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-15 Борисик Владислав Тарасович* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Ахаладзе Ілля Елдарійович* _____
(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.

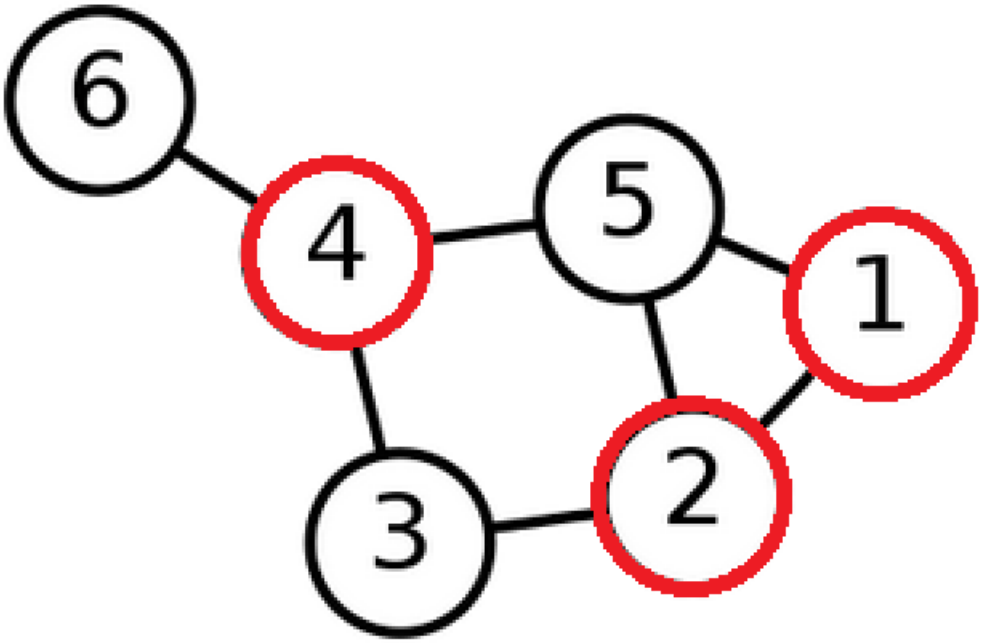
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

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 ⊆ 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

Варіант 2

Задача про рюкзак + Бджолиний алгоритм

3 ВИКОНАННЯ

3.1 Покроковий алгоритм

```
Функція bee_colony_method(generated_items: list[Item], greedy_value: int):
    bee_colony_value = 0

    поки bee_colony_value < greedy_value:
        items = generated_items.copy()
        backpack = Backpack()

        поки backpack.weight + items[foragers_accumulation].weight <= backpack.volume:
            scouts = spawn_scouts(items)
            foragers_accumulation = define_foragers_accumulation(scouts)

            backpack.add_item(items[foragers_accumulation])
            items.pop(foragers_accumulation)

        print(f"Artificial bee colony value: {backpack.value}.")
        save_data_to_file("bee_colony.txt", backpack)

        bee_colony_value = backpack.value

    повернути bee_colony_value

Функція spawn_scouts(items: list[Item]):
    scouts = []
    occupied_positions = []

    # найменше значення серед кількістю розвідників і кількістю предметів
    # потрібно для того, щоб кількість розвідників не перевищувала кількість предметів
    smallest_value = SCOUTS_AMOUNT+1 якщо SCOUTS_AMOUNT+1 < len(items) інакше len(items)

    для і від 0 до smallest_value - 1:
        position = random.randint(0, len(items) - 1)

        поки position в occupied_positions:
```

```
            position = random.randint(0, len(items) - 1)

        scout = Scout(position, items[position])

        scouts.append(scout)
        occupied_positions.append(position)

    повернути scouts

Функція define_foragers_accumulation(scouts: list[Scout]):
    foragers = spawn_foragers()

    information = collect_items_information(scouts)
    foragers_travel(foragers, scouts, information)

    foragers_accumulation = find_maximum_accumulated_item(foragers)

    return foragers_accumulation

Функція collect_items_information(scouts: list[Scout]):
    information = [scout.collect_information() для scout в scouts]

    повернути information

Функція foragers_travel(foragers: list[Forager], scouts: list[Scout], information: list[int]):
    для forager в foragers:
        forager.travel_to_position(information, scouts)

Функція find_maximum_accumulated_item(foragers: list[Forager]):
    accumulation = {}

    для forager в foragers:
        якщо forager.position не в accumulation:
            accumulation[forager.position] = 1
        інакше:
            accumulation[forager.position] += 1
```

```
повернути max(accumulation)
```

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

main.py

```
from functions import generate_items, greedy_method, bee_colony_method
from classes import Backpack, Item


backpack = Backpack()


# генеруємо предмети
items = generate_items()


# вирішуємо задачу жадібним алгоритмом
greedy_value = greedy_method(items)


# вирішуємо задачу бджолиним алгоритмом
bee_colony_method(items, greedy_value)
```

functions.py

```
from classes import *
from constants import *



def generate_items():
    items = []

    for i in range(ITEMS_AMOUNT):
        value = random.randint(MIN_VALUE, MAX_VALUE)
        weight = random.randint(MIN_WEIGHT, MAX_WEIGHT)

        item = Item(value, weight)

        items.append(item)

    return items



def greedy_method(items: list[Item]):
    backpack = Backpack()
    counter = 0

    while True:
        items_copy = items.copy()

        scout = [Scout(counter, items_copy[counter])]
        foragers_accumulation = define_foragers_accumulation(scout)

        if backpack.weight + items_copy[foragers_accumulation].weight > backpack.volume:
            break

        backpack.add_item(items_copy[foragers_accumulation])
        items_copy.pop(foragers_accumulation)
```

```
        counter += 1

    print(f"Greedy value: {backpack.value}.")
    save_data_to_file("greedy_method.txt", backpack)

    return backpack.value



def bee_colony_method(generated_items: list[Item], greedy_value: int):
    bee_colony_value = 0

    while bee_colony_value < greedy_value:
        items = generated_items.copy()
        backpack = Backpack()

        while True:
            scouts = spawn_scouts(items)
            foragers_accumulation = define_foragers_accumulation(scouts)

            if backpack.weight + items[foragers_accumulation].weight > backpack.volume:
                break

            backpack.add_item(items[foragers_accumulation])
            items.pop(foragers_accumulation)

        print(f"Artificial bee colony value: {backpack.value}.")
        save_data_to_file("bee_colony.txt", backpack)

        bee_colony_value = backpack.value

    return bee_colony_value



def define_foragers_accumulation(scouts: list[Scout]):
    foragers = spawn_foragers()
```

```
    information = collect_items_information(scouts)
    foragers_travel(foragers, scouts, information)

    foragers_accumulation = find_maximum_accumulated_item(foragers)

    return foragers_accumulation



def spawn_scouts(items: list[Item]):
    scouts = []
    occupied_positions = []

    # найменше значення серед кількістю розвідників і кількістю предметів
    # потрібно для того, щоб кількість розвідників не перевищувала кількість предметів
    smallest_value = SCOUTS_AMOUNT+1 if SCOUTS_AMOUNT+1 < len(items) else len(items)

    for i in range(smallest_value - 1):
        position = random.randint(0, len(items) - 1)

        while position in occupied_positions:
            position = random.randint(0, len(items) - 1)

        scout = Scout(position, items[position])

        scouts.append(scout)
        occupied_positions.append(position)

    return scouts



def spawn_foragers():
    foragers = []

    for i in range(FORAGERS_AMOUNT):
```

```
        forager = Forager(0)

        foragers.append(forager)

    return foragers


def collect_items_information(scouts: list[Scout]):
    information = [scout.collect_information() for scout in scouts]

    return information


def foragers_travel(foragers: list[Forager], scouts: list[Scout], information: list[int]):
    for forager in foragers:
        forager.travel_to_position(information, scouts)


def find_maximum_accumulated_item(foragers: list[Forager]):
    accumulation = {}

    for forager in foragers:
        if forager.position not in accumulation:
            accumulation[forager.position] = 1
        else:
            accumulation[forager.position] += 1

    return max(accumulation)


def save_data_to_file(file_name: str, backpack: Backpack):
    string_to_write = f"Backpack value: {backpack.value}.
Average item value: {round(backpack.value / len(backpack.items), 2)}
Average item weight: {round(backpack.weight / len(backpack.items), 2)}
```

```
Items in backback:\n'"

    for item in backpack.items:
        string_to_write += str(item)

    with open(file_name, mode='w') as file:
        file.write(string_to_write)
```

classes.py

```
import random



class Item:
    def __init__(self, value: int, weight: int):
        self.value = value
        self.weight = weight

    def __str__(self):
        return f"Item - value: {self.value}, weight: {self.weight}\n"

    def get_nectar_amount(self):
        return self.value / self.weight



class Backpack:
    def __init__(self):
        self.volume = 500
        self.weight = 0
        self.value = 0
        self.items = []

    def add_item(self, item: Item):
        if self.weight + item.weight <= self.volume:
            self.items.append(item)
            self.value += item.value
            self.weight += item.weight



class Bee:
    def __init__(self, position: int):
        self.position = position
```

```python
class Scout(Bee):
    def __init__(self, position: int, item: Item):
        super().__init__(position)
        self.explored_item = item

    def collect_information(self):
        return self.explored_item.get_nectar_amount()



class Forager(Bee):
    def travel_to_position(self, information: list[int], scouts: list[Scout]):
        # визначаємо куди летіти за допомогою дискретної величини
        discrete_quantity = []
        discrete_quantity_sum = 0

        for i in range(len(information)):
            discrete_quantity_sum += information[i]
            discrete_quantity.append(discrete_quantity_sum)

        random_number = random.uniform(0, discrete_quantity[len(discrete_quantity) - 1])

        for i in range(len(discrete_quantity)):
            if i == 0:
                if random_number < discrete_quantity[i]:
                    self.position = scouts[i].position
                    break
            else:
                if discrete_quantity[i - 1] < random_number <= discrete_quantity[i]:
                    self.position = scouts[i].position
                    break
```

constants.py

```
ITEMS_AMOUNT = 100
MIN_VALUE = 2
MAX_VALUE = 30
MIN_WEIGHT = 1
MAX_WEIGHT = 20


FORAGERS_AMOUNT = 5
SCOUTS_AMOUNT = 5
```

3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

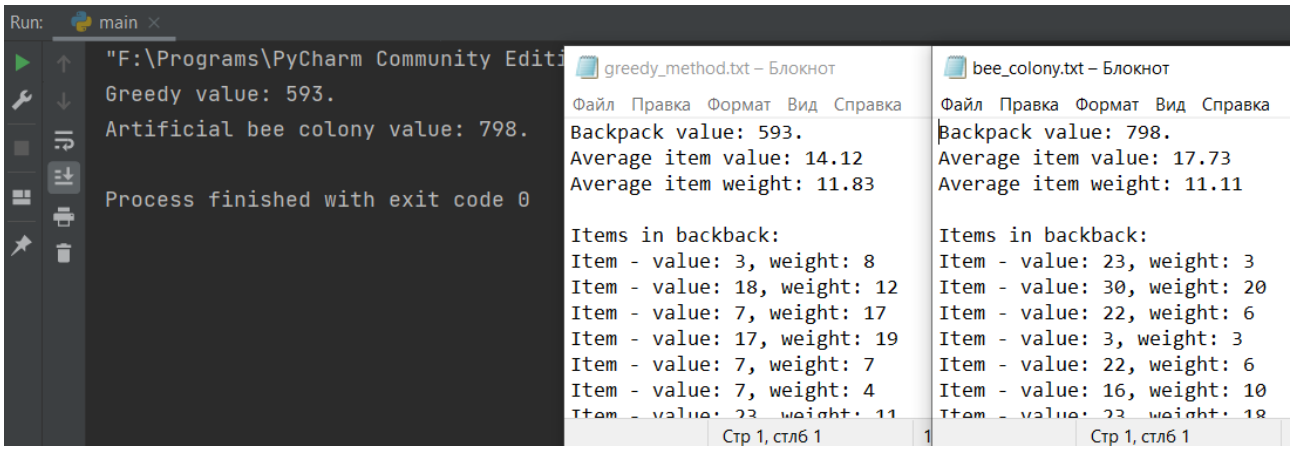


Рисунок 3.1 – Приклад роботи програми (5 фуражирів і 5 розвідників)

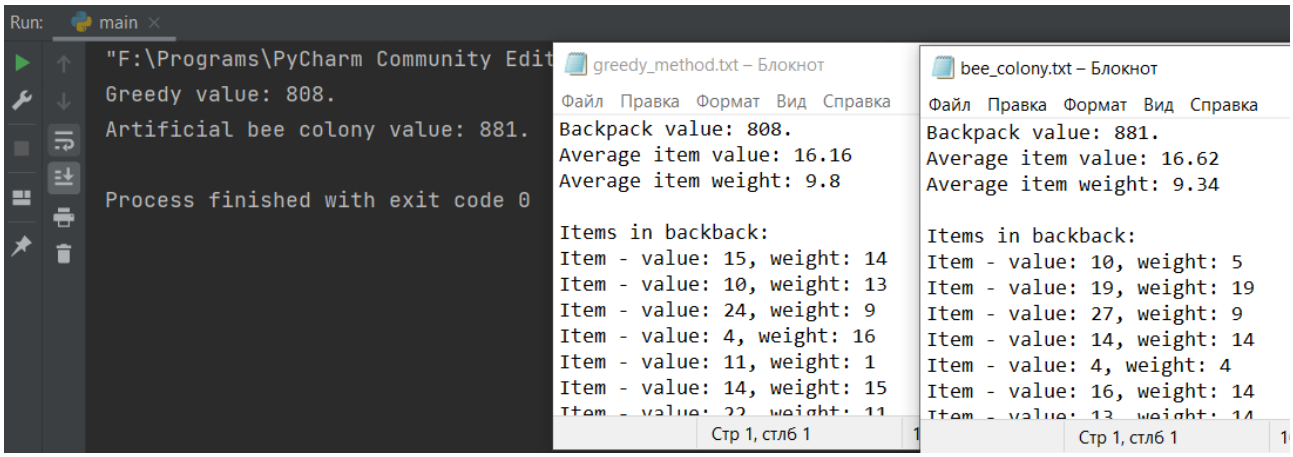


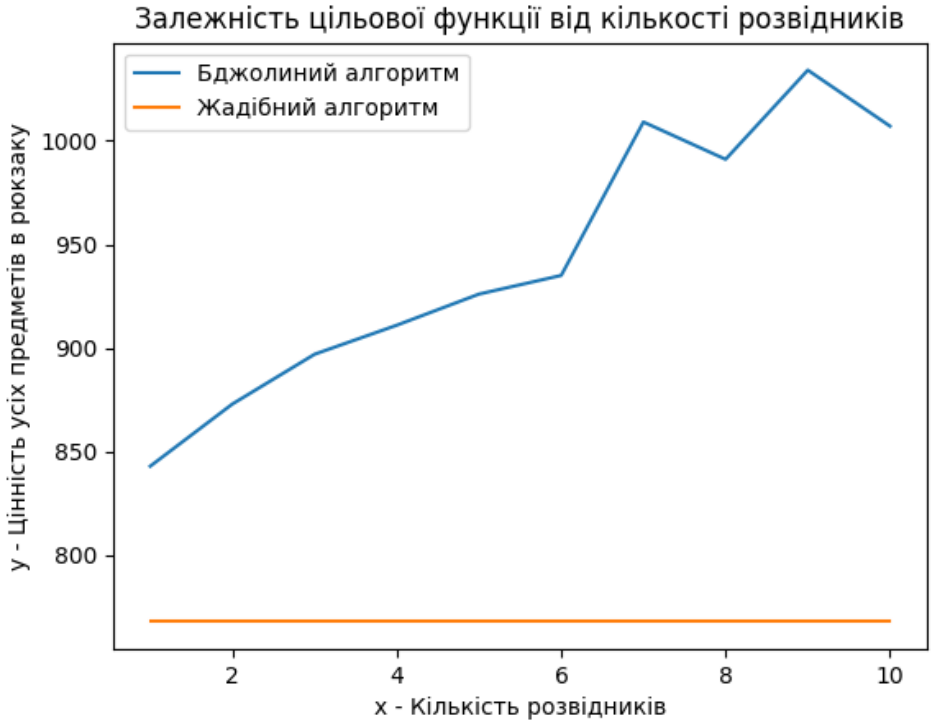
Рисунок 3.2 – Приклад роботи програми (20 фуражирів і 3 розвідники)

3.3 Тестування алгоритму

Для даної задачі (Задача про рюкзак) кількість ділянок визначається кількістю бджіл-розвідників, але не більше ніж кількість предметів (Якщо за умовою дано 3 розвідники, то ділянок для розвідування також буде 3. Якщо за умовою дано 150 розвідників, то ділянок для розвідування буде 100 (тобто максимальна кількість предметів)).

Кількість фуражирів: 10

Кількість розвідників	Цінність усіх предметів в рюкзаку
1	843
2	873
3	897
4	911
5	926
6	935
7	1009
8	991
9	1034
10	1007

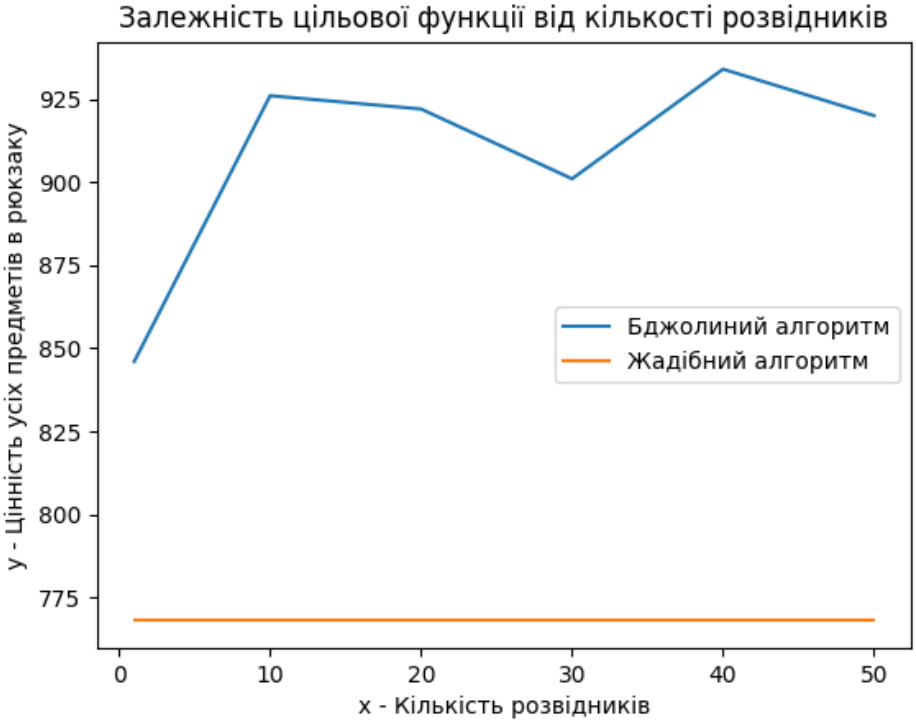


в

Рисунок 3.1 — Графік залежності цільової функції від кількості розвідників (Кількість фуражирів: 10)

Кількість фуражирів: 50

Кількість розвідників	Цінність усіх предметів в рюкзаку
1	846
10	926
20	922
30	901
40	934
50	920



в

Рисунок 3.2 — Графік залежності цільової функції від кількості розвідників (Кількість фуражирів: 50)

Кількість фуражирів: 100

Кількість розвідників	Цінність усіх предметів в рюкзаку
5	834
10	847
20	890
30	864
40	860
50	873
60	870
70	899
80	875
90	884
100	901

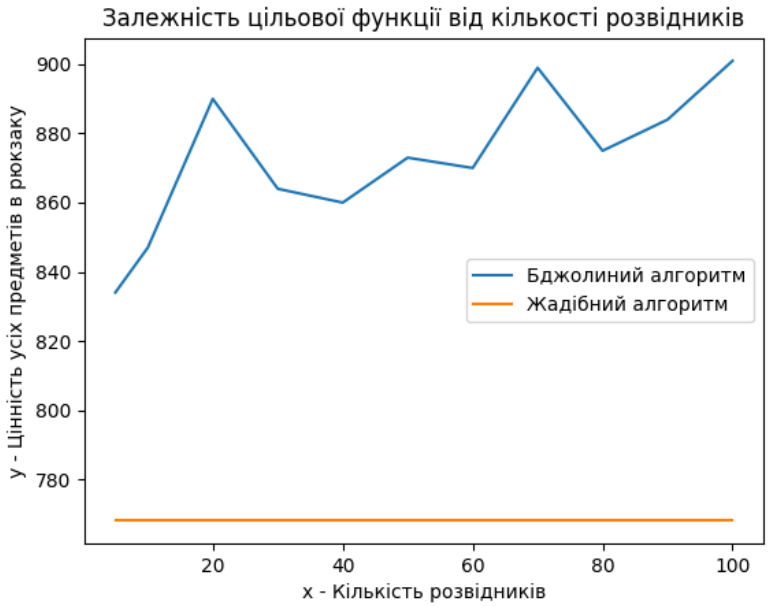


Рисунок 3.3 — Графік залежності цільової функції від кількості розвідників (Кількість фуражирів: 100)

Кількість розвідників: 10

Кількість фуражирів	Цінність усіх предметів в рюкзаку
1	1152
2	1041
3	1074
4	1019
5	1042
6	984
7	974
8	1013
9	958
10	946

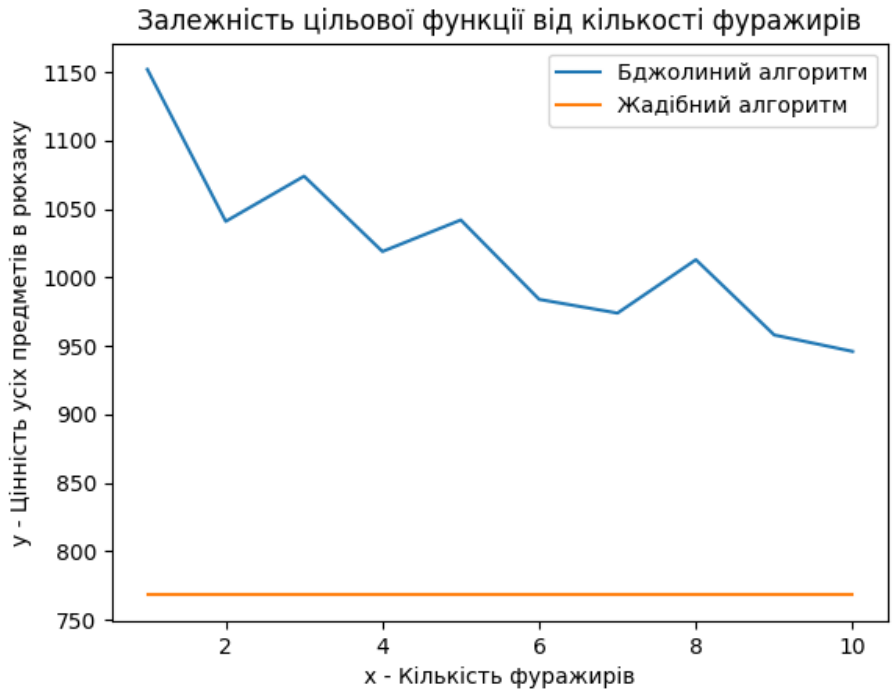
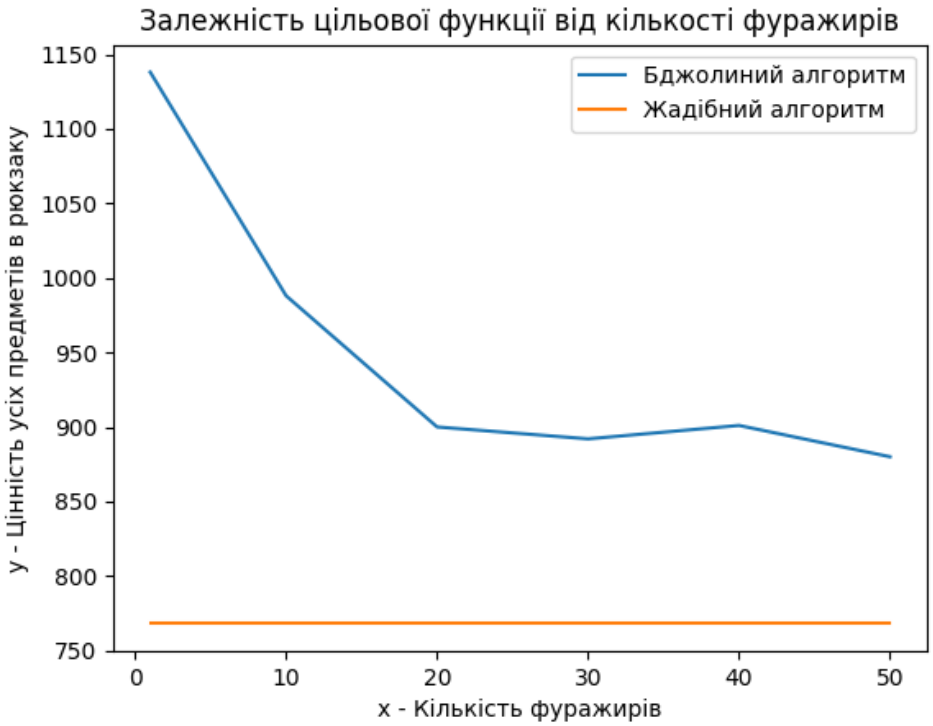


Рисунок 3.4 — Графік залежності цільової функції від кількості фуражирів (Кількість розвідників: 10)

Кількість розвідників: 50

Кількість фуражирів	Цінність усіх предметів в рюкзаку
1	1136
10	988
20	900
30	892
40	901
50	880



в

Рисунок 3.5 — Графік залежності цільової функції від кількості фуражирів (Кількість розвідників: 50)

ВИСНОВОК

В рамках даної лабораторної роботи я вивчив основні підходи розробки метаевристичних алгоритмів для типових прикладних задач. Опрацювати методологію підбору прийнятних параметрів алгоритму, формалізував алгоритм вирішення задачі відповідно загальної методології, записав розроблений алгоритм у покроковому вигляді.

Можу зробити висновок, що для вирішення задачі про рюкзак за допомогою бджолиного алгоритму найкращими параметрами є відносно невелика кількість розвідників (а саме ~10 штук) і кількість фуражирів залежна від кількості розвідників (10%-15% від кількості розвідників). Саме такі параметри дали нам найоптимальніший розв’язок (При кількості розвідників 10 штук і фуражирів 1 штука ми отримали загальну цінність рюкзаку 1152, що є найбільшим результатом в експериментах).

КРИТЕРІЇ ОЦІНЮВАННЯ

Критерії оцінювання у відсотках від максимального балу:

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