Київський національний університет імені Тараса Шевченка

Коновалов А.М.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ ДО ЛАБОРАТОРНИХ РОБІТ З ДИСЦИПЛІНИ «ЕВОЛЮЦІЙНІ ОБЧИСЛЕННЯ»

Київ – 2025

УДК 519.168:004.02

Коновалов А.М.

Методичні рекомендації до лабораторних робіт з дисципліни «Еволюційні обчислення». – К.: Київський національний університет імені Тараса Шевченка, 2025. – 40 с.

Рецензенти:

д-р. техн. наук, проф. С.Д. Погорілий,

канд. фіз.-мат. наук, доц. Б.В. Довгай

Методичні рекомендації складені згідно з робочою програмою дисципліни «Еволюційні обчислення», що викладається для студентів кафедри комп'ютерної інженерії факультету радіофізики, електроніки та комп’ютерних систем, які навчаються за освітньо-науковою програмою «Комп’ютерні системи та мережі» спеціальності «Комп’ютерна інженерія».

Затверджено вченою радою

факультету радіофізики, електроніки та комп’ютерних систем

(протокол від 28 квітня 2025 року № 10)

© Коновалов А.М., 2025

© Київський національний університет імені Тараса Шевченка, 2025

Зміст

Вступ

У методичному посібнику наведені описи лабораторних робіт з дисципліни «Еволюційні обчислення», яка викладається для студентів спеціальності «Комп'ютерна інженерія» освітнього рівня «магістр» факультету радіофізики, електроніки та комп'ютерних систем. Лабораторні роботи присвячені практичному застосуванню генетичних алгоритмів для вирішення класичних задач комбінаторної оптимізації [1-3] з використанням мови програмування Python і бібліотеки еволюційних обчислень **DEAP**.

Лабораторні роботи передбачають підготовку студента до лабораторного заняття, яка полягає у засвоєнні матеріалів розділу «Теоретичні відомості», а також виконання завдань самостійної роботи.

На лабораторних заняттях студент виконує завдання, які детально описані у розділі «Завдання до виконання на лабораторному занятті». Для виконання цих завдань на початку заняття студенту надаються додаткові матеріали в електронному вигляді, зокрема файли наборів даних різних варіантів.

Під час виконання лабораторних робіт використовуються мова програмування Python версії 3.11 або новішої, а також бібліотеки **DEAP**, **NumPy**, **Matplotlib**, **SciPy**.

Студент готує звіт з лабораторної роботи у вигляді Jupyter-записника, що створюється у хмарному середовищі інтерактивної розробки програм **Google Colaboratory** або інших Jupyter-подібних середовищ (**JupyterLab**, **VS Code**), встановлених на локальному комп'ютері.

Складання лабораторної роботи викладачеві передбачає демонстрацію особисто студентом роботи розробленої програми та відповіді на питання викладача.

Лабораторна робота 1

Асиметрична задача комівояжера зі змінною швидкістю

14 годин самостійної роботи студентів,

7 годин лабораторних занять

Мета роботи:

- ознайомитись з принципами використання програмних інструментів бібліотеки DEAP для пошуку розв'язків задач оптимізації за допомогою генетичних алгоритмів;
- розробити програму для розв'язку асиметричної задачі комівояжера за допомогою генетичного алгоритму;
- дослідити вплив параметрів генетичного алгоритму на результати його роботи.

Теоретичні відомості

Постановка задачі

Задача комівояже́ра (Travelling salesman problem, TSP) – одна з найвідоміших класичних задач, яка полягає у знаходженні оптимального маршруту комівояжера (бродячого торговця), який проходить через всі зазначені міста по одному разу *з* наступним *поверненням у початкове місто* [1, 2, 4]. Шлях, за яким комівояжер проходить через всі міста і повертається в початкове, також називається *туром*. Оптимальність маршруту визначається критерієм, що задається конкретним варіантом задачі (найкоротший, найдешевший, найшвидший тур тощо).

У задачі може задаватись матриця вартостей (відстаней, часів тощо) переміщення між n містами:

$$C = \{c_{ij}\}_{i,j=0}^{n-1} = \begin{pmatrix} c_{00} & \cdots & c_{n-1,0} \\ \vdots & \ddots & \vdots \\ c_{0,n-1} & \cdots & c_{n-1,n-1} \end{pmatrix}, \quad (1.1)$$

де c_{ij} – вартість переміщення між містом i та j. Діагональні елементи матриці c_{ii} у задачі не використовуються, адже передбачається однократне відвідування міст. Матриця вартості C може бути *симетричною* ($c_{ij} = c_{ji}$, тобто вартість переміщення між двома довільними містами i, j *не залежить* від напрямку переміщення) *або асиметричною* ($c_{ij} \neq c_{ji}$, тобто вартість переміщення у загальному випадку *залежить* від напрямку). Таким чином, для симетричної задачі сумарна вартість довільного маршруту не залежить від напрямку обходу фіксованої послідовності міст (прямий або зворотний), наприклад, маршрути $0 \to 3 \to 1 \to 2 \to 0$ і $0 \to 2 \to 1 \to 3 \to 0$ мають однакову вартість. Для асиметричної задачі вартості таких маршрутів у загальному випадку різні.

У ще більш загальному випадку вартість переміщення між двома містами може залежати від порядку відвідуваності міст. До таких випадків відноситься задача, у якій швидкість руху комівояжера залежить від кількості вже відвіданих міст (або кількості пройдених ділянок маршруту). Наприклад, комівояжер, який розвозить товар по містам, може збільшувати свою швидкість руху після кожного часткового розвантаження у місті. Або навпаки, комівояжер, який збирає товар (відправлення), може зменшувати свою швидкість після кожного часткового завантаження у місті. У такому випадку замість матриці вартостей у задачі задається алгоритм розрахунку вартості переміщення між містами з урахуванням кількості вже відвіданих міст. Звісно, що такий варіант задачі комівояжера є асиметричним, адже вартість маршруту залежить від його напрямку.

В цій лабораторній роботі розглядається задача з n містами, в якій комівояжер змінює свою швидкість v_k лінійно з кількістю відвіданих міст k (або номером ділянки шляху k) від заданого початкового значення v_0 до заданого кінцевого значення v_{final}, яка може бути більшим або меншим за v_0:

$$v_k = v_0 + \Delta v \cdot k, \qquad \Delta v = \frac{v_{final} - v_0}{n-1},$$
$$k = 0, 1, \dots, n-1, \qquad \Delta v_{final} \equiv v_{n-1}. \tag{1.2}$$

Сумарна вартість туру C_Σ визначається сумарним витраченим комівояжером часом T_Σ:

$$C_\Sigma = T_\Sigma = \sum_{k=0}^{n-1} t_k\,, \qquad t_k = \frac{d_k}{v_k}, \tag{1.3}$$

де t_k – час, витрачений комівояжером на ділянці шляху k, d_k – пройдена відстань на цій ділянці зі швидкістю v_k.

У залежності від варіанту задачі відстань d_{ij} між кожною парою міст i та j визначається або як довжина відрізку, що з'єднує цю пару міст із заданими координатами (x_i, y_i) та (x_j, y_j), або із заданої матриці відстаней між містами

$$D = \{d_{ij}\}_{i,j=0}^{n-1} = \begin{pmatrix} d_{00} & \cdots & d_{n-1,0} \\ \vdots & \ddots & \vdots \\ d_{0,n-1} & \cdots & d_{n-1,n-1} \end{pmatrix}, \tag{1.4}$$

яка може бути симетричною ($d_{ij} = d_{ji}$) або асиметричною ($d_{ij} \neq d_{ji}$). Для окремого випадку симетричної матриці відстаней D і $v_0 = v_{final}$ задача перетворюється у симетричну. Початковим містом для комівояжера вважається місто із індексом $i = 0$. Приклад можливого туру комівояжера наведений на рис. 1.1.

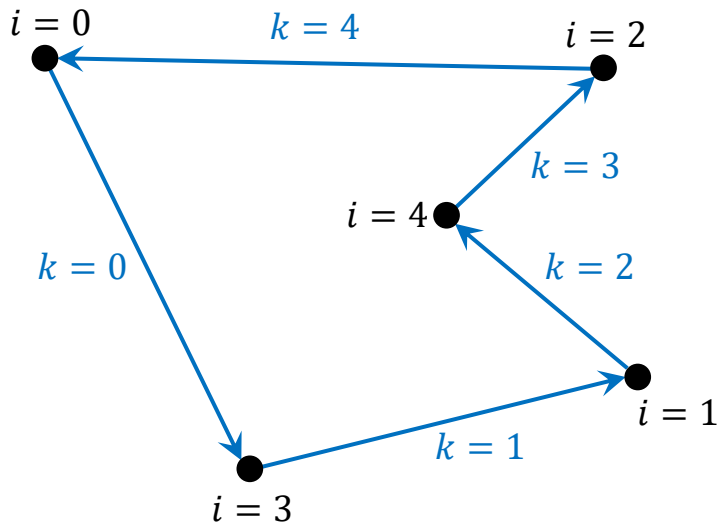


Рис. 1.1. Приклад туру комівояжера $0 \to 3 \to 1 \to 4 \to 2 \to 0$ для $n = 5$ (i – індекс міста, k – індекс ділянки туру).

Генетичні алгоритми

Методом повного перебору в принципі можна знайти точний розв'язок задачі комівояжера шляхом обчислення вартостей всіх можливих маршрутів та знаходження маршруту з найменшою сумарною вартістю. Однак, у такий спосіб навіть для невеликої кількості міст задача комівояжера не розв’язується практично, за прийнятний час, адже вона є NP-складною [1]. Для простого варіанту симетричної задачі з n містами, існує $(n-1)!\ /\ 2$ можливих варіантів турів. Наприклад, для 10 міст існує 181440 турів, для 20 міст – приблизно $\approx 6 \times 10^{16}$ турів, для 50 міст – вже $\approx 3 \times 10^{62}$.

Існують інші точні методи розв’язку задачі комівояжера, які намагаються зменшити обчислювальну складність задачі шляхом скорочення пошукового простору в різний спосіб. Однак часова складність таких алгоритмів залишається високою внаслідок явища «прокляття розмірності» (“curse of dimensionality”), яке полягає у суттєвому (за показниковим законом) ускладнені задачі при збільшені розмірності пошукового простору. Детальніше з точними методами

комбінаторної оптимізації, зокрема розв'язку задачі комівояжера, можна ознайомитись в [1].

Для знаходження задовільного (*субоптимального*) розв'язку задачі комівояжера за значно менший час, ніж час повного перебору всіх маршрутів, широко використовуються *генетичні алгоритми* [1-5]. Хоча вони не гарантують знаходження найбільш оптимального розв'язку, проте можуть бути застосовані для великої кількості міст.

Для використання генетичного алгоритму кожний тур комівояжера розглядається як *особина*, яка кодується *хромосомою* – масивом *генів*, тобто індексів міст фіксованої довжини n. Наприклад, тур $0 \to 3 \to 1 \to 4 \to 2 \to 0$ кодується хромосомою $H = (0, 3, 1, 4, 2)$. Значення генів приймають дискретні значення та є унікальними в межах хромосоми (кожне місто відвідується по одному разу), тому така хромосома відноситься до *негомологічних хромосом*. До того ж хромосоми відрізняються між собою лише перестановкою значень генів із заданої послідовності індексів від 0 до $n-1$. Така негомологічна хромосома називається *перестановочною* (permutation chromosome).

Набір хромосом (турів), з якою працює генетичний алгоритм на кожній ітерації називається *популяцією*. Задача полягає у знаходженні найпристосованішої (найкращої) хромосоми (найоптимальнішого туру) серед усіх можливих шляхом підбору (перестановки) генів.

Пристосованість хромосоми описується *функцією пристосованості* f або, як ще її часто називають, *цільовою функцією* або *фітнес-функцією* (*fitness function*), яка конструюється таким чином, щоб її мінімізація або максимізація шляхом підбору хромосом, що кодують тури комівояжера, приводила до мінімізації сумарної вартості туру C_Σ. У найпростішому випадку з врахуванням (1.3) фітнес-функцію можна визначити таким чином:

$$f = C_{\Sigma} = \sum_{k=0}^{n-1} \frac{d_k}{v_k} \to \min . \qquad (1.4)$$

На рис. 1.2 показана схема роботи узагальненого генетичного алгоритму, який можна поділити на такі етапи:

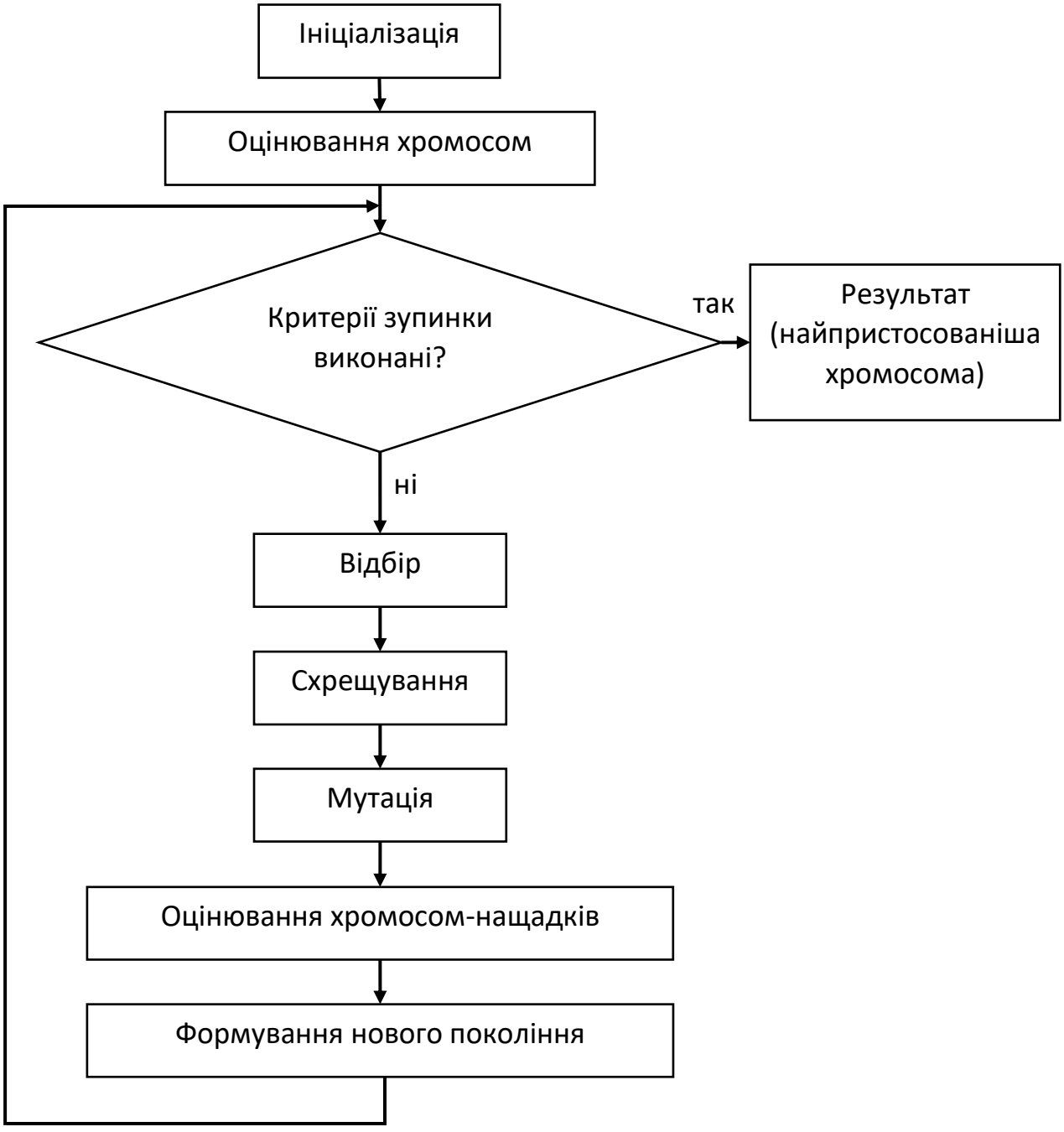


Рис. 1.2. Схема роботи узагальненого генетичного алгоритму.

1. **Створення початкової популяції** (ініціалізація) шляхом генерації заданої кількості хромосом з випадковими генами.

2. **Оцінювання хромосом** – обчислення фітнес-функції для всіх хромосом популяції.

3. Якщо виконуються **критерії зупинки** алгоритму, то відбувається завершення алгоритму і повернення найбільш пристосованої хромосоми як результату пошуку.

4. **Відбір** певної частки хромосом (*батьківських* хромосом), що допускаються до схрещування на наступному етапі, у так званий *батьківський пул*. Ця операція називається *оператором відбору, селекції*.

5. **Схрещування** (як правило по-парне) батьківських хромосом із заданою імовірністю для породження нових хромосом-нащадків, що успадковують від своїх батьків деякі риси – ділянки хромосом (послідовності генів). Ця операція називається *оператором схрещування, кросовером* (crossover), *кросинговером або оператором рекомбінації*. Метою операції є породження нових корисних комбінацій ген у хромосомах-нащадках.

6. **Мутація** хромосом-нащадків із заданою імовірністю, тобто випадкова зміна їх генів. Ця операція називається *оператором мутації*. Метою оператора мутації є диверсифікація розв'язків, тобто підвищення різноманітності пошуку і введення нового генетичного матеріалу в популяцію для забезпечення більш повного дослідження простору пошуку.

7. **Оцінювання хромосом-нащадків**, отриманих в результаті схрещування та мутації.

8. **Формування нового покоління** хромосом, яке може складатись з хромосом-нащадків і, можливо, найкращих батьківських хромосом. Ця операція називається *оператором редукції* і є одним і варіантів оператора відбору.

9. Перехід до пункту 3 алгоритму.

Серед критеріїв зупинки алгоритму можуть бути:

- знаходження оптимального (задовільного) розв'язку,
- досягнення максимально допустимої кількості поколінь (ітерацій) еволюції,
- досягнення максимальної кількості обчислень фітнес-функцій,
- вичерпання часу розрахунків,
- припинені поліпшення фітнес-функції найпристосованішої у популяції хромосоми,
- досягнення стану адаптації популяції.

Різні варіанти генетичних операторів кожного типу та критеріїв зупинки, а також інші деталі роботи генетичного алгоритму розглядаються на лекційних заняттях.

Бібліотека DEAP

DEAP (Distributed Evolutionary Algorithms in Python) – широко відома відкрита бібліотека еволюційних обчислень, що використовується для розробки програм мовою Python. Вихідний код бібліотеки представлений на веб-сервісі **GitHUB**. Бібліотека надає гнучкі можливості для самостійного створення різних типів елементів еволюційних алгоритмів: фітнес-функції, особини-хромосоми, популяції, еволюційні оператори, алгоритми тощо, а також використання вже створених у бібліотеці базових елементів.

Для встановлення бібліотеки у середовищі Google Colaboratory або іншому Jupyter-подібному середовищі слід виконати команду:

```
!pip install deap
```

Інші варіанти встановлення бібліотеки описані в онлайн-документації.

Приклад розв'язку задачі One Max problem

Використання бібліотеки **DEAP** для розв'язку задач за допомогою генетичних алгоритмів продемонструємо на тривіальній тестовій класичній задачі **One Max problem**. В задачі розглядається популяція бінарних хромосом фіксованої довжини (наприклад, 011000101), ініціалізованих випадковим чином. Фітнес-функція визначається як кількість одиничних значень всіх генів хромосоми (для наведеного прикладу хромосоми фітнес-функція повертає 4). Генетичному алгоритму ставиться задача знайти хромосому з максимальним значенням фітнес-функції, тобто знайти тривіальний розв'язок – хромосому, для якої всі гени мають одиничне значення.

Короткий приклад програми, що вирішує задачу **One Max problem**, наводиться у файлі deap/examples/ga/onemax_short.py репозиторію бібліотеки **DEAP**. Розглянемо код цієї програми з незначними змінами і доповненнями методичного характеру. Подальші приклади коду наполегливо рекомендується протестувати у записнику середовища **Google Colab** або іншого Jupyter-подібного середовища в інтерактивному режимі. Записник з модифікованим кодом, що розглядається, надається у файлі *onemax_demo.ipynb* матеріалів до лабораторної роботи.

Імпортуємо модулі бібліотек **random**, **numpy**, **matplotlib** і **DEAP**, що задіяні у подальшій частині коду:

```
import random
import numpy as np
from deap import creator, base, tools, algorithms
import matplotlib.pyplot as plt
```

За допомогою методу `creator.create` на основі абстрактного базового класу `base.fitness` створимо новий клас з назвою `"Fitness"`, що описує основні властивості фітнес-функції, що потребуватимете максимізації:

```
creator.create("Fitness", base.Fitness, weights=(1,))
```

Для створення класу фітнес-функції, що потребуватимете мінімізації, слід змінити ваговий коефіцієнт 1 на -1 в одноелементному кортежі, що визначає опцію `weights`. Надалі визначений клас `Fitness` є частиною модуля-контейнера `creator`, і до нього можна звертатись як `creator.Fitness`.

За допомогою того ж методу `creator.create` створимо клас `"Individual"` (особина-хромосома) на основі стандартного Python-класу `list`, зазначивши створений клас фітнес-функції `creator.Fitness`, що буде оцінювати особини-хромосоми:

```
creator.create("Individual", list, fitness=creator.Fitness)
```

Таким чином, особини-хромосоми можна буде задавати звичайним списком чисел, наприклад, `[0, 1, 1, 0, 0, 0, 1, 0, 1]`.

Для можливості роботи з різноманітними об’єктами генетичного алгоритму (особинами, популяцією, фітнес-функцією, генетичними операторами тощо) створимо спеціальний об’єкт-контейнер класу `base.Toolbox()`:

```
toolbox = base.Toolbox()
```

За допомогою методу `register` зареєструємо в контейнері `toolbox` перший об’єкт під назвою `"attr_bool"` – функцію, що генерує атрибути особини (тобто гени хромосоми) у вигляді випадкових цілих чисел 0 або 1, скориставшись функцією `random.randint` з двома значеннями обов’язкових параметрів цієї функції 0 і 1:

```
toolbox.register("attr_bool", random.randint, 0, 1)
```

Зареєструємо в контейнері toolbox другу функцію під назвою "individual":

```
toolbox.register("individual", tools.initRepeat, creator.Individual,
                 toolbox.attr_bool, n=100)
```

Ця функція буде генерувати бінарну хромосому за допомогою функції tools.initRepeat, яка застосовує n разів функцію генерації генів attr_bool і повертає об'єкт класу creator.Individual.

Зареєструємо в контейнері toolbox наступний об'єкт "population", що являтиме функцію генерації популяції – списку особин довільного розміру, який буде створений пізніше під час ініціалізації популяції:

```
toolbox.register("population", tools.initRepeat, list,
                 toolbox.individual)
```

Визначимо фітнес-функцію, яка сприймає бінарну хромосому individual у вигляді списку цілих чисел 0 та 1 і повертає кількість одиниць у хромосомі:

```
def evalOneMax(individual):
    return sum(individual),
```

Зверніть увагу, що функція має повертати результат у вигляді одноелементного кортежу, тобто у тому ж форматі, в якому задається опція weights при створені класу фітнес-функції creator.FitnessMax. Використання такого формату пов'язано з тим, що у загальному випадку фітнес-функція може бути багатоцільовою для можливості вирішення задач багатокритеріальної оптимізації.

Зареєструємо фітнес-функцію evalOneMax як об'єкт контейнера toolbox під назвою "evaluate":

```
toolbox.register("evaluate", evalOneMax)
```

Тепер фітнес-функція `toolbox.evaluate` може бути використана генетичним алгоритмом бібліотеки **DEAP** для оцінювання особин.

Залишилось зареєструвати генетичні оператори відбору (селекції), схрещування (кросовер) та мутації. Кожен оператор може бути визначений користувачем самостійно, як була визначена фітнес-функція. Але бібліотека **DEAP** у своєму модулі `tools` пропонує для використання вже готові функції, що реалізують генетичні оператори для різних типів хромосом. Описи доступних операторів наведені у розділі Operators документації. У нашому випадку для бінарної хромосоми скористаємось операторами двоточкового схрещування (функція `tools.cxTwoPoint`), щільнісної мутації (функція `tools.mutFlipBit`), а також турнірним відбором особин у батьківський пул (функція `tools.selTournament`):

```
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", tools.selTournament, tournsize=3)
```

Для функції щільнісної мутації задається параметр `indpb` – імовірність мутації (зміни на інше бінарне значення) кожного гена обраної хромосоми (не плутати з імовірністю обрання хромосоми для мутації). Для функції турнірного відбору також задається параметр розміру турніру `tournsize`.

Зверніть увагу, що хоча під час реєстрації об’єктів генетичного алгоритму в контейнері `toolbox` їх назви непринципові, однак об’єкти `evaluate`, `select`, `mate`, `mutate` використовуються іншими функціями бібліотеки DEAP, тому їх назви змінювати не слід.

Для можливості аналізу зміни статистичних характеристик популяції з кожною ітерацією (кожним поколінням) алгоритму створимо об’єкт-контейнер класу `tools.Statistics`:

```
stats = tools.Statistics(lambda ind: ind.fitness.values)
```

Єдиним параметром конструктору класу `tools.Statistics` є lambda-функція, яка буде застосовуватись до кожної особини популяції і повертати значення її фітнес-функції. Таким чином, статистичні характеристики будуть визначатись саме для множини значень фітнес-функції всіх особин популяції. Зареєструємо функції для розрахунку статистичних характеристик у контейнері `stats`, скориставшись функціями бібліотеки **numpy** для обчислення середнього значення `np.avg`, стандартного відхилення `np.std`, мінімального і максимального значень `np.min` і `np.max`:

```
stats.register("avg", np.mean)
stats.register("std", np.std)
stats.register("min", np.min)
stats.register("max", np.max)
```

Для збереження найпристосованіших особин, що знаходились генетичним алгоритмом протягом всієї роботи, створимо об’єкт класу `tools.HallOfFame` – так звана «зала слави». Він дозволяє алгоритму зберігати задану кількість найпристосованіших особин (параметр `maxsize` конструктору класу) у впорядкованому вигляді таким чином, щоб першим елементом (елементом з індексом 0) зали слави завжди була особина з найоптимальнішою фітнес-функцією:

```
hof = tools.HallOfFame(1)
```

Методи цього об’єкту будуть використовуватись генетичним алгоритмом після кожного оновлення популяції.

Створимо початкову популяцію розміром 300 особин за допомогою раніше зареєстрованого методу `toolbox.population`:

```
random.seed(64)
pop = toolbox.population(n=300)
```

Команда `random.seed` встановлює базу («зерно») генератора псевдовипадкових чисел бібліотеки `random`, яка використовується функціями бібліотеки DEAP для генерації випадкових чисел. Ця команда фіксує послідовність псевдовипадкових чисел, що генерується у подальшому, що дозволяє відтворювати всі результати обчислень.

Модуль `algorithms` надає готові функції, що реалізують різні варіанти еволюційних алгоритмів. Скористаємось функцією `algorithms.eaSimple`, яка реалізує простий алгоритм, розглянутий у підрозділі «Генетичні алгоритми» (див. також рис. 1.2):

```
pop, log = algorithms.eaSimple(
    pop, toolbox,cxpb=0.5, mutpb=0.2,
    ngen=40, stats=stats, halloffame=hof, verbose=True
)
```

Ця функція використовує генетичні оператори, зареєстровані в об'єкті `toolbox`, який передається на віх функції, нове покоління формується виключно з особин-нащадків, еволюція популяції фіксованого розміру виконується задану кількість поколінь `ngen`. Параметр `cxpb` задає імовірність застосування оператора схрещування до випадково обраної пари особин, `mutpb` – імовірність застосування оператора мутації до кожної особини (не плутати з параметром `indpb` конкретного оператора мутації `tools.mutFlipBit`). Опціональний параметр `verbose=True` вказує функції друкувати статистичну інформацію після кожної ітерації алгоритму у вигляді таблиці з колонками **gen** (номер покоління), **nevals** (кількість обчислень фітнес-функції на кожній ітерації) та іншими колонками, що визначаються зареєстрованими статистичними функціями в контейнері `stats`. Функція `eaSimple` повертає кортеж з двох об'єктів: `pop` – кінцева популяція у вигляді списку особин (об'єктів раніше створеного класу `creator.Individual`), `log` – журнал зі статистикою еволюції (об'єкт класу `tools.Logbook`).

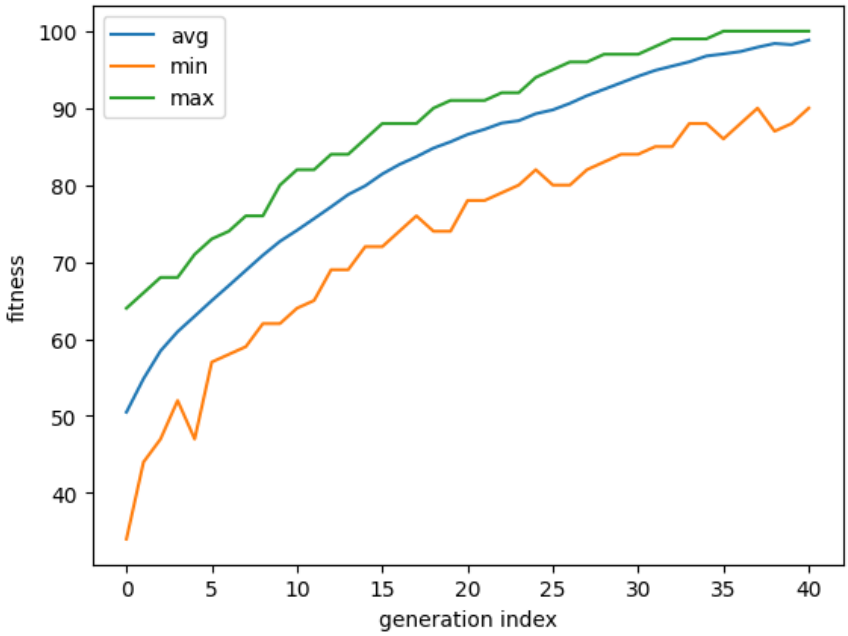
Повторний запуск клітинки з останньою командою продовжить роботу алгоритму ще на ngen=40 поколінь, сприймаючи останню популяцію pop як початкову. Для ініціалізації алгоритму слід виконати клітинку зі створенням об'єкту «зала слави» hof і початкової популяції pop.

Зверніть увагу, що після виконання функції eaSimple найпристосованіша особина кінцевої популяції pop може виявитись ненайкращою особиною в усіх попередніх популяціях, згенерованих алгоритмом, адже функція eaSimple формує на кожній ітерації нове покоління лише з особин-нащадків, в той час як найкращими розв'язками можуть виявитись батьківські особини. А отже у такому випадку найкращі розв'язки будуть втрачені. Отримати найпристосованішу особину за весь час роботи алгоритму можна, використовуючи перший елемент зали слави:

```
best_individual = hof[0]
print(best_individual)
```

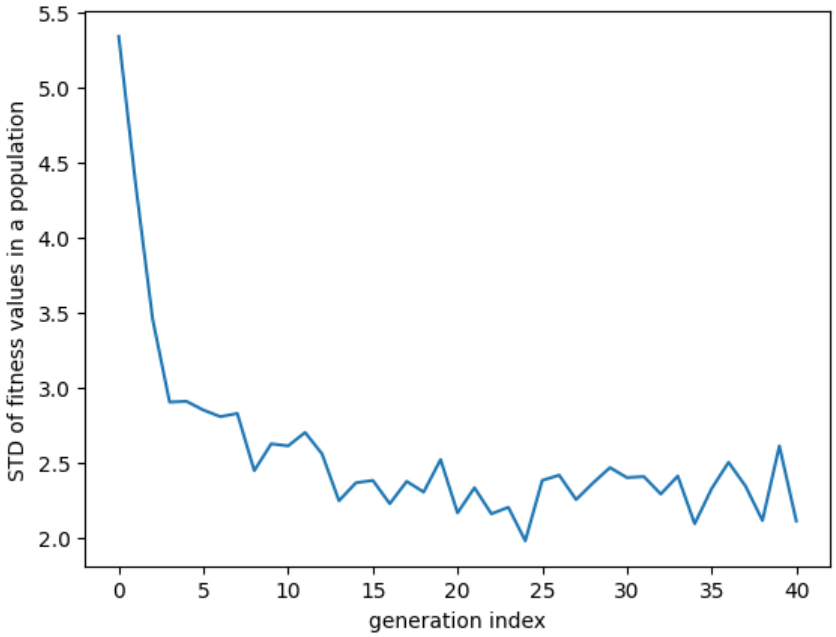
Журнал log дозволяє отримати залежності статистичних характеристик популяції, зареєстрованих раніше в об'єкті stats, від індексу покоління gen. Метод log.select повертає кортеж списків значень зазначених статистик на кожній ітерації. Отже для побудови історичних залежностей статистик на одному графіку можна скористатись командами:

```
plt.plot(*log.select("gen", "avg"), label="avg")
plt.plot(*log.select("gen", "min"), label="min")
plt.plot(*log.select("gen", "max"), label="max")
plt.xlabel("generation index")
plt.ylabel("fitness")
plt.legend()
plt.show()
```



Як видно з графіка, з кожним поколінням значення фітнес-функції особин у популяції зростає, досягаючи максимального значення 100, що свідчить про знаходження найоптимальнішого розв’язку задачі **One Max problem**. Водночас стандартне відхилення значень фітнес-функції особин у популяції в цілому спадає під час еволюції:

```
plt.plot(*log.select("gen", "std"))
plt.xlabel("generation index")
plt.ylabel("STD of fitness values in a population")
plt.show()
```



Сумарна кількість обчислень фітнес-функції протягом еволюції отримується за допомогою простого виразу:

```
sum(log.select('nevals'))
```

Завдання для самостійного ознайомлення з документацією бібліотеки DEAP

1. Розгляньте приклад реєстрації у контейнері `toolbox` функції створення перестановочної хромосоми-особини. Зверніть увагу на використання у цьому випадку функцій `random.sample` і `tools.initIterate` на відміну від випадку створення бінарної хромосоми.

2. Розгляньте функції операторів схрещування `tools.cxOrdered` (*двоточковий впорядковуючий кросовер* або *2OX-кросовер*) та `tools.cxPartialyMatched` (PMX-кросовер) [3, 4], які можуть бути використані для перестановочних хромосом (точкові кросовери не забезпечують унікальність генів у хромосомах-нащадках, тому не можуть використовуватись для негомологічних хромосом).

3. Розгляньте функцію оператору мутації обміну `tools.mutShuffleIndexes` [3, 4], який може бути використаний для перестановочних хромосом (щільнісна мутація не забезпечує унікальність генів у хромосомах-нащадках, тому не може використовуватись для негомологічних хромосом).

Завдання до виконання на лабораторному занятті

1. Використовуючи середовище **Google Colab** або інше Jupyter-подібне середовище створіть записник для виконання завдань лабораторної роботи в інтерактивному режимі. Встановіть бібліотеку **DEAP**, якщо вона ще не

встановлена у середовищі. У разі використання **Google Colab** завантажте файли **.dat*, що надаються до матеріалів лабораторної роботи, у поточну теку */content/* віртуальної машини **Google Colab** або підключіть Google-диск, якщо ці файли завчасно завантажені на ньому.

2. В окремій клітинці імпортуйте модулі, класи і функції бібліотек, що плануються до використання. Доповнюйте цю клітинку в ході виконання лабораторної роботи у разі необхідності додаткового імпортування. Використовуйте стандартні псевдоніми модулів.

3. Завантажте мапу 10 міст з файлу *map10.dat* текстового формату, що містить список умовних координат міст x та y у двох колонках, у двовимірний numpy-масив за допомогою функції `np.genfromtxt` або pandas-таблицю за допомогою функції `pd.read_csv`. Отриману структуру даних присвойте змінній `map`.

4. Обчисліть квадратну матрицю відстаней між містами D за допомогою функції `distance_matrix` (або самостійно побудованої функції) і присвойте її змінній D.

5. Для тестових значень початкової і кінцевої швидкостей $v_0 = 1$ і $v_{final} = 2$ обчисліть список або numpy-масив значень швидкостей на кожній ділянці туру комівояжера і присвойте його глобальній змінній v.

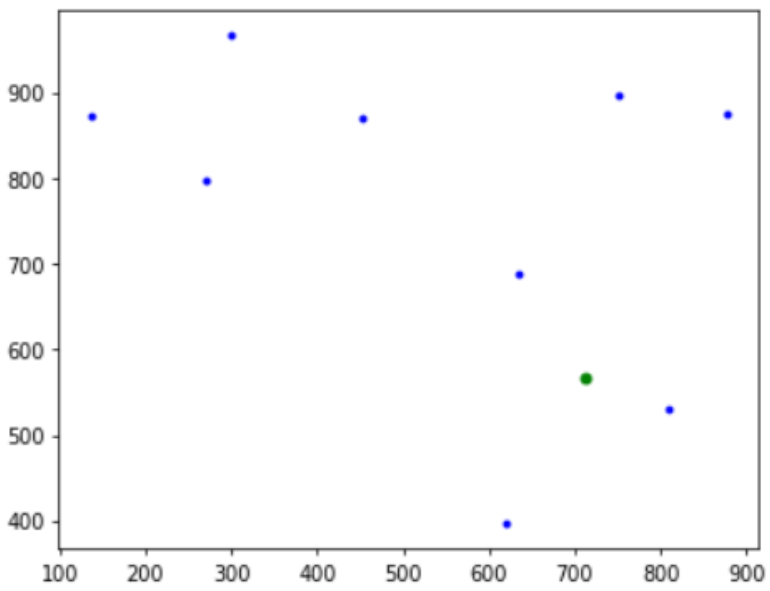
6. Визначте функцію `length_time(individual)`, яка для заданої особини-хромосоми у вигляді списку `individual`, що кодує шлях комівояжера послідовністю індексів відвідуваних міст, обчислює і повертає двоелементний кортеж зі значеннями довжини пройденого комівояжером шляху і витраченого часом. Наприклад:

```
length_time([6, 9, 0, 2, 4, 3, 5, 1, 8, 7])
(4029.0199719700768, 2829.089593787049)
```

Пам’ятайте, що за умовами задачі комівояжер завжди виїжджає з міста 0 (і повертається у нього), тому, наприклад, хромосоми `[6,9,0,2,4,3,5,1,8,7]`, `[0,2,4,3,5,1,8,7,6,9]`, `[2,4,3,5,1,8,7,6,9,0]`, які можуть утворюватись як на етапі ініціалізації популяції, так і в результаті застосування генетичних операторів схрещування і мутації, кодують один шлях комівояжера $0 \to 2 \to 4 \to 3 \to 5 \to 1 \to 8 \to 7 \to 6 \to 9 \to 0$, а отже функція `length_time` для таких хромосом має повертати однаковий результат. Для тестування функції різні хромосоми можна генерувати за допомогою виразу `random.sample(range(len(map)), len(map))`.

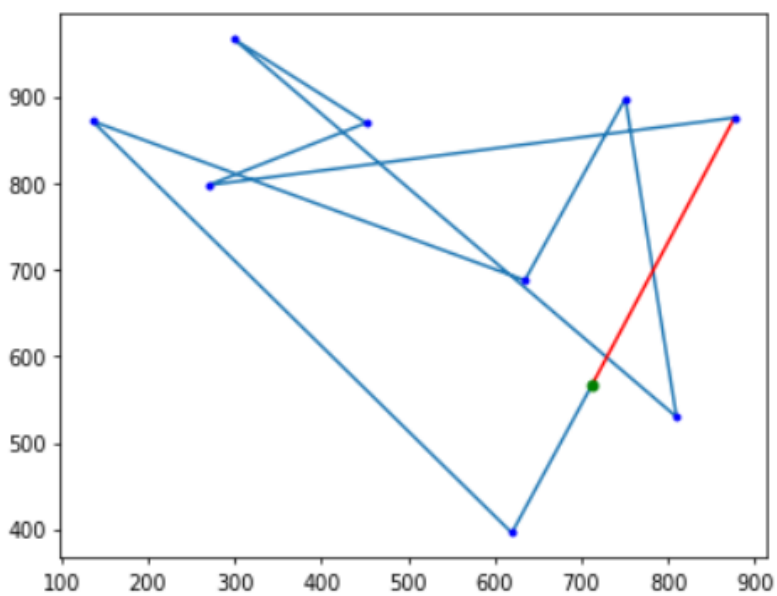
7. Визначте функцію `show_map_path(map, individual=None)`, яка відображає завантажену мапу міст у вигляді точок синього кольору з координатами, що задані структурою даних `map`. Початкове для комівояжера місто (місто з індексом 0) виділить зеленим кольором. Якщо задана особина-хромосома у вигляді списку `individual`, що кодує шлях комівояжера послідовністю індексів відвідуваних міст, то функція також має відобразити цей шлях замкнутою ломаною лінією. Щоб позначити напрям руху комівояжера виділіть червоним відрізком останню ділянку шляху, яким комівояжер повертається у початкове місто (при $v_0 = v_{final}$ напрям руху неважливий, задача стає симетричною, тому виділення останньої ділянки шляху непотрібне). Також функція `show_map_path` має обрахувати за допомогою функції `length_time` і вивести на екран довжину пройденого комівояжером шляху і витрачений час. Наприклад, для мапи міст з файлу *map10.dat* робота функції `show_map_path` з незаданою хромосомою має виглядати таким чином:

```
show_map_path(map)
```



Для заданої хромосоми з прикладу попереднього пункту робота функції має виглядати таким чином:

```
show_map_path(map, [6, 9, 0, 2, 4, 3, 5, 1, 8, 7])
```




```
v_0 = 1, v_final = 2
Path length: 4029.0199719700768
Path time: 2829.089593787049
```

8. Визначте фітнес-функцію `eval_time(individual)` відповідно до формули (1.4) яка має повертати одноелементний кортеж зі значенням витраченого комівояжером часу, що обчислюється за допомогою функції `length_time`.

9. В окремій клітинці визначте параметри генетичного алгоритму тестовими (неоптимальними) значеннями:

- `population_size` – розмір популяції;
- `tournsize` – розмір турніру оператора турнірного відбору;
- `crossover` – ім'я функції оператора схрещування модуля `tools` (рядок `"cxOrdered"` або `"cxPartialyMatched"`);
- `cxpb`, `mutpb`, `indpb` – параметри операторів схрещування і мутації, що розглядались вище;
- `ngen` – кількість поколінь еволюції, яка визначає зупинку алгоритму.

В середовищі **Google Colab** зробіть це, використовуючи зручний інструмент «Форми»: в контекстному меню вхідної клітинки з кодом оберіть пункт «Додати форму», надалі для створення поля форми і відповідної змінної використати пункт контекстного меню «Форма / Додати поле форми»:

10. Використайте програмний код файлу *onemax_demo.ipynb* як основу для програмування генетичного алгоритму, що вирішує задачу комівояжера із

заданими у попередньому пункті параметрами. Врахуйте специфіку цієї задачі: фітнес-функція `eval_time` має мінімізуватись, тип хромосом – перестановочний, оператор схрещування – двоточковий впорядковуючий кросовер `tools.cxOrdered` або PMX-кросовер `tools.cxPartialyMatched` у залежності від заданого параметра `crossover`, оператор мутації – `tools.mutShuffleIndexes` (мутація обміну). Побудуйте історичні залежності статистичних характеристик популяцій. Після налагодження роботи програми опцію `verbose` функції `algorithms.eaSimple` встановіть у значення `False`.

11. Побудуйте графічно знайдений алгоритмом найшвидший маршрут комівояжера за допомогою функції `show_map_path`.

12. Після перевірки коректності роботи програми підготуйте записник для зручності проведення досліджень впливу параметрів генетичного алгоритму на його роботу та використання програми для оптимізації маршруту комівояжера для довільно заданої мапи міст. Для цього об’єднайте вирази, що завжди потребують спільного послідовного обчислення, в окремі комірки. Записник має містити 5 комірок з такими групами команд (для зручності структура записника надається у файлі *Lab1-структура.ipynb* матеріалів до лабораторної роботи):

- команди, що потребуватимуть однократного обчислення лише після створення нового середовища виконання (нової віртуальної машини), але не після його перезапуску: встановлення бібліотек, підключення Google-диску;
- команди, що потребуватимуть однократного обчислення після запуску або перезапуску середовища виконання: підключення бібліотек, ініціалізація глобальних змінних (за потреби), визначення всіх функцій користувача командою `def`, створення нових класів `Fitness` та `Individual` в модулі `creator`, створення об’єкту `stats` класу `tools.Statistics` і реєстрація в ньому чотирьох статистичних функцій;

- завантаження і візуалізації мапи міст, обчислення матриці відстаней між містами D;

- визначення параметрів задачі (змінні `v_0`, `v_final`) і генетичного алгоритму, обчислення списку значень швидкостей на кожній ділянці туру комівояжера (змінна `v`), створення об'єкту `toolbox` класу `base.Toolbox` і реєстрація у ньому всіх необхідних функцій;

- створення об'єкту «зала слави» `hof`, початкової популяції з використанням команди встановлення бази генератора псевдовипадкових чисел `random.seed` для відтворюваності подальших обчислень, запуск генетичного алгоритму, візуалізація знайденого алгоритмом найшвидшого маршруту комівояжера за допомогою функції `show_map_path`, побудова історичних залежностей чотирьох статистик популяції у кожному поколінні.

Така структура записника зокрема дозволяє у разі зміни параметрів у певній клітинці обчислювати лише останню частину записника за допомогою комбінації клавіш **Ctrl-F10** (обчислення поточної клітинки й інших клітинок під нею) без необхідності переобчислення всього записника.

13. Використовуючи створений записник вищенаведеної структури дослідіть вплив параметрів генетичного алгоритму на його роботу. Підберіть «оптимальні» параметри генетичного алгоритму `tournsize`, `crossover`, `cxpb`, `mutpb`, `indpb` таким чином, щоб при встановленні функцією `random.seed` бази генератора псевдовипадкових чисел у цілі значення від 1 до 5 ***для всіх 5 варіантів*** початкових популяцій розміру не більше 200 осіб генетичний алгоритм ***одночасно*** знаходив оптимальний (субоптимальний) тур комівояжера з тривалістю шляху

- рівною 1556.78 протягом 200 поколінь для мапи міст *map10.dat* і параметрів швидкостей $v_0 = 1, v_{final} = 2$;

- не більше 3800 протягом 700 поколінь для мапи міст *map50.dat* і параметрів швидкостей $v_0 = 1, v_{final} = 3$.

Для демонстрації роботи генетичного алгоритму з мапою *map50.dat* можна зберегти копії відповідних клітинок наприкінці записника.

Лабораторна робота 2

Задача лінійного розкрою матеріалу

14 годин самостійної роботи студентів,

7 годин лабораторних занять

Мета роботи:

- розробити програму для розв'язку задачі лінійного розкрою матеріалу за допомогою генетичного алгоритму;
- дослідити вплив параметрів генетичного алгоритму на результати його роботи.

Теоретичні відомості

Постановка задачі

Задача лінійного (одновимірного) розкрою матеріалу (one dimensional cutting-stock problem, 1D CSP) є цікавою з огляду на її широке використання в різних галузях промисловості. Вона подібна до іншої відомої класичної задачі одновимірної упаковки в контейнери (one dimensional bin packing problem) [1-2], але має свої особливості.

Задача лінійного розкрою формулюється таким чином [2]. Матеріал надходить у вигляді *заготівок* (смуг, стрижнів) довжини L, з яких потрібно виготовити N деталей різної довжини $l_i < L,\ i = \overline{0, N-1}$. У задачі ставиться за мету знайти оптимальний план розкрою з мінімальною кількістю витрачених заготівок.

Ця задача є NP-складною, що не дозволяє її вирішувати точно за прийнятний час. Тому прийнятний (субоптимальний) розв'язок цієї задачі часто шукається за допомогою генетичних алгоритмів.

Кодування розв'язків задачі за допомогою хромосом

Одним із можливих способів кодування розв'язків задачі полягає в тому, що хромосома, що кодує план розкрою, представляється на основі упорядкування деталей в заготівках [2]. У цьому разі хромосома містить послідовність номерів деталей i, яка визначає порядок їх розташування в заготівках за правилом: перша деталь – деталь з номером, вказаним у гені хромосоми з індексом 0, – розташовується у першій заготівки; кожна наступна деталь з номером, вказаним у наступному гені хромосоми, розташовується у поточній заготівки, якщо це можливо (сума довжин деталей в кожній заготівці має не перевищувати довжину заготівки L), в іншому випадку – у новій заготівці. Наприклад, для довжини заготівок $L = 100$ і довжин деталей [39, 25, 34, 34, 32, 62] хромосома [4, 0, 3, 5, 2, 1] буде кодувати план розкрою, показаний на рисунку 2.1. Таким чином, для

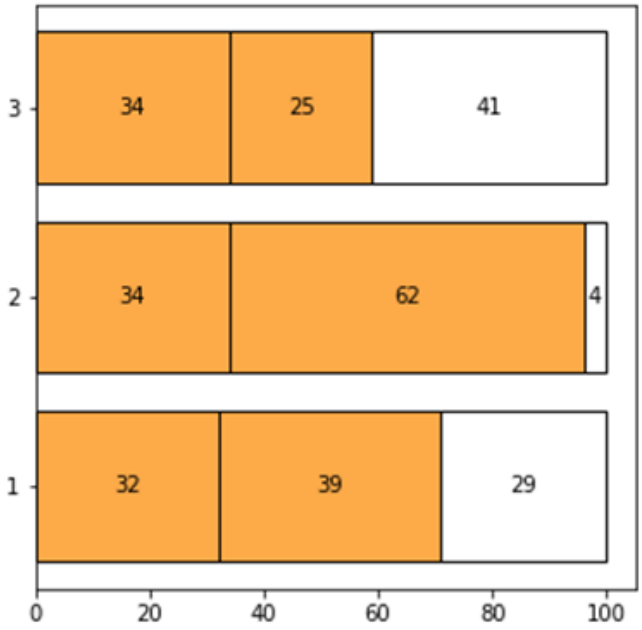


Рис. 2.1. Приклад плану розкрою 6 деталей з довжинами [39, 25, 34, 34, 32, 62], що кодує хромосома [4, 0, 3, 5, 2, 1] для заготівок довжиною $L = 100$.

вказаного способу кодування плану розкрою використовується той самий тип хромосоми, який вже використовувався в лабораторній роботі 1 для кодування шляху комівояжера, – перестановочна хромосома.

Фітнес-функція

У задачі, що розглядається, мінімізації підлягає кількість задіяних у плані розкрою заготівок M, тому в найпростіших випадках фітнес-функцію можна побудувати таким чином, щоб її слід було мінімізувати:

$$f = M \to \min, \tag{2.1}$$

або максимізувати:

$$f = \frac{1}{M} \to \max. \tag{2.2}$$

Однак такі варіанти фітнес-функції в багатьох випадках будуть сильно виродженими внаслідок існування великої кількості *різних* планів розкрою із однаковою кількістю задіяних заготівок M. Експериментально показано, що кращим порівняно з виразами (2.1) – (2.2) є такий вигляд фітнес-функції, що враховує заповненість заготівок [2]:

$$f = \frac{\sum_{k=0}^{M-1} p_k^2}{M} \to \max, \tag{2.3}$$

де p_k – ступінь заповнення k-тої заготівки, тобто відношення сумарної довжини деталей, що умістились у k-ту заготівку, до довжини заготівки L. Використання виразу (2.3) замість (2.1) – (2.2) суттєво зменшує виродженість фітнес-функції, хоч і не усуває її повністю.

Завдання для самостійної роботи

1. Реалізуйте мовою Python функцію `get_plan(individual)`, яка декодує вищеописаним способом вхідну особину-хромосому `individual` (список довжиною N) у план розкрою і повертає його у вигляді списку списків довжин деталей, розміщених у кожній використаній заготівці. У своїй роботі функція має використовувати дві глобальні змінні: L (довжина заготівок) і `item_lengths` (список або numpy-масив довжин всіх деталей). Наприклад:

```
L = 100
item_lengths = [39, 25, 34, 34, 32, 62]
individual = [3, 5, 0, 1, 4, 2]
plan = get_plan(individual)
plan
```

```
[[34, 62], [39, 25, 32], [34]]
```

У цьому прикладі план розкрою, що повертає функція `get_plan`, показує, що у першій заготівці розміщені 2 деталі з довжинами 34 і 62 (їх індекси у списку довжин деталей – 3 і 5 відповідно), у другій заготівці розміщені 3 деталі з довжинами 39, 25 і 32 (їх індекси– 0, 1 і 4 відповідно), у третій заготівці розміщена 1 деталь довжиною 34 (її індекс у списку довжин деталей – 2).

2. Реалізуйте мовою Python функцію `show_plan(plan)`, яка візуалізує план розкрою, що заданий вхідним параметром `plan`, у зручному текстовому ***або*** графічному вигляді і повертає кількість використаних заготівок у плані. Наприклад, робота текстової версії функції може виглядати таким чином:

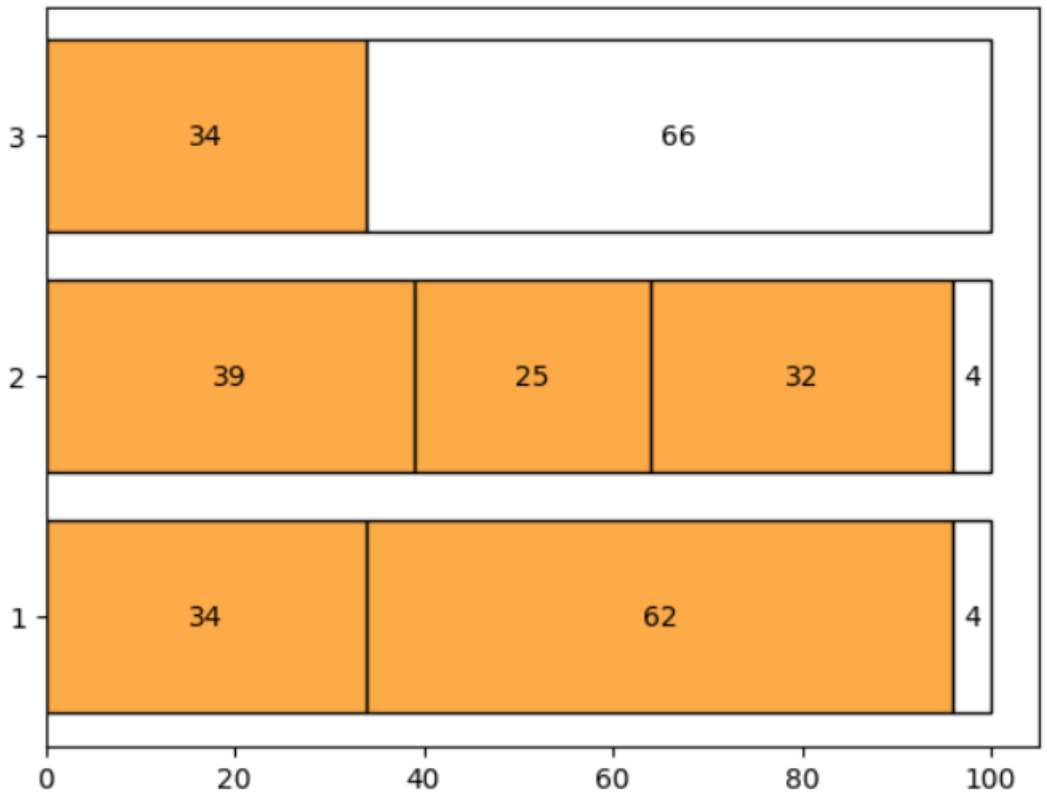
```
used_billets_count = show_plan(plan) # виводить план у текстовому вигляді
used_billets_count # кількість використаних заготівок
```

```
1: 34 + 62 | 4
2: 39 + 25 + 32 | 4
3: 34 | 66
3
```

У першій колонці текстової версії плану розкрою (до знаку **:**) вказуються порядкові номери використаних заготівок, а в останній (після знаку **|**) – залишок матеріалу заготівки. Не слід виводити кінцеві незначущі цифри 0 для цілочислених значень довжин деталей і залишків матеріалів заготівок. Робота графічної версії функції `show_plan` може виглядати таким чином:

```
used_billets_count = show_plan(plan) # виводить план у графічному вигляді
used_billets_count # кількість використаних заготівок
```

```
3
```



По вертикальній осі графічної версії плану розкрою відкладаються порядкові номери використаних заготівок, а по горизонтальній осі – довжини деталей і залишків матеріалів.

Завдання до виконання на лабораторному занятті

1. Використовуючи середовище **Google Colab** або інше Jupyter-подібне середовище створіть записник для виконання завдань лабораторної роботи в

інтерактивному режимі. Встановіть бібліотеку **DEAP**, якщо вона ще не встановлена у середовищі. У разі використання **Google Colab** завантажте файли **.dat*, що надаються до матеріалів лабораторної роботи, у поточну теку */content/* віртуальної машини **Google Colab** або підключіть Google-диск, якщо ці файли завчасно завантажені на ньому.

2. В окремій клітинці імпортуйте модулі, класи і функції бібліотек, що плануються до використання. Доповнюйте цю клітинку в ході виконання лабораторної роботи у разі необхідності додаткового імпортування. Використовуйте стандартні псевдоніми модулів.

3. Завантажте список довжин 21 деталей з файлу *21-773.dat* текстового формату в одновимірний numpy-масив за допомогою функції `np.genfromtxt`. Отриману структуру даних присвойте змінній `item_lengths`.

4. Оцініть спрощено мінімально можливу кількість використаних заготівок, виходячи із загальної довжини всіх завантажених деталей і фіксованої довжини заготівок L, яку у цій лабораторній роботі слід покласти рівною 100. Ця мінімальна кількість буде вважатись для генетичного алгоритму цільовою (хоча і необов'язково досяжною в принципі для довільного набору деталей):

$$M_{target} = \left\lceil \frac{\sum_{j=0}^{N-1} l_j}{L} \right\rceil.$$

Отримане цілочисельне значення присвойте змінній `M_target` і виведіть у вихідну комірку.

5. Використовуючи раніше розроблені функції `get_plan` і `show_plan`, візуалізуйте тривіальний план розкрою заготівок із послідовним розташуванням у них деталей у порядку, зазначеному в списку `item_lengths`.

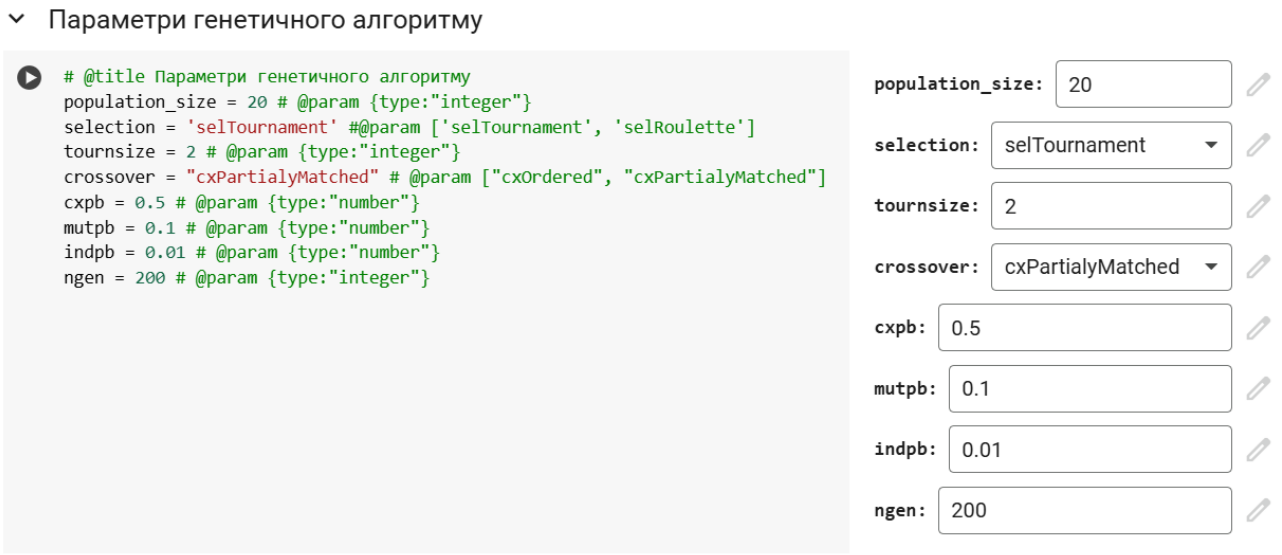
6. Визначте фітнес-функцію `evalFitness(individual)` відповідно до формули (2.3), яка має повертати одноелементний кортеж з обчисленим значенням f, наприклад:

```
individual = range(len(item_lengths))
evalFitness(individual)
```

```
(0.62675,)
```

7. Аналогічно завданню 9 попередньої лабораторної роботи в окремій клітинці визначте параметри генетичного алгоритму тестовими (неоптимальними) значеннями, однак у цій роботі додайте можливість обрання оператора відбору у батьківський пул (при максимізації фітнес-функції, що повертає лише позитивні значення, можна використовувати відбір методом рулетка):

- `population_size` – розмір популяції;
- `selection` – ім'я функції оператора відбору модуля `tools` (рядок `"selTournament"` або `"selRoulette"`);
- `tournsize` – розмір турніру оператора турнірного відбору (змінна використовується лише при виборі оператора турнірного відбору);
- `crossover` – ім'я функції оператора схрещування модуля `tools` (рядок `"cxOrdered"` або `"cxPartialyMatched"`);
- `cxpb`, `mutpb`, `indpb` – параметри операторів схрещування і мутації, що розглядались вище;
- `ngen` – кількість поколінь еволюції, яка визначає зупинку алгоритму.



8. Використайте програмний код, створений при виконанні завдання 10 попередньої лабораторної роботи, для програмування генетичного алгоритму, що вирішує задачу лінійного розкрою із заданими у попередньому пункті параметрами. Врахуйте специфіку цієї задачі, зокрема те, що фітнес-функція `evalFitness` має максимізуватись.

9. За допомогою функцій `get_plan` і `show_plan` візуалізуйте знайдений алгоритмом план розкрою і виведіть його елементарні характеристики (внаслідок виродження фітнес-функції її значення може варіюватись навіть для плану розкрою з однаковою кількістю використаних заготівок):

```
Пройдено епох: 200
Використано заготівок: 8
Значення фітнес-функції: 0.9382375
```

Проведіть декілька перезапусків алгоритму для досягнення оптимального плану розкрою з цільовою кількістю заготівок (за необхідності змінюючи базу генератора псевдовипадкових чисел, якщо вона раніше встановлювалась).

10. Як можна було помітити під час виконання попереднього завдання, генетичний алгоритм може досягати цільової кількості використаних заготівок `M_target` за меншу кількість поколінь, ніж задане значення `ngen`. Враховуючи, що значення `M_target` є відомим до запуску алгоритму, можна задати критерій

зупинки алгоритму таким чином, щоб він зупинявся при досягнені значення M_target. Для цього модифікуйте функцію eaSimple:

1) скопіюйте її код із модуля deap.algorithms в окрему комірку свого записника, вилучіть рядки документації задля скорочення коду і змініть назву функції на eaSimple_mod;
2) знайдіть основний цикл функції, у тілі якого на кожній ітерації генерується покоління особин;
3) наприкінці тіла циклу додайте код, що завершує цикл еволюції при досягнені цільової кількості використаних заготівок.

Отримана функція eaSimple_mod використовує у своїй роботі функцію varAnd із модуля deap.algorithms, тому імпортуйте її в окремій комірці. У раніше створеній комірці з кодом, що запускає генетичний алгоритм, замініть функцію eaSimple на eaSimple_mod. Перевірте коректність роботи модифікованої функції eaSimple_mod.

11. На основі коду, розробленого у попередніх завданнях, визначте функцію runGA(seed=None), яка створює об'єкт «зала слави» hof, встановлює базу генератора псевдовипадкових чисел у значення вхідного параметру seed для відтворюваності подальших обчислень, створює початкову популяцію, запускає генетичний алгоритм з використанням функції eaSimple_mod, будує історичні залежності чотирьох статистик популяції у кожному поколінні, візуалізує знайдений алгоритмом найкращий план розкрою за допомогою функції show_plan і виводить його характеристики. Перевірте роботу визначеної функції при різних значеннях seed.

12. Після перевірки коректності роботи всього програмного коду підготуйте записник для зручності проведення досліджень впливу параметрів генетичного алгоритму на його роботу та використання програми для оптимізації маршруту комівояжера для довільно заданої мапи міст. Для цього об'єднайте вирази, що

завжди потребують спільного послідовного обчислення, в окремі комірки. Записник має містити 5 комірок з такими групами команд (для зручності структура записника надається у файлі *Lab2-структура.ipynb* матеріалів до лабораторної роботи):

1) команди, що потребуватимуть однократного обчислення лише після створення нового середовища виконання (нової віртуальної машини), але не після його перезапуску: встановлення бібліотек, підключення Google-диску;
2) команди, що потребуватимуть однократного обчислення після запуску або перезапуску середовища виконання: підключення бібліотек, ініціалізація глобальних змінних (за потреби), визначення всіх функцій користувача командою def, створення нових класів Fitness та Individual в модулі creator, створення об'єкту stats класу tools.Statistics і реєстрація в ньому чотирьох статистичних функцій;
3) завантаження списку довжин деталей із заданого файлу і візуалізація тривіального плану розкрою заготівок із послідовним розташуванням у них деталей у порядку, зазначеному у файлі;
4) визначення параметрів генетичного алгоритму, створення об'єкту `toolbox` класу `base.Toolbox` і реєстрація у ньому всіх необхідних функцій;
5) п'ятикратний запуск генетичного алгоритму за допомогою циклу і функції `runGA(seed)` при 5 послідовних цілочисельних значеннях `seed` від 1 до 5.

13. Використовуючи створений записник вищенаведеної структури дослідіть вплив параметрів генетичного алгоритму на його роботу. Для списку довжин деталей з файлу *50-1471.dat* підберіть значення параметрів генетичного алгоритму `selection`, `tournsize`, `crossover`, `cxpb`, `mutpb`, `indpb` таким чином, щоб для 5 значень `seed`, вказаних у попередньому завданні, алгоритм знаходив не менше 4 разів оптимальний план розкрою з цільовою кількістю

використаних заготівок `M_target` з розміром популяції 60 особин протягом не більше 600 поколінь.

14. Для підібраних у попередньому завданні значень параметрів запустіть генетичний алгоритм для 5 значень `seed` з тими самими значеннями параметрів розміру популяції 60 особин і максимальної кількості поколінь 600 для списку довжин деталей з файлів *21-773.dat* і *63-1472.dat*. Скільки разів і за яку кількість поколінь алгоритму вдалось знайти оптимальні плани розкрою з цільовою кількістю використаних заготівок? Зробіть висновки з отриманих результатів.

Для демонстрації роботи генетичного алгоритму для 3 списків деталей при 5 значеннях `seed` збережіть копії відповідних клітинок (разом з вихідними графіками) наприкінці записника.

Список літератури

1. Погорілий С. Д. Застосування генетичних алгоритмів у комп'ютерних системах: монографія / С. Д. Погорілий, Р. В. Білоус, І. В. Білоконь; за ред. проф. С. Д. Погорілого. — К.: Видавничо-поліграфічний центр «Київський університет», 2014. — 319 с.
2. Глибовець М.М., Гулаєва Н.М. Еволюційні алгоритми: підручник. – Київ: НаУКМА, 2013. – 828 с.
3. Субботін С.О., Олійник А.О., Олійник О.О. Неітеративні, еволюційні та мульти-агентні методи синтезу нечіткологічних і нейромережних моделей: Монографія / Під заг. ред. С. О. Субботіна. – Запоріжжя: ЗНТУ, 2009. – 375 с.
4. Keresztury B. Genetic algorithms and the Traveling Salesman Problem. BSc thesis, Supervisor: Tamás Király. ELTE, Department of Operations Research, Budapest, 2017. – 50 pages.
5. Katoch S., Chauhan S.S., Kumar V. A review on genetic algorithm: past, present, and future. Multimed Tools Appl 80, 8091–8126 (2021). https://doi.org/10.1007/s11042-020-10139-6