**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
«Проєктування алгоритмів»

„**Проєктування і аналіз алгоритмів для вирішення NP-складних задач ч.2**”

**Виконав(ла)** *ІП-35 Адаменко Арсен Богданович* _____
(шифр, прізвище, ім'я, по батькові)

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

Київ 2024

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 з варіантом, формалізувати алгоритм вирішення задачі відповідно до 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 кращі значення вхідних параметрів для даної задачі або встановлена залежність одних параметрів від інших.

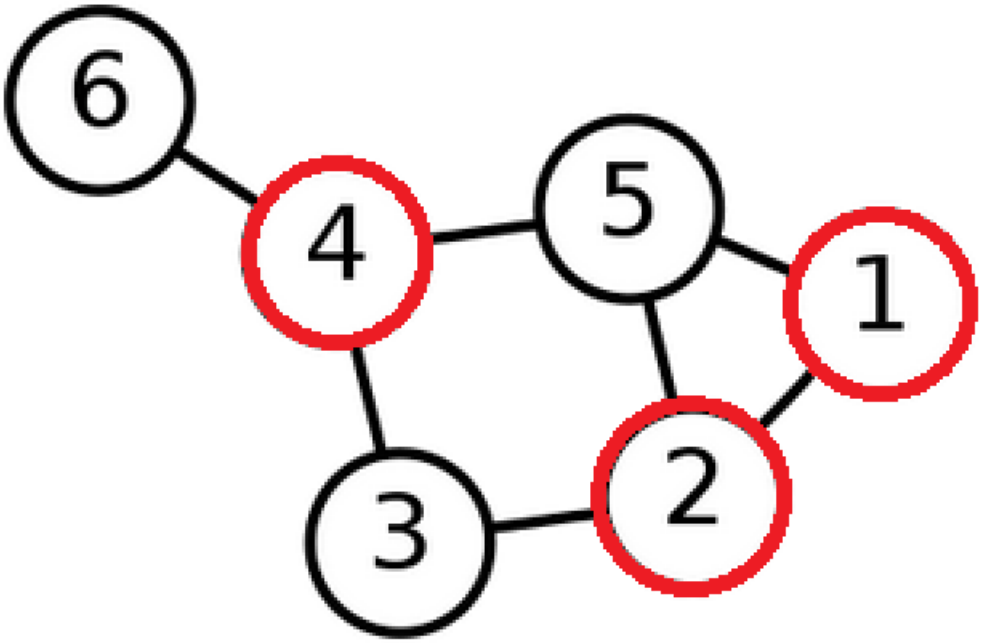
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

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.<br>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.<br>**Розглядається симетричний, асиметричний та змішаний варіанти.**<br>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.<br>У випадку симетричної задачі всі пари ребер між одними й тими самими вершинами мають однакову вагу.<br>У випадку реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості або довжини маршрутів і напряму руху.<br>Застосування:<br>- доставка товарів (в цьому випадку може бути більш доречна постановка транспортної задачі - доставка в кілька магазинів з декількох складів);<br>- доставка води;<br>- моніторинг об'єктів; |

| | |
|---|---|
| | - поповнення банкоматів готівкою;<br>- 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.<br>Застосування:<br>- розкладу для освітніх установ;<br>- розкладу в спорті;<br>- планування зустрічей, зборів, інтерв'ю;<br>- розклади транспорту, в тому числі - авіатранспорту;<br>- 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.<br>Задача вершинного покриття полягає в пошуку вершинного покриття найменшого розміру для заданого графа (цей розмір називається числом вершинного покриття графа).<br>На вході: Граф G = (V, E).<br>Результат: множина C ⊆ V - найменше вершинне покриття графа G. |

| | |
|---|---|
| |  <br>Застосування:<br>- розміщення пунктів обслуговування;<br>- призначення екіпажів на транспорт;<br>- 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.<br>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).<br>Застосування:<br>- біоінформатика;<br>- 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.<br>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:**<br>- оператор схрещування (мінімум 3);<br>- мутація (мінімум 2);<br>- оператор локального покращення (мінімум 2). |
| 2 | **Мурашиний алгоритм**:<br>- α;<br>- β;<br>- ρ;<br>- Lmin;<br>- кількість мурах M і їх типи (елітні, тощо…);<br>- маршрути з однієї чи різних вершин. |
| 3 | **Бджолиний алгоритм:**<br>- кількість ділянок;<br>- 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

## 3.1 Покроковий алгоритм

Нижче зображено псевдокод-текстовий опис усіх варіацій параметрів генетичного алгоритму при вирішення задачі з рюкзаком.

**Налаштування мутації №1:**

Шанс мутації є нульовим (0%).

**Налаштування мутації №2:**

Шанс мутації є 5%.

**Налаштування мутації №3:**

Шанс мутації є 10%.

**Налаштування мутації №4:**

Шанс мутації є 20%.

**Налаштування мутації №5:**

Шанс мутації є результатом підкидання монети (50%).

**Налаштування мутації №6:**

Шанс мутації є стовідсотковим (100%).

**Оператор кросоверу №1:**

Цей оператор є одноточковим зі значенням розділення 50%:

```
child_a = parent_a[0%:50%] + parent_b[50%:100%]
child_b = parent_b[0%:50%] + parent_a[50%:100%]
childs = [child_a, child_b]
```

**Оператор кросоверу №2:**

Цей оператор є одноточковим зі значенням розділення 50%:

```
child_a = parent_a[0%:50%] + parent_b[50%:100%]
child_b = parent_b[0%:50%] + parent_a[50%:100%]
childs = [child_a, child_b]
```

**Оператор кросоверу №3:**

Цей оператор є оператором кросоверу з двохточковим розділенням:

```
child_a = parent_a[0%:33%] + parent_b[33%:66%] + parent_a[66%:100%]
child_b = parent_b[0%:33%] + parent_a[33%:66%] + parent_b[66%:100%]
childs = [child_a, child_b]
```

**Оператор кросоверу №4:**

Цей оператор є оператором кросоверу з трьохточковим розділенням:

```
      child_a = parent_a[0%:25%] + parent_b[25%:50%] + parent_a[50%:75%] +
parent_b[75%:100%]
      child_b = parent_b[0%:25%] + parent_a[25%:50%] + parent_b[50%:75%] +
parent_a[75%:100%]
      childs = [child_a, child_b]
```

**Оператор локального покращення №1:**

Він вставляє предмет з найменшою вагою, якщо при його вставці рюкзак не буде переповненим. Потім вибирається особина з хромосомою, що є найкращою з поточної та з нової після оптимізації.

**Оператор локального покращення №2:**

Він робить кілька випадкових спроб видалення предмету з рюкзака, якщо це вдається, він його видаляє. Потім намагаємося встанити кілька найбільш корисних предметів з точки зору ціна/якість (чим більш цінний та менш важкий, тим краще). Потім вибирається особина з хромосомою, що є найкращою з поточної та з нової після оптимізації.

## 3.2 Програмна реалізація алгоритму

### 3.2.1 Вихідний код

**Вихідний файл «Source/Config.py»:**

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

import Item
import Entity
import Evolutor

from random import randint, choice, uniform

PHI = (1 + 5**0.5) / 2

def CrossoverA(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

def CrossoverB(parent_a: Entity.Entity, parent_b: Entity.Entity) →
list[Entity.Entity]:
    gens_count = parent_a.GensCount

    separate_point = int(gens_count / PHI)

    if uniform(0, 1) < 0.5:
        separate_point = gens_count - separate_point

    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

def CrossoverC(parent_a: Entity.Entity, parent_b: Entity.Entity) →
list[Entity.Entity]:
    separate_point_a = parent_a.GensCount // 3
    separate_point_b = separate_point_a * 2

    parent_a_p1_gens  = parent_a.Gens[                :separate_point_a]
    parent_a_p2_gens  = parent_a.Gens[separate_point_a:separate_point_b]
    parent_a_p3_gens  = parent_a.Gens[separate_point_b:                ]

    parent_b_p1_gens  = parent_b.Gens[                :separate_point_a]
    parent_b_p2_gens  = parent_b.Gens[separate_point_a:separate_point_b]
    parent_b_p3_gens  = parent_b.Gens[separate_point_b:                ]

    childs = []

    childs += [Entity.Entity(gens = parent_a_p1_gens + parent_b_p2_gens +
parent_a_p3_gens)]
    childs += [Entity.Entity(gens = parent_b_p1_gens + parent_a_p2_gens +
parent_b_p3_gens)]

    return childs

def CrossoverD(parent_a: Entity.Entity, parent_b: Entity.Entity) →
list[Entity.Entity]:
    separate_point_a = parent_a.GensCount // 4
    separate_point_b = separate_point_a * 2
    separate_point_c = separate_point_a * 3

    parent_a_p1_gens  = parent_a.Gens[                :separate_point_a]
    parent_a_p2_gens  = parent_a.Gens[separate_point_a:separate_point_b]
    parent_a_p3_gens  = parent_a.Gens[separate_point_b:separate_point_c]
    parent_a_p4_gens  = parent_a.Gens[separate_point_c:                ]

    parent_b_p1_gens  = parent_b.Gens[                :separate_point_a]
```

```
    parent_b_p2_gens  = parent_b.Gens[separate_point_a:separate_point_b]
    parent_b_p3_gens  = parent_b.Gens[separate_point_b:separate_point_c]
    parent_b_p4_gens  = parent_b.Gens[separate_point_c:                ]

    childs = []

    childs += [Entity.Entity(gens = parent_a_p1_gens + parent_b_p2_gens +
parent_a_p3_gens + parent_b_p4_gens)]
    childs += [Entity.Entity(gens = parent_b_p1_gens + parent_a_p2_gens +
parent_b_p3_gens + parent_a_p4_gens)]

    return childs

def LocalOptimizerA(entity: Entity.Entity, items: list[Entity.Entity],
capacity: int) → Entity.Entity:
    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

_best_eff_items = None

def LocalOptimizerB(entity: Entity.Entity, items: list[Entity.Entity],
capacity: int) → Entity.Entity:
    global _best_eff_items

    fitness_func = Evolutor.Helper.Fitness

    gens_count = entity.GensCount
    new_entity = entity.Copy()
    new_weight = fitness_func(new_entity, items, capacity)

    for _ in range(min(6, gens_count)):
        rem_idx = randint(0, gens_count - 1)

        if entity.Gens[rem_idx] == 1:
            entity.Gens[rem_idx] = 0
            new_weight -= items[rem_idx].Weight

    adds_count = 0
    for item_info in _best_eff_items[min(24, gens_count):]:
        item = item_info['item']
```

```
        index = item_info['index']

        if adds_count < 8:
            break

        if entity.Gens[index] == 0 and new_weight + item.Weight <= capacity:
            entity.Gens[index] = 1
            new_weight += item.Weight

            adds_count += 1

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

    return entity

def SetItems(items: list[Item.Item]):
    global _best_eff_items

    _best_eff_items = sorted(
        [
            [{
                'index': i,
                'item': item
            }]
            for i, item in enumerate(items)
        ],
        key = lambda ii: ii['item'].Efficiency
    )

STABILITY_THRESHOLD_VALUE = 100

CAPACITY        = 500
ITEMS_COUNT     = 100
VALUE_RANGE     = {'min': 2, 'max': 30}
WEIGHT_RANGE    = {'min': 1, 'max': 20}
ENTITIES_COUNT  = 100

MUTATION_CHANCES = {
    '0.00': 0.00,
    '0.05': 0.05,
    '0.10': 0.10,
    '0.20': 0.20,
    '0.50': 0.50,
    '1.00': 0.1
}
CROSSOVERS       = {
    'Half':      CrossoverA,
    'Moreless':  CrossoverB,
    'ThreeAsym': CrossoverC,
    'Four':      CrossoverD
}
LOCAL_OPTIMIZERS = {
    'AddLightest': LocalOptimizerA,
    'RemSevAddSevEff': LocalOptimizerB
}
```

**Вихідний файл «Source/Drawer.py»:**

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

import Item
import Entity
import Evolutor

from random import randint, choice, uniform

PHI = (1 + 5**0.5) / 2

def CrossoverA(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

def CrossoverB(parent_a: Entity.Entity, parent_b: Entity.Entity) →
list[Entity.Entity]:
    gens_count = parent_a.GensCount

    separate_point = int(gens_count / PHI)

    if uniform(0, 1) < 0.5:
        separate_point = gens_count - separate_point

    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

def CrossoverC(parent_a: Entity.Entity, parent_b: Entity.Entity) →
list[Entity.Entity]:
    separate_point_a = parent_a.GensCount // 3
    separate_point_b = separate_point_a * 2

    parent_a_p1_gens  = parent_a.Gens[               :separate_point_a]
```

```
    parent_a_p2_gens  = parent_a.Gens[separate_point_a:separate_point_b]
    parent_a_p3_gens  = parent_a.Gens[separate_point_b:                ]

    parent_b_p1_gens  = parent_b.Gens[                :separate_point_a]
    parent_b_p2_gens  = parent_b.Gens[separate_point_a:separate_point_b]
    parent_b_p3_gens  = parent_b.Gens[separate_point_b:                ]

    childs = []

    childs += [Entity.Entity(gens = parent_a_p1_gens + parent_b_p2_gens +
parent_a_p3_gens)]
    childs += [Entity.Entity(gens = parent_b_p1_gens + parent_a_p2_gens +
parent_b_p3_gens)]

    return childs

def CrossoverD(parent_a: Entity.Entity, parent_b: Entity.Entity) →
list[Entity.Entity]:
    separate_point_a = parent_a.GensCount // 4
    separate_point_b = separate_point_a * 2
    separate_point_c = separate_point_a * 3

    parent_a_p1_gens  = parent_a.Gens[                :separate_point_a]
    parent_a_p2_gens  = parent_a.Gens[separate_point_a:separate_point_b]
    parent_a_p3_gens  = parent_a.Gens[separate_point_b:separate_point_c]
    parent_a_p4_gens  = parent_a.Gens[separate_point_c:                ]

    parent_b_p1_gens  = parent_b.Gens[                :separate_point_a]
    parent_b_p2_gens  = parent_b.Gens[separate_point_a:separate_point_b]
    parent_b_p3_gens  = parent_b.Gens[separate_point_b:separate_point_c]
    parent_b_p4_gens  = parent_b.Gens[separate_point_c:                ]

    childs = []

    childs += [Entity.Entity(gens = parent_a_p1_gens + parent_b_p2_gens +
parent_a_p3_gens + parent_b_p4_gens)]
    childs += [Entity.Entity(gens = parent_b_p1_gens + parent_a_p2_gens +
parent_b_p3_gens + parent_a_p4_gens)]

    return childs

def LocalOptimizerA(entity: Entity.Entity, items: list[Entity.Entity],
capacity: int) → Entity.Entity:
    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

_best_eff_items = None

def LocalOptimizerB(entity: Entity.Entity, items: list[Entity.Entity],
capacity: int) → Entity.Entity:
    global _best_eff_items

    fitness_func = Evolutor.Helper.Fitness

    gens_count = entity.GensCount
    new_entity = entity.Copy()
    new_weight = fitness_func(new_entity, items, capacity)

    for _ in range(min(6, gens_count)):
        rem_idx = randint(0, gens_count - 1)

        if entity.Gens[rem_idx] == 1:
            entity.Gens[rem_idx] = 0
            new_weight -= items[rem_idx].Weight

    adds_count = 0
    for item_info in _best_eff_items[min(24, gens_count):]:
        item = item_info['item']
        index = item_info['index']

        if adds_count < 8:
            break

        if entity.Gens[index] == 0 and new_weight + item.Weight ⩽ capacity:
            entity.Gens[index] = 1
            new_weight += item.Weight

            adds_count += 1

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

    return entity

def SetItems(items: list[Item.Item]):
    global _best_eff_items

    _best_eff_items = sorted(
        [
            [{
                'index': i,
                'item': item
            }]
            for i, item in enumerate(items)
        ],
        key = lambda ii: ii['item'].Efficiency
    )
```

```
STABILITY_THRESHOLD_VALUE = 100

CAPACITY        = 500
ITEMS_COUNT     = 100
VALUE_RANGE     = {'min': 2, 'max': 30}
WEIGHT_RANGE    = {'min': 1, 'max': 20}
ENTITIES_COUNT  = 100

MUTATION_CHANCES = {
    '0.00': 0.00,
    '0.05': 0.05,
    '0.10': 0.10,
    '0.20': 0.20,
    '0.50': 0.50,
    '1.00': 0.1
}
CROSSOVERS       = {
    'Half':      CrossoverA,
    'Moreless':  CrossoverB,
    'ThreeAsym': CrossoverC,
    'Four':      CrossoverD
}
LOCAL_OPTIMIZERS = {
    'AddLightest': LocalOptimizerA,
    'RemSevAddSevEff': LocalOptimizerB
}
```

**Вихідний файл «Source/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)
```

**Вихідний файл «Source/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
```

**Вихідний файл «Source/Main.py»:**

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

from random import randint
from typing import Callable, Dict, List
```

```
def MakeExperiment(
    *,
    items: list[Item.Item],
    mutation_chance: float,
    crossover: Callable[[Entity.Entity, Entity.Entity], list[Entity.Entity]],
    local_optimizer: Callable[[Entity.Entity, list[Item.Item], int],
Entity.Entity]
):
    entities = []

    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(mutation_chance)
    evolutor.SetCrossover(crossover)
    evolutor.SetLocalOptimization(local_optimizer)

    best_values = []

    stability_length = 0
    stability_value = -1

    while True:
        evolutor.StepEvolution()

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

        if best_values[-1] > stability_value:
            stability_value = best_values[-1]
            stability_length = 0
        else:
            if not (stability_length ⩽ Config.STABILITY_THRESHOLD_VALUE):
                break
            else:
                stability_length += 1

    return best_values

def PrintBestStat(conf_name: str, results: Dict[str, List[int]]):
    print(f'Best statistic for configurable parameter <{conf_name}>:')

    index = 1
    for var in results:
        print(f'{index: >3}: version = <{var: >16}>, best result =
<{results[var][-1]: >5}>')
        index += 1

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

```
    entities = []

    Config.SetItems(items)

    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'])
        )]

    results_of_mutations  = {}
    results_of_crossovers = {}
    results_of_local_optimizers = {}

    for mutation_chance in Config.MUTATION_CHANCES:
        best_values = MakeExperiment(
            items = items,
            mutation_chance = Config.MUTATION_CHANCES[mutation_chance],
            crossover = Config.CROSSOVERS['Half'],
            local_optimizer = Config.LOCAL_OPTIMIZERS['AddLightest']
        )

        results_of_mutations[mutation_chance] = best_values

    for crossover in Config.CROSSOVERS:
        best_values = MakeExperiment(
            items = items,
            mutation_chance = Config.MUTATION_CHANCES['0.05'],
            crossover = Config.CROSSOVERS[crossover],
            local_optimizer = Config.LOCAL_OPTIMIZERS['AddLightest']
        )

        results_of_crossovers[crossover] = best_values

    for local_optimizer in Config.LOCAL_OPTIMIZERS:
        best_values = MakeExperiment(
            items = items,
            mutation_chance = Config.MUTATION_CHANCES['0.05'],
            crossover = Config.CROSSOVERS['Half'],
            local_optimizer = Config.LOCAL_OPTIMIZERS[local_optimizer]
        )

        results_of_local_optimizers[local_optimizer] = best_values

    PrintBestStat('mutation chance', results_of_mutations)
    print()
    PrintBestStat('crossover method', results_of_crossovers)
    print()
    PrintBestStat('local optimizer', results_of_local_optimizers)

    drawer = Drawer.Drawer()

    drawer.AddNewComparison(
        'mutation chance',
        results_of_mutations,
        [
```

```
            'purple',
            'navy',
            'darkgreen',
            'olive',
            'darkorange',
            'darkred'
        ]
    )
    drawer.AddNewComparison(
        'crossover method',
        results_of_crossovers,
        [
            'darkred',
            'olive',
            'darkgreen',
            'navy'
        ]
    )
    drawer.AddNewComparison(
        'local optimizer',
        results_of_local_optimizers,
        [
            'navy',
            'darkgreen'
        ]
    )

    drawer.Draw()

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

**Вихідний файл «Source/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 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])

    @staticmethod
    def ChooseTwoParents(entities, items, capacity):
        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)

        return [parent_a, parent_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 SetCrossover(self, crossover: Callable[[Entity.Entity,
list[Item.Item], int], Entity.Entity]):
        self.crossover = crossover

    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
        parents = Helper.ChooseTwoParents(entities, items, capacity)

        parent_a = parents[0]
        parent_b = parents[1]

        # bear childs
        childs = self.crossover(parent_a, parent_b)

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

        # it is always removed and nothing changes
        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]
```

### 3.2.2 Приклади роботи

Під час аналізу задачі рюкзака було виявлено, що середня максимальна цінність всіх предметів у непереповненому рююкзаку є 1229.928 (1230) при виконанні 100 разів на різних значень предметів при одній і тієїж самій конфігурації. На рисунках 3.1 і 3.2 показані приклади роботи програми.

Для визначення, чи є кількість пророблених ітерацій еволюції, було зроблено евристику, яка на кожному кроці визначає, чи є покращення цінності найкращого члена популяції та скидає лічильник незмінності поточної цінності найкращого члена популяції. За допомогою лічильника незмінності поточної цінності найкращого члена популяції можна визначити, як довго встановлена конфігурація параметрів не дає жодного покращення. Тому якщо він перевищує певне число, то генетичний алгоритм зупиняється, бо шанс знаходження більш цінного рішення є дуже низькою, проте кількість зекономлених обчислювальних ресурсів може бути значною.

```
Best statistic for configurable parameter <mutation chance>:
  1: version = <            0.00>, best result = < 1083>
  2: version = <            0.05>, best result = < 1079>
  3: version = <            0.10>, best result = < 1042>
  4: version = <            0.20>, best result = < 1089>
  5: version = <            0.50>, best result = <  940>
  6: version = <            1.00>, best result = < 1008>

Best statistic for configurable parameter <crossover method>:
  1: version = <            Half>, best result = < 1067>
  2: version = <        Moreless>, best result = <  878>
  3: version = <       ThreeAsym>, best result = < 1141>
  4: version = <            Four>, best result = < 1102>

Best statistic for configurable parameter <local optimizer>:
  1: version = <     AddLightest>, best result = < 1103>
  2: version = < RemSevAddSevEff>, best result = <  909>
Best combination of variants of parameters:
| Mutation:        <0.20>
| Crossover:       <ThreeAsym>
| Local optimizer: <AddLightest>
```

Рисунок 3.1 – максимальна цінність рішень при різних варіантах одних і тих самих оперпторів за призначенням у генетичному алгоритмі

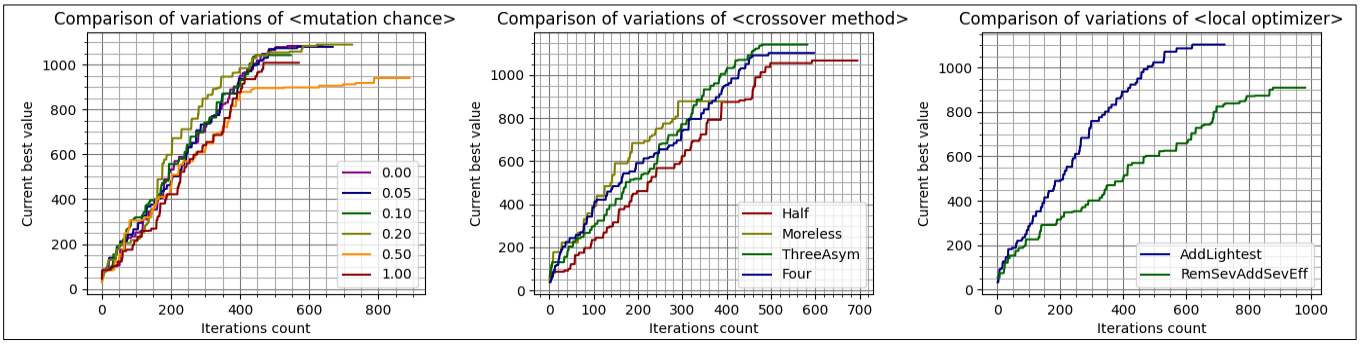



Рисунок 3.2 – графік роботи зростання максимальної цінності предмету в залежності від поточної ітерації

## 3.3 Тестування алгоритму

Нижче в таблиці наведено комбінацію параметрів, що є найбільш підходящими при вирішенні одного з наборів даних з множини всіх можливих наборів даних задачі з рюкзаком генетичним алгоритмом при даних конфігураціях. В таблиці 3.3.1 наведено кілька комбінацій варіацій параметрів алгоритму, які найбілш підійшли для даного набору значень задачі з рюказаком.

Таблиця 3.3.1 — комінації найбільш оптимальних варіацій параметрів при вирішення задачі

| Номер виміру | Варіант мутації | Варіант кросоверу | Варіант локального пошуку |
|---|---|---|---|
| 1 | 0.50 | Four | AddLightest |
| 2 | 0.00 | ThreeAsym | AddLightest |
| 3 | 0.05 | ThreeAsym | AddLightest |
| 4 | 0.50 | ThreeAsym | AddLightest |
| 5 | 0.10 | ThreeAsym | AddLightest |
| 6 | 0.10 | ThreeAsym | AddLightest |
| 7 | 0.05 | ThreeAsym | AddLightest |
| 8 | 0.10 | Four | AddLightest |
| 9 | 1.00 | Four | AddLightest |
| 10 | 1.00 | ThreeAsym | AddLightest |

По таблиці чітко видно, що варіанти мутації слабо впливають на роботу алгоритму та знаходження найкращого рішення, тому його значення знаходиться в широких межах (від 0% до 100%). Проте найбільш кращим варіантом просоверу є двохточковим та трьохточковим, бо їх кількість є

найбільшою в цій таблиці з найкращими підібраними комбінаціями варіантів параметрів.

Найбільш ефектичним методом локального пошуку є «AddLightest», також він знаходить напів-оптимальне рішення швидше, а ніж інші варіації локального пошуку, як «RemSevAddSevEff».

# ВИСНОВОК

В рамках даної лабораторної роботі я нарешті зміг зробити емпіричні виміри для різних операторів генетичного алогритму для вирішення такої важкої проблеми, як задачу з рюкзаком.

Під час аналізу різних варіантів параметрів алгортму генетичного алгоритму для задачі з рюкзаком було помічено та емпірично доведено, що мутації, на жаль, слабо впливають на найбільшу знайдену цінність особини, бо різні варіації в талблиці присутні в практично однакових співвідшеннях, тому мутації не є найбільш ефективним методом забезпечення на кінцевих ітераціях еволюції найкраще рішення.

Проте варіації оператора кросоверу справді впливають на остаточний результат виконаня алгоритму, бо співвідношення між різними операторами є значне. Найбільш ефективним оператором кросоверу є двоточкове, але трьохточкове теж є достатньо ефективним на деких даних за таблицею вище.

Найбільш ефективним та найбільш швидким оператором локального пошуку є «AddLightest» за графіками та таблицею найбільш оптимальних комбінацій варіантів параметрів для різних наборів даних. На жаль, оператор локального пошуку «RemSevAddSevEff» є не дуже ефективним та швидким для цієї задачі.

Під час лабораторної роботі я зміг поекспериментувати з різними варіантами параметрів генетичного алгоритму для задачі рюкзака для визначення найбільш оптимальних варіацій параметрів та їх комбінації.

## КРИТЕРІЇ ОЦІНЮВАННЯ

Критерії оцінювання у відсотках від максимального балу:

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