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
«Проєктування алгоритмів»

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

Київ 2024

ЗМІСТ

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 (допускається самостійній вибір кроку та верхньої границі ітерацій) 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 ($\alpha = 4$, $\beta = 2$, $\rho = 0,3$, 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 ($\alpha = 2$, $\beta = 4$, $\rho = 0,7$, 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 (α = 2, β = 4, ρ = 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 (α = 2, β = 3, ρ = 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

Файл “Main.py”:

```
import Config
import Evolutor
import Entity
import Item
import Drawer

from random import randint

def main():
    items = []
    entities = []

    for _ in range(Config.ITEMS_COUNT):
        items += [Item.Item(
            weight = randint(Config.WEIGHT_RANGE['min'],
Config.WEIGHT_RANGE['max']),
            value  = randint(Config.VALUE_RANGE ['min'], Config.VALUE_RANGE
['max'])
        )]

    for eid in range(Config.ENTITIES_COUNT):
        entities += [Entity.Entity(
            gens = [eid == iid for iid in range(Config.ITEMS_COUNT)]
        )]

    evolutor = Evolutor.Evolutor()

    evolutor.SetItems(items)
    evolutor.SetEntities(entities)

    evolutor.SetCapacity(Config.CAPACITY)
    evolutor.SetMutationChance(Config.MUTATION_CHANCE)
    evolutor.SetLocalOptimization(Config.LocallyOptimize)

    best_values = []

    for _ in range(Config.GENERATIONS_COUNT):
        evolutor.StepEvolution()

        best_values += [Evolutor.Helper.Fitness(evolutor.BestEntity, items,
Config.CAPACITY)]

    print(f'Best entity value: {Evolutor.Helper.Fitness(evolutor.BestEntity,
items, Config.CAPACITY)}')
```

```
    print(f'Best weight: {Evolutor.Helper.Weight(evolutor.BestEntity, items)}')

    drawer = Drawer.Drawer()
    drawer.SetBestValues(best_values)
    drawer.Draw()

if __name__ == '__main__':
    main()
else:
    print(f'File {__file__} can be used only as an executable')
    exit(1)
```

Файл “Drawer.py”:

```
if __name__ == '__main__':
    print(f'File {__file__} can be used only as a library')
    exit(1)

import matplotlib.pyplot as plt

# TODO
class Drawer:
    def __init__(self):
        pass

    def SetBestValues(self, best_values: list[int]):
        self.best_values = best_values

    def Draw(self):
        plt.plot(
            range(len(self.best_values)),
            self.best_values,
            label='Цінність рішення'
        )
        plt.legend()
        plt.show()
```

Файл “Item.py”:

```
if __name__ == '__main__':
    print(f'File {__file__} can be used only as a library')
    exit(1)

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

    @property
    def Weight(self) -> int:
        return self.weight

    @property
```

```
    def Value(self) → int:
        return self.value

    @property
    def Efficiency(self) → float:
        return self.value / self.weight
```

Файл “Entity.py”:

```
if __name__ == '__main__':
    print(f'File {__file__} can be used only as a library')
    exit(1)

from random import uniform, randint

class Entity:
    def __init__(self, *, gens=[]):
        self.gens = gens

    @property
    def Gens(self: list[int]):
        return self.gens

    @property
    def GensCount(self) → int:
        return len(self.gens)

    def Copy(self) → 'Entity':
        return Entity(gens = self.gens[:])

    def Mutate(self, chance: float) → dict:
        if uniform(0, 1) < chance:
            index = randint(0, self._gens_count() - 1)
            gens = self.gens

            gens[index] = 1 - gens[index]

            return {'is_mut': True}

        return {'is_mut': False}

    def GetMutated(self, chance: float) → tuple[bool, 'Entity']:
        if uniform(0, 1) < chance:
            index = randint(0, self._gens_count() - 1)

            new_gens = gens = self.gens[:]
            new_gens[index] = 1 - new_gens[index]

            return {'is_mut': True, 'entity': Entity(gens = new_gens)}

        return {'is_mut': False, 'entity': Entity(gens = self.gens[:])}

    def __repr__(self):
        return self._to_string()
```

```
    def __str__(self):
        return self._to_string()

    def _to_string(self):
        return ''.join('_#'[gen] for gen in self.gens)

    def _gens_count(self) → int:
        return len(self.gens)
```

Файл “Config.py”:

```
if __name__ == '__main__':
    print(f'File {__file__} can be used only as a library')
    exit(1)

import Entity
import Evolutor

from random import randint, choice

def LocallyOptimize(entity: Entity.Entity, items: list[Entity.Entity], capacity:
int):
    fitness_func = Evolutor.Helper.Fitness

    new_entity = entity.Copy()

    min_weight_item = None
    min_weight_item_index = None

    for iid, item in enumerate(items):
        if not entity.Gens[iid] and (not min_weight_item or item.Weight <
min_weight_item.Weight):
            min_weight_item = item
            min_weight_item_index = iid

    new_entity.Gens[min_weight_item_index] = 1

    if fitness_func(new_entity, items, capacity) > fitness_func(entity, items,
capacity):
        return new_entity

    return entity

GENERATIONS_COUNT = 600

CAPACITY     = 250
ITEMS_COUNT  = 100
VALUE_RANGE  = {'min': 2, 'max': 20}
WEIGHT_RANGE = {'min': 1, 'max': 10}

ENTITIES_COUNT  = 100
MUTATION_CHANCE = 0.05
```

Файл “Evolutor.py”:

```
if __name__ == '__main__':
    print(f'File {__file__} can be used only as a library')
    exit(1)

import Entity
import Item

import math
from typing import Callable, Optional
from random import choice, randint

class Helper:
    @staticmethod
    def Weight(entity: Entity.Entity, items: list[Item.Item]) → int:
        return sum(
            item.Weight if (entity.Gens[iid] == 1) else 0

            for iid, item in enumerate(items)
        )

    @staticmethod
    def Value(entity: Entity.Entity, items: list[Item.Item]) → int:
        return sum(
            item.Value if (entity.Gens[iid] == 1) else 0

            for iid, item in enumerate(items)
        )

    @staticmethod
    def Fitness(entity: Entity.Entity, items: list[Item.Item], capacity: int) →
int:
        return \
            Helper.Value(entity, items) \
            if Helper.Weight(entity, items) ⩽ capacity \
            else -1

    @staticmethod
    def Crossover(parent_a: Entity.Entity, parent_b: Entity.Entity) →
list[Entity.Entity]:
        separate_point = parent_a.GensCount // 2

        parent_a_fp_gens  = parent_a.Gens[:separate_point]
        parent_a_sp_gens  = parent_a.Gens[separate_point:]

        parent_b_fp_gens = parent_b.Gens[:separate_point]
        parent_b_sp_gens = parent_b.Gens[separate_point:]

        childs = []

        childs += [Entity.Entity(gens = parent_a_fp_gens + parent_b_sp_gens)]
        childs += [Entity.Entity(gens = parent_a_sp_gens + parent_b_fp_gens)]
```

```
    return childs

@staticmethod
def TryGetMutatedAndImprove(
    child_a: Entity.Entity,
    child_b: Entity.Entity,
    items: list[Item.Item],
    capacity: float,
    mutate_chance: float
) → Optional[Entity.Entity]:
    child_a_mut_info = child_a.GetMutated(mutate_chance)
    child_b_mut_info = child_b.GetMutated(mutate_chance)

    if not (child_a_mut_info['is_mut'] or child_b_mut_info['is_mut']):
        return {'is_mut_useful': False, 'entity': None}

    child_a_mut = child_a_mut_info['entity']
    child_b_mut = child_b_mut_info['entity']

    # choose best
    child_a_fitness = Helper.Fitness(child_a, items, capacity)
    child_b_fitness = Helper.Fitness(child_b, items, capacity)

    child_a_mut_fitness = Helper.Fitness(child_a_mut, items, capacity)
    child_b_mut_fitness = Helper.Fitness(child_b_mut, items, capacity)

    is_mutation_useless = True

    if child_a_mut_fitness ⩾ child_a_fitness:
        child_a_better = child_a_mut
        is_mutation_useless = False
    else:
        child_a_better = child_a

    if child_b_mut_fitness ⩾ child_b_fitness:
        child_b_better = child_b_mut
        is_mutation_useless = False
    else:
        child_b_better = child_b

    # mutation is useless
    if is_mutation_useless:
        return {'is_mut_useful': False, 'entity': None}

    child_a_better_fitness = Helper.Fitness(child_a_better, items, capacity)
    child_b_better_fitness = Helper.Fitness(child_b_better, items, capacity)

    if child_a_better_fitness > child_b_better_fitness:
        best_child = child_a_better
    if child_b_better_fitness > child_a_better_fitness:
        best_child = child_b_better
    else:
        best_child = choice([child_a_better, child_b_better])

    return {'is_mut_useful': True, 'entity': best_child}
```

```
    @staticmethod
    def GetWithBestFitness(
            child_a: Entity.Entity,
            child_b: Entity.Entity,
            items: list[Item.Item],
            capacity: int
    ) -> Entity.Entity:
        child_a_fitness = Helper.Fitness(child_a, items, capacity)
        child_b_fitness = Helper.Fitness(child_b, items, capacity)

        if child_a_fitness > child_b_fitness:
            return child_a
        elif child_b_fitness > child_a_fitness:
            return child_b
        else:
            return choice([child_a, child_b])

class Evolutor:
    def __init__(self):
        self.items = []
        self.entities = []
        self.best_entity = None

    def SetItems(self, items: list[Item.Item]):
        self.items = items

    def SetEntities(self, entities: list[Entity.Entity]):
        self.entities = entities

    def SetCapacity(self, capacity: int):
        self.capacity = capacity

    def SetMutationChance(self, chance: float):
        self.mutation_chance = chance

    def SetLocalOptimization(self, local_optimizer: Callable[[Entity.Entity],
None]):
        self.local_optimizer = local_optimizer

    @property
    def BestEntity(self):
        return self.best_entity

    def StepEvolution(self):
        entities = self.entities
        items = self.items

        capacity = self.capacity

        mutate_chance = self.mutation_chance

        # choose a parent
        if True:
            parent_a = None
```

```
            for entity in entities:
                if not (parent_a ≠ entity):
                    continue

                if not parent_a or Helper.Fitness(entity, items, capacity) >
Helper.Fitness(parent_a, items, capacity):
                    parent_a = entity

            parent_b = choice(entities)

            while parent_a == parent_b:
                parent_b = choice(entities)

        # bear childs
        childs = Helper.Crossover(parent_a, parent_b)

        child_a = childs[0]
        child_b = childs[1]

        if not (child_a ≠ -1 or child_b ≠ -1):
            return

        # mutate
        mut_child_info = Helper.TryGetMutatedAndImprove(child_a, child_b, items,
capacity, mutate_chance)

        if mut_child_info['is_mut_useful']:
            mut_child = mut_child_info['entity']
            best_child = mut_child
        else:
            best_candidate = Helper.GetWithBestFitness(child_a, child_b, items,
capacity)

            best_child = self.local_optimizer(
                best_candidate,
                items,
                capacity
            )

        if not self.best_entity \
                or Helper.Fitness(best_child, items, capacity) >
Helper.Fitness(self.best_entity, items, capacity):
            self.best_entity = best_child

        # add to population and remove worst
        entities += [best_child]

        if True:
            worst_index   = None
            worst_fitness = math.inf

            for eid, entity in enumerate(entities):
                if (fitness := Helper.Fitness(entity, items, capacity)) <
worst_fitness:
```

```
                worst_fitness = fitness
                worst_index = eid

        if worst_index ≠ None:
            del entities[worst_index]
```

3.1.2 Вихідний код розв’язання задачі рюкзака з використанням динамічного програмування.

```
from random import randint

# Генеруємо випадкові предмети
items = [{'value': randint(2, 20), 'weight': randint(1, 10)} for _ in
range(100)]

# Ємність рюкзака
capacity = 250

# Ініціалізуємо таблицю DP
dp = [0] * (capacity + 1)

# Алгоритм динамічного програмування
for item in items:
    value, weight = item['value'], item['weight']
    for c in range(capacity, weight - 1, -1):  # Перебір від кінця, щоб уникнути
перезаписів
        dp[c] = max(dp[c], dp[c - weight] + value)

# Максимальна цінність
print("Максимальна цінність:", dp[capacity])
```

3.1.1 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми. Параметрами алгоритму є розмір рюкзака 250, кількість предметів 100, розмір популяції 100, одноточкове схрещування, шанс 1 мутації 5%, кількість кроків еволюції 2400.

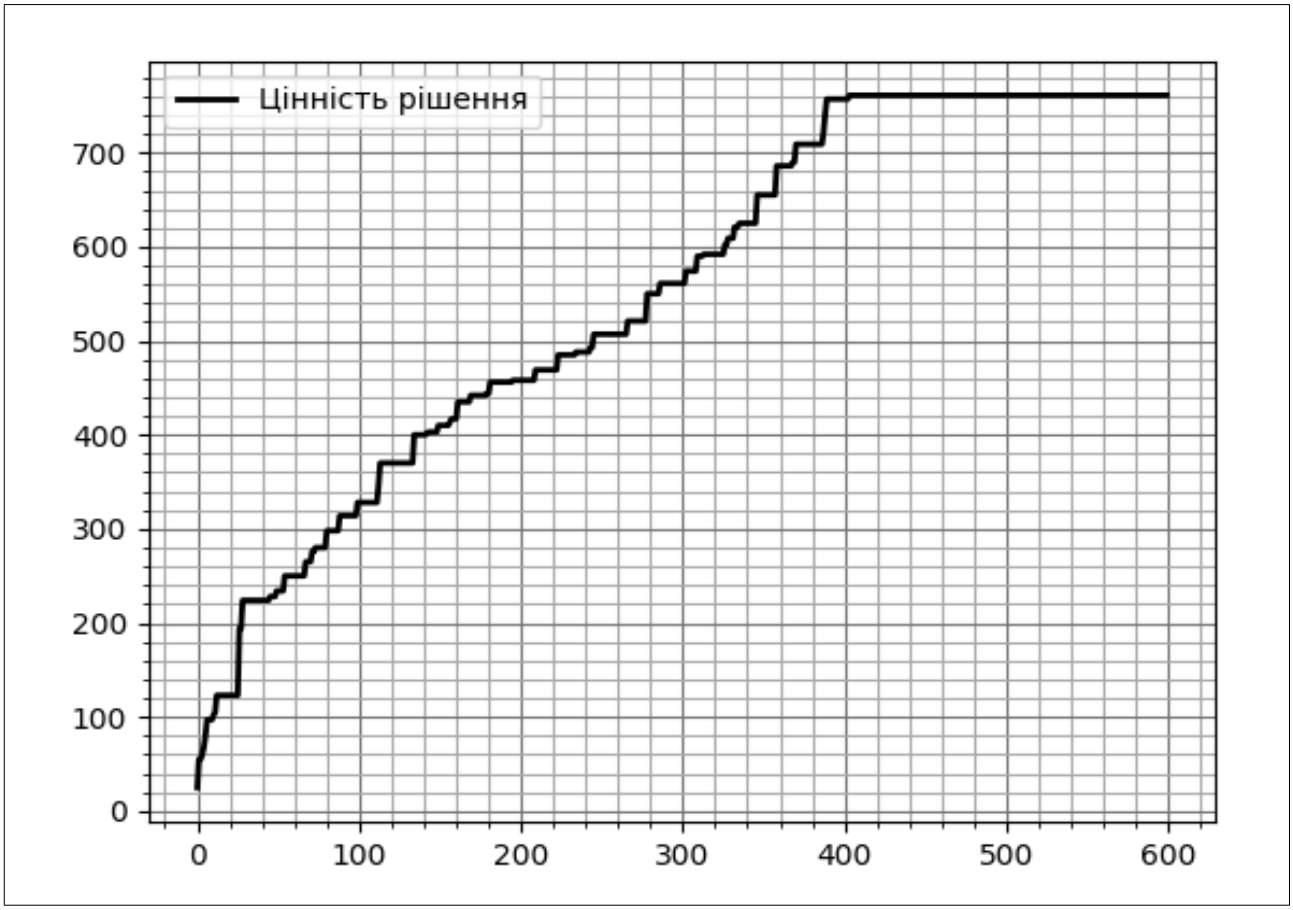

Рисунок 3.1 – показ цінності рішення на кожній ітерації алогритму

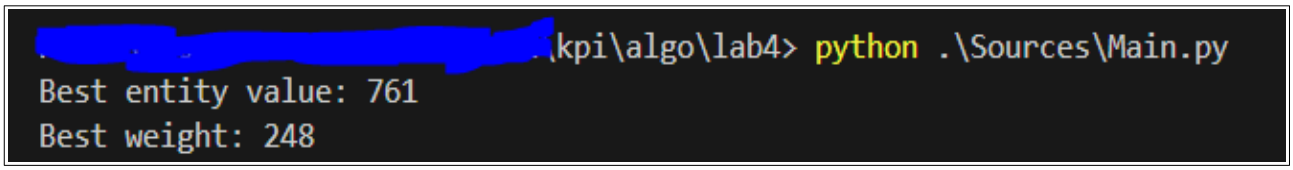

Рисунок 3.2 – найкраще рішення або його шлях досягнення та абсолютно найкраще числове рішення

На рисунку 3.1 та 3.2 видно, що найкращим рішенням задачі про рюкзак є 815, використовуючи генетичний алгоритм. Гени, або рішення чи шлях досягнення рішення, зображені на рисунку 3.2 для огляду кінцевого результату виконання цього алгоритму для більш повної картини розуміння цього алгоритму.

Його ефективним застосуванням є знаходження напів-оптимального рішення з великої множини всіх рішень. При знаходженні рішення, найбільш близького до оптимального, він починає дуже повільно працювати через те, що він намагається лише покращити поточну популяцію для створення нової на її основі, опираючись на рішення, яке може бути локальним, а не глобальним максимумом до найкращого рішення такої складної задачі як задача про рюкзак.

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.

Номер ітерації	Якість
1	25
11	102
21	123
31	224
41	224
51	234
61	250
71	265
81	298
91	314
101	328
111	328
121	370
131	370
141	400
151	410
161	417
171	442
181	444

191	456
201	458
211	469
221	469
231	485
241	488
251	507
261	507
271	521
281	550
291	561
301	561
311	590
321	592
331	609
341	625
351	655
361	686
371	709
381	709
391	757
401	757
411	761
421	761

431	761
441	761
451	761
461	761
471	761
481	761
491	761
501	761
511	761
521	761
531	761
541	761
551	761
561	761
571	761
581	761
591	761

3.2.2 Графіки залежності розв’язку від числа ітерацій

На рисунку 3.3 наведений графік, який показує якість отриманого розв’язку. Чим ближче знаходиться графік до значення по ординаті 195 (середнє значення максимальної цінності рюкзака та предметів), тим більш оптимальним є розв’язок генетичним алгоритмом.

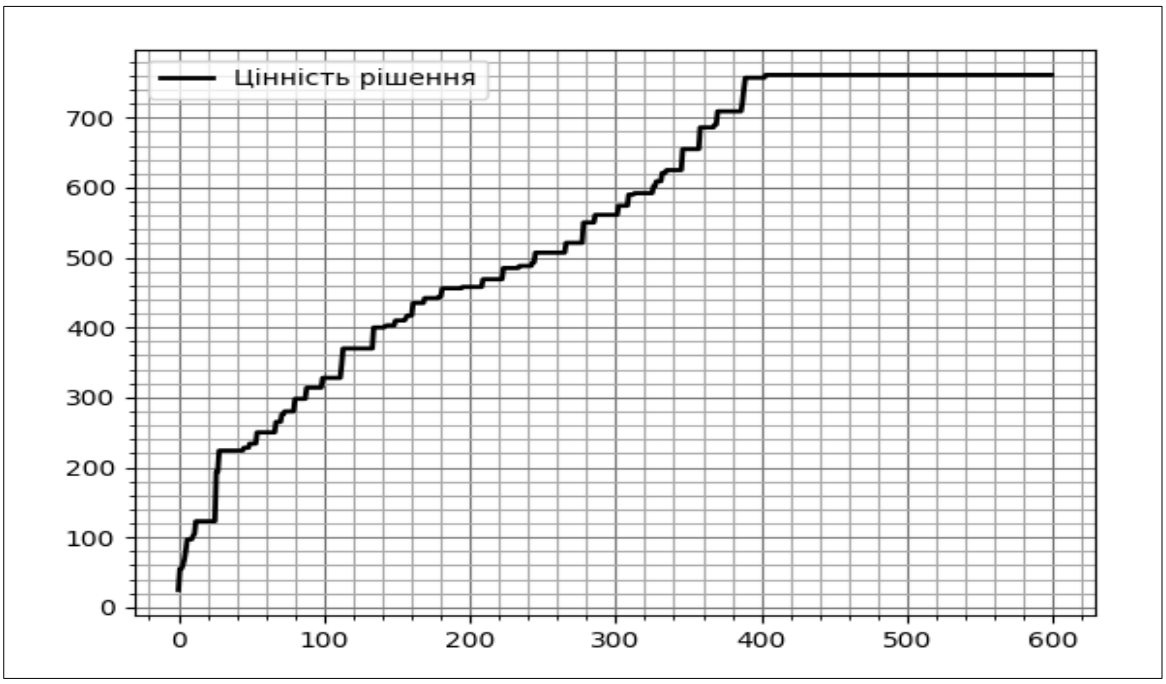

Рисунок 3.3 – Графіки залежності розв’язку від числа ітерацій

В даному випадку генетичний алгоритм непогано справляється з задачею про рюкзак, але він довго добирається до плато, тому він трохи неефективний в даному плані для задачі про рюкзак.

Судячи по кривій якості розв’язку, він добре підходить для знаходження напів-оптимальних розв’язків задач через його природу.

Під час реалізації алгоритму використовуючи свою стратегію відбору[1] батьків, створення нового покоління, та використовуючи більш агресивну функцію локального пошуку було помічено, що воно було більш ефективним для виходу на плато та до більш оптимального рішення, а ніж варіант даний лектором під час лекції з описом цього алгоритму та застосування його в цій задачі на практичному прикладі.

ВИСНОВОК

В рамках даної лабораторної роботи я нарешті зміг проаналізувати, що таке генетичні алгоритми та як їх можна використовувати для вирішення складних задач, як задача про рюкзак, та як він працює в теорії та на практичному прикладі та досвіді.

Було виконано програмну реалізацію генетичного алгоритму для задачі про рюкзак з використанням певних налаштувань програмного забезпечення. Під час тестування було помічено, що його легко перевірити на коректність та послідовність виконання. Функція локального пошуку індивідів є однією з найбільш цікавих та складних в плані комплексності та багатогранності функції цього мета-евристичного алгоритму.

Результат викання програмної реалізації цього алгоритму є досить приємним та швидким. Він дозволяє отримати все більш оптимальні рішення з плином виконання головного циклу виконання цього алгоритму (найкраще значення розв’язку для рюкзака постійно збільшується).

Абсолютне найкраще рішення задачі про рюкзак з кожною ітерацією становиться все кращим, але воно досягає плато після стрімкого початкового збільшення реального значення цінності розв’язку задачі при випадкових вхідних параметрах. В порівнянні з еталонним алгоритмом (той, що використовує динамічне програмування) він дає хороше рішення через перші 400 ітерацій, але потім він знаходить більш оптимальні рішення все більш повільніше при спробі досягти найбільш якісне рішення задачі.

Цей алгоритм, як і інші мета-евристичні, дає змогу зрозуміти, що задача може бути вирішена навіть напів-оптимально та використовуючи нестандартні техніки та стратегії досягнення хорошого результату для обраної задачі.

ПОСИЛАННЯ

1. Своя перша реалізація стратегії відбору батьків та стратегія створення нового покоління URL: https://arpitbhayani.me/blogs/genetic-knapsack/

КРИТЕРІЇ ОЦІНЮВАННЯ

Критерії оцінювання у відсотках від максимального балу:

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