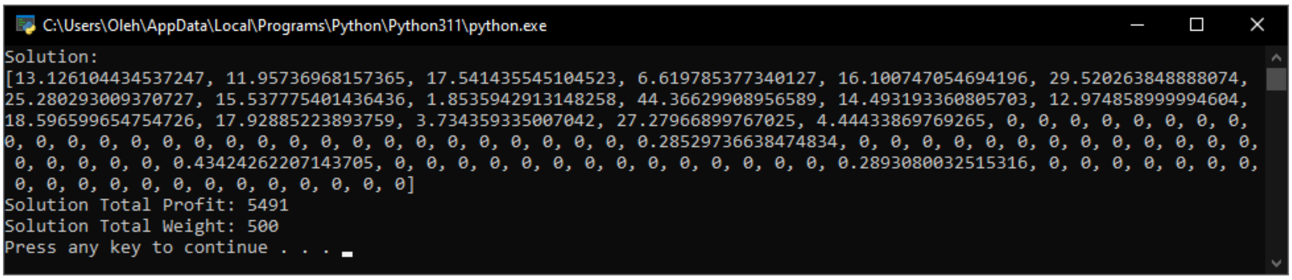


Рисунок 3.2 – Приклад роботи програми при розмірі колонії – 40, кількості ітерацій – 200, кількості бджіл-розвідників – 10

Варто зауважити, що хоч в розв'язках фігурують нецілі числа, це пов'язано винятково з особливістю роботи алгоритму штучної бджолиної колонії. При підрахунку цінності впакованого рюкзака нецілі числа з розв'язку заокруглюються до найближчих цілих.

3.3 Тестування алгоритму

Тестування алгоритму проводилося з використанням максимальної кількості ітерацій (100) як критерію зупинки роботи алгоритму.

Алгоритм тестувався на задачі про рюкзак, до якого можна покласти необмежену, за винятком місткості рюкзака, кількість предметів одного типу.

Задля того, аби в кожному досліді використовувалися одні й ті самі дані про можливі предмети в рюкзаку, було введено параметр зерна генерації для генерації псевдовипадкових чисел.

Спершу досліджувався параметр – розмір бджолиної колонії. Варто зауважити, що від розміру бджолиної колонії в даній реалізації алгоритму залежали також кількість бджіл-робітників (employed) та бджіл-спостерігачів (onlooker) (їх було порівну; кількість бджіл кожного з цих двох типів дорівнювала половині розміру бджолиної колонії).

У таблиці 3.1 наведено дані про це дослідження.

Таблиця 3.1 – Цінність впакованого рюкзака до розміру бджолиної колонії

Розмір бджолиної колонії	Цінність впакованого рюкзака	Розмір бджолиної колонії	Цінність впакованого рюкзака
4	5296	44	5447
8	5500	48	5573
12	5195	52	5502
16	5281	56	5535
20	5504	60	5455
24	5375	64	5454
28	5352	68	5725
32	5327	72	5625
36	5581	76	5559
40	5467	80	5628

Продовження таблиці 3.1

Розмір бджолиної колонії	Цінність впакованого рюкзака	Розмір бджолиної колонії	Цінність впакованого рюкзака
84	5484	184	5748
88	5396	188	5734
92	5562	192	5659
96	5516	196	5692
100	5696	200	5485
104	5447	204	5769
108	5522	208	5678
112	5701	212	5689
116	5459	216	5711
120	5638	220	5654
124	5758	224	5802
128	5618	228	5655
132	5472	232	5810
136	5569	236	5572
140	5632	240	5737
144	5586	244	5524
148	5839	248	5624
152	5723	252	5605
156	5630	256	5759
160	5583	260	5634
164	5664	264	5554
168	5620	268	5590
172	5726	272	5600
176	5715	276	5650
180	5620	280	5674

Продовження таблиці 3.1

Розмір бджолиної колонії	Цінність впакованого рюкзака	Розмір бджолиної колонії	Цінність впакованого рюкзака
284	5660	344	5584
288	5771	348	5556
292	5738	352	5596
296	5685	356	5637
300	5716	360	5654
304	5581	364	5667
308	5511	368	5666
312	5640	372	5623
316	5699	376	5793
320	5555	380	5777
324	5614	384	5660
328	5498	388	5644
332	5755	392	5750
336	5634	396	5673
340	5694	400	5693

На рисунку 3.3 можна побачити графік залежності цінності впакованого рюкзака від розміру бджолиної колонії.

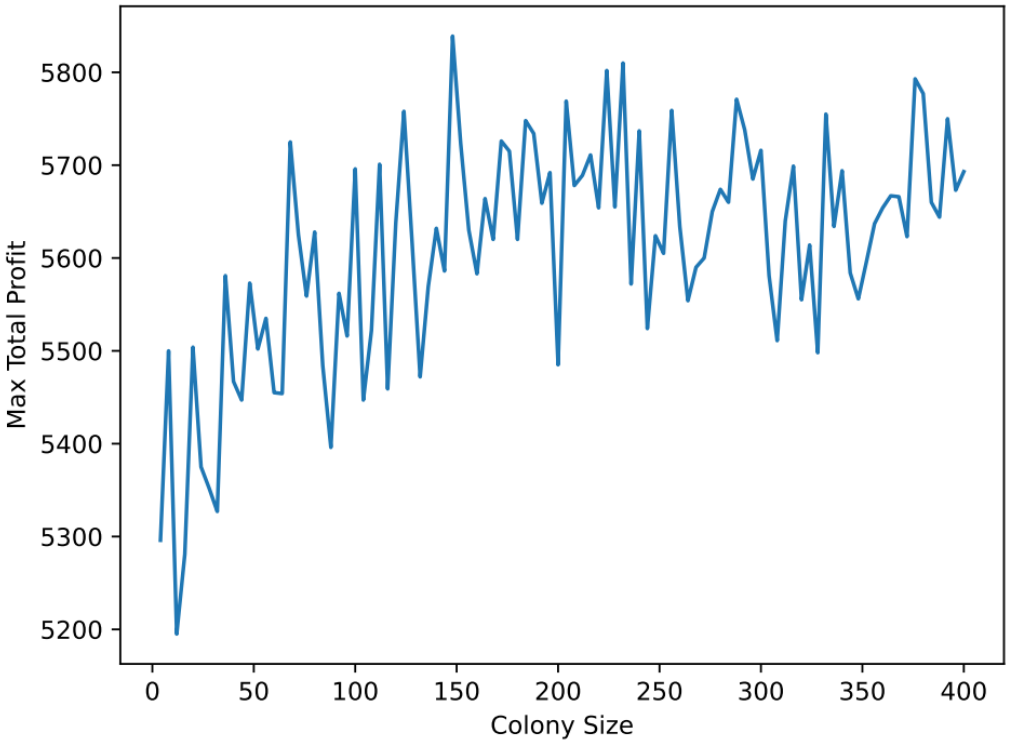


Рисунок 3.3 – Графік залежності цінності впакованого рюкзака від розміру бджолиної колонії

З графіку на рисунку 3.3 і таблиці 3.1 отримуємо, що розв’язок починає стагнувати при розмірі колонії близькій до 148 і далі. Зафіксовуємо дане значення як оптимальне.

Наступним і останнім параметром, який було досліджено, стала кількість бджіл-розвідників (scouts) на ітерації.

У таблиці 3.2 наведено дані про це дослідження.

Таблиця 3.2 – Цінність впакованого рюкзака до кількості бджіл-розвідників

Кількість бджіл розвідників	Цінність впакованого рюкзака	Кількість бджіл розвідників	Цінність впакованого рюкзака
5	5839	20	5774
10	5585	25	5808
15	5563	30	5599

Продовження таблиці 3.2

Кількість бджіл розвідників	Цінність впакованого рюкзака	Кількість бджіл розвідників	Цінність впакованого рюкзака
35	5764	95	5575
40	5663	100	5672
45	5733	105	5659
50	5685	110	5658
55	5614	115	5640
60	5624	120	5657
65	5649	125	5671
70	5571	130	5674
75	5576	135	5655
80	5804	140	5657
85	5575	145	5631
90	5641	150	5627

На рисунку 3.4 можна побачити графік залежності цінності впакованого рюкзака від кількості бджіл-розвідників.

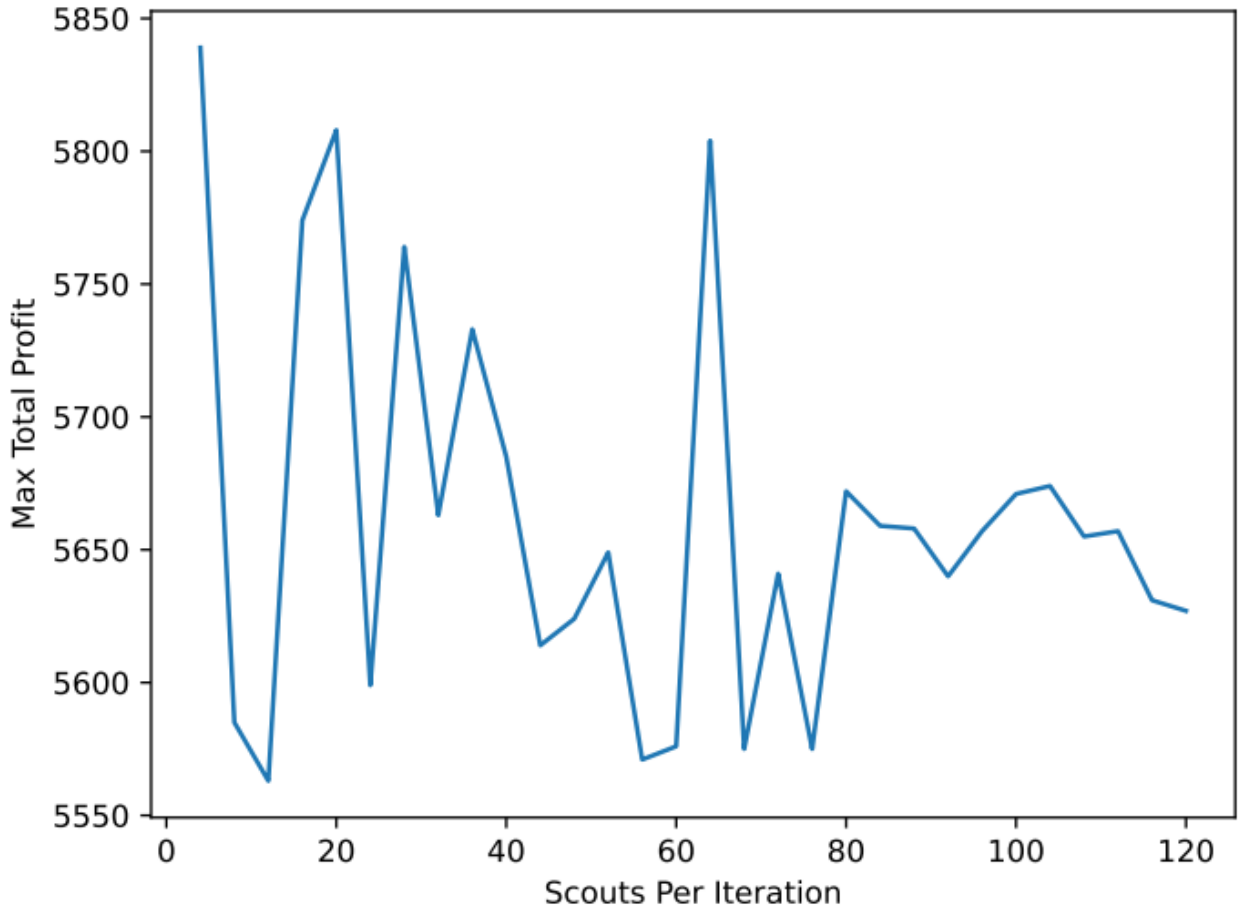


Рисунок 3.3 – Графік залежності цінності впакованого рюкзака від кількості бджіл-розвідників

З графіку на рисунку 3.4 і таблиці 3.2 отримуємо, що розв’язок починає стагнувати і навіть погіршуватися уже при кількості бджіл-розвідників – 5. Зафіксовуємо дане значення як оптимальне.

ВИСНОВОК

В рамках даної лабораторної роботи було опрацьовано та здійснено програмну реалізації алгоритму штучної бджолиної колонії для розв’язку задачі про рюкзак (в який можна класти декілька предметів одного типу) мовою python. Основною ціллю даної роботи було визначення оптимальних значень змінних параметрів, що впливають на ефективність алгоритму. Було визначено, що для даної задачі про рюкзак (місткість – 500, мінімальна допустима маса предмету – 1, максимально допустима маса предмету – 20, мінімальна допустима цінність предмету – 2, максимально допустима цінність предмету – 30), оптимальним значенням розміру бджолиної колонії є 148 (відповідно, кількість ділянок з їжею – 74, кількість бджіл-робітників – 74, кількість бджіл-спостерігачів – 74). Оптимальне значення кількості бджіл-розвідників – 5. Визначення цих оптимальних параметрів здійснювалося завдяки аналізу таблиць і графік, що відображають залежність максимальної вартості впакованого рюкзака до розміру колонії і кількості бджіл-розвідників відповідно.

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%.