КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теоретичної кібернетики

Кваліфікаційна робота

на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки
на тему:

PYTHON-БІБЛІОТЕКА ДЛЯ ВИРІШЕННЯ ЗАДАЧ З ВИКОРИСТАННЯМ ГЕНЕТИЧНОГО АЛГОРИТМУ

Виконала: студентка 4-го курсу
Марія КРИВЕЦЬ

(підпис)

Науковий керівник:
професор кафедри теоретичної кібернетики,
доктор фіз.-мат. наук, професор
Анатолій ПАШКО

(підпис)

Засвідчую, що в цій роботі
немає запозичень з праць інших авторів
без відповідних посилань.
Студент

(підпис)

Роботу розглянуто й допущено до захисту
на засіданні кафедри теоретичної кібернетики
«01» червня 2022 р.,
протокол № 11
Завідувач кафедри, доктор фіз.-мат. наук, професор
Юрій КРАК

(підпис)

КИЇВ-2022

РЕФЕРАТ

Обсяг роботи 54 сторінки, 12 ілюстрацій, 11 таблиць, 6 джерел посилань.

Ключові слова: ГЕНЕТИЧНИЙ АЛГОРИТМ, ЕВОЛЮЦІНІ МЕТОДИ, ЕКСТРЕМУМ ФУНКЦІЇ, ЗАДАЧА КОМІВОЯЖЕРА, ОПТИМІЗАЦІЯ, ПОШУК МІНІМУМУ.

Об’єктом розробки програмного забезпечення є створення бібліотеки для розв’язання задач оптимізації з використанням генетичного алгоритму та різних його модифікацій.

Метою кваліфікаційної роботи є ознайомлення з різними варіаціями генетичного алгоритму, методами кожного з етапів його роботи, виявлення їх переваг та недоліків. Визначення найбільш оптимальних модифікацій алгоритму для різних класів оптимізаційних задач. Застосування отриманих знань при написанні зручної у використанні та гнучкої для доповнень python-бібліотеки.

Інструментом створення є інтегроване середовище розробки програмного забезпечення PyCharm. Мова програмування – Python, допоміжні бібліотеки – numpy, math, matplotlib.

Результат роботи: бібліотека з усіма необхідними функціями для вирішення обраних типів задач. Досліджені найпопулярніші на сьогоднішній модифікації, обрані найефективніші методи для різних етапів алгоритму. Створена структура, зручна для швидкого вдосконалення існуючих методів, а також для додавання нових видів задач. Розглянуті на конкретних прикладах відмінності між генетичними та іншими пошуковими алгоритмами.

ЗМІСТ

ВСТУП

Актуальність роботи та підстави для її виконання. Задачі оптимізації в тому чи іншому вигляді постають перед людством протягом всього часу його існування. Наприклад, мінімізувати відстань при обранні маршруту зазвичай означало пришвидшення процесу перевезення товарів, зменшення витрат ресурсів та ризиків.

Загалом, оптимізація полягає в тому, щоб обрати найбільш вигідний варіант серед множини усіх можливих варіантів. Таким чином можна помітити, що цей процес присутній у багатьох сферах нашого повсякденного життя, при цьому іноді ми навіть не усвідомлюємо, що займаємось оптимізацією певного рішення – це здається абсолютно природнім максимізувати для себе вигоду (прибуток, наприклад) та мінімізувати витрату ресурсів.

Так як на даний момент у нас є всі можливості для автоматизації підбору найбільш оптимального вирішення майже будь-якої проблеми, питання постає у тому, щоб обрати найкращий алгоритм. Одним із них є генетичний алгоритм, який буде детально досліджено в даній роботі.

Оцінка сучасного стану об’єкта дослідження. Генетичний алгоритм застосовується у різноманітних задачах: обранні ознак у машинному навчанні, оптимізації молекулярної структури в хімії, задачі комівояжера, розміщенні файлів у розподілених системах, створенні розкладів (які будуть однаково зручними для усіх учасників навчального або робочого процесу), обробці зображень, тощо [1]. Для цих же завдань можна використовувати як аналог нейронні мережі.

Генетичний алгоритм зазвичай розглядають як спосіб вирішення певної задачі, але набагато зручнішим буде створити з нього універсальний інструмент, який можна буде застосовувати для різних класів задач. При

вдосконаленні вирішення для конкретного випадку, можна буде легко помітити загальні тенденції та точки для зросту.

Мета і завдання роботи. Метою кваліфікаційної роботи є реалізація усіх етапів роботи генетичного алгоритму, починаючи від написання класів для його сутностей, і завершуючи використанням алгоритму для вирішення прикладних задач; визначення точок для зросту і вдосконалення створеного алгоритму; забезпечення можливостей для масштабування створеної python-бібліотеки.

Сфери застосування. Кінцевий продукт – бібліотека з загальною реалізацією генетичного алгоритму, прикладами його застосування на конкретних задачах, а також можливостями для доповнення та вдосконалення. Базову версію алгоритму можна буде використовувати як шаблон для вирішення більшості існуючих задач оптимізації, частина з них вже наведена вище.

Таким чином, якщо говорити про сфери науки та життя, які можна покращити шляхом використання якісної реалізації генетичного алгоритму, одразу на думку спадає логістика, машинне навчання, чисельні методи, молекулярна хімія, розподілені системи, розробка програмного забезпечення та багато інших.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ВІДОМОСТІ

Розділ 1.1. Історія та принципи роботи генетичного алгоритму

Засновником генетичних алгоритмів вважається Джон Голланд, який разом зі своїми колегами та студентами університету Мічиган займався дослідженнями на базі клітинних автоматів. Його робота «Адаптація в природних і штучних системах» є однією з ключових у сфері еволюційних алгоритмів пошуку.

Генетичні алгоритми є евристичними пошуковими алгоритмами, що базуються на принципах природньої селекції. Вони поєднують у собі виживання найбільш відповідних заданим правилам об’єктів (задаються у вигляді рядків даних) разом із структурованим, але при цьому рандомізованим, обміном інформації між ними.

На кожному новому етапі створюється множина штучних наборів даних за допомогою складових частин найкращих особин попереднього етапу, утворені штучні набори перевіряються на відповідність заданим умовам, у випадку невідповідності відкидаються, в успішному ж випадку – входять до складу поточного покоління.

Рандомізування під час утворення нових об’єктів – це не просто підбор нових варіантів «наосліп», насправді воно ефективно використовує історичні дані і дає змогу знаходити нові точки для пошуку, які потенційно дадуть кращий результат.

Розділ 1.2. Відмінність генетичних алгоритмів від інших

Для того щоб зрозуміти, як саме традиційні алгоритми для оптимізації можуть бути перевершені генетичними, особливо у питаннях стійкості і стабільності, необхідно визначити, чим саме вони відрізняються [2]:

- генетичні алгоритми здійснюють пошук виходячи з набору точок (популяції), а не з однієї точки
- для оцінки відповідності точок заданим умовам, використовують функцію пристосованості, яка по суті є цільовою для мінімізації або максимізації в залежності від умов задачі, в противагу похідним та іншим допоміжним функціям у традиційних алгоритмах
- вони працюють з кодуванням набору параметрів, а не з самими параметрами
- використовують не детерміновані, а імовірнісні правила переходу

Розділ 1.3. Основні поняття

Дамо визначення базовим поняттям, що використовуються в описі роботи генетичних алгоритмів:

- Ген – елемент хромосоми, що несе в собі інформацію про певну ознаку.
- Хромосома – впорядкований набір генів, зазвичай представлений у вигляді рядка з даними. Насправді в них містяться множини рішень задачі, які іноді називають точками пошуку.
- Генотип – набір хромосом конкретної особини. Несе в собі всю інформацію про її ознаки і властивості. Часто трапляється, що генотип складається лише з однієї хромосоми.
- Особина – елемент популяції, що має власний генотип.
- Популяція – набір особин.
- Особини-батьки – ті, у яких беруть хромосоми для схрещування та утворення нових особин.
- Особини-потомки – результати схрещування особин-батьків.
- Фітнес-функція (або функція пристосованості) – дає можливість визначити відповідність конкретної особини заданим умовам задачі.

В залежності від значення цієї функції, особа або виживає, залишаючись у популяції, або помирає при недостатньому рівні пристосованості.

Для більш чіткого розуміння того, як пов’язані вищезгадані поняття, розглянемо як приклад задачу пошуку максимуму функції. Ген в даному випадку – одна з координат, хромосома – точка з конкретними координатами, особина матиме одну хромосому і по суті буде репрезентувати собою точку в просторі. Популяцією буде набір таких точок. Функцією пристосованості буде сама функція, максимум якої ми шукаємо, основний критерій відбору особин – максимізувати своїми координатами дану функцію. Дана ситуація зображена на наступному графіку:

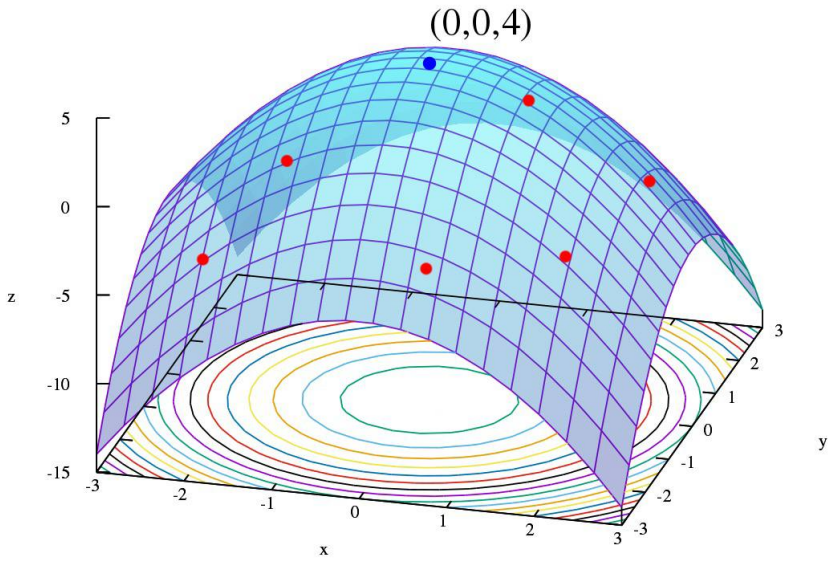


Рис. 1.1. Приклади особин у випадку пошуку максимуму функції

Точка (0,0,4), позначена синім, виявиться найкращою особиною, в той час як червоні точки будуть поступово виходити з популяції.

Розділ 1.4. Етапи роботи алгоритму

Генетичний алгоритм можна представити у вигляді послідовності таких дій:

1) Утворення початкової популяції.
2) Оцінка фітнес-функції особин в популяції.
3) *Оператор селекції-репродукції* – обрання особин-батьків для подальшого процесу розмноження.
4) *Оператор схрещування-кросинговеру* – створення особин-потомків від обраних пар батьків.
5) *Оператор мутації* – зміна хромосом нових особин.
6) Додавання новоутворених особин до популяції.
7) *Оператор редукції* – скорочення розширеної на попередньому кроці популяції до початкового розміру.
8) Перевірка умов виконання алгоритму – якщо не завершений – переходимо до другого пункту, якщо звершений – йдемо далі.
9) Обрання найкращої особини серед множини осіб кінцевих популяцій (оптимальний варіант не обов’язково буде в останній популяції). Це і буде результатом роботи алгоритму.

РОЗДІЛ 2. БАЗОВА РЕАЛІЗАЦІЯ АЛГОРИТМУ

Розділ 2.1. Реалізація основних класів

В першу чергу треба приблизно оцінити, з якими вхідними даними матиме справу алгоритм, а також у яких структурах будемо зберігати проміжні результати. Після цього програмно реалізуємо спроектовані структури. Всі описані нижче класи можна знайти в Додатку А.

Розділ 2.1.1. Обрання структур даних

Для початку треба пам’ятати те, що бібліотека має бути універсальною і гнучкою для будь-якого роду доповнень і покращень. Більшість етапів роботи генетичного алгоритму має більше двох методів, які будуть представлені як параметри відповідних функцій – таким чином ми дамо користувачу можливість обрати найкращий для його задачі методи. А задля того, щоб зрозуміти, наскільки гнучкими мають бути наші структури даних, зробимо порівняльну таблицю для двох можливих задач:

Завдання / Структура	Пошук екстремуму функції	Задача комівояжера
Вхідні дані	Функція, мінімум або максимум, обмеження на координати, розмірність	Список точок з їх координатами, перша точка в списку – початкова і кінцева
Хромосома	Точка, її координати – гени	Шлях - список точок через які треба послідовно пройти
Популяція	Набір точок	Набір шляхів

Функція пристосованості	Значення заданої в умові функції в обраній точці	Сумарна відстань, яку треба подолати, щоб пройти обраний шлях

Можна зробити висновок, що навіть при наявності лише двох доступних типів задач в бібліотеці, постають проблеми з обранням чітких і при цьому універсальних структур даних.

Розділ 2.1.2. Клас Solver

Ключовим у вирішенні задач буде клас *Solver* – при створенні його об’єкту, користувач має ввести такі атрибути:

- *task_type* - в початковому варіанті бібліотеки це буде *func_extremum* (пошук екстремуму функції) або *TSP* (задача комівояжера)
- *create_population_method* – метод для створення стартової популяції. Доступними будуть *shotgun* (стратегія рушниці) та *focusing* (стратегія фокусування) – вона потребує додаткового параметра *focus_boundaries*
- *selection_method* – метод селекції, тобто визначення множини батьків, можна обрати *proportional* (метод рулетки), *ranging* (метод ранжування), *annealing* (алгоритм імітації відпалу, або його ще називають методом Больцмана)
- *recombination_method* – метод рекомбінації/кросинговеру, варіанти: *single_point, multi_point, homogeneous* - двійкові кросинговери; *discrete, transitional, linear* – дійсні рекомбінації
- *mutation_method* – метод мутації, можливі *binary* (класичний двійковий), *inversion* (оператор інверсії) та *real_number* (мутації для дійсних чисел)
- *reduction_method* – метод скорочення популяції, для обрання доступними є такі: *net_replacement* (чистої заміни), *elite_scheme*

(елітарний), *uniform_random*_replacement (рівномірно-випадкової заміни), *proportional* (пропорційний) та *selective_scheme* (селективний)

- *stop_criterion* – опис умови для зупинки роботи алгоритму, можна обрати *generations_limit* (алгоритм обробить конкретне число поколінь) або *fitness_stability* (алгоритм зупиниться, коли найкраще значення фітнес-функції для останніх трьох поколінь буде відрізнятись по модулю не більше, ніж на певне числове значення)
- *stop_value* – числове значення для зупинки, в залежності від попереднього параметра це буде відповідно або максимальна кількість поколінь, або точність, тобто різниця між фітнес-функціями найкращих особин
- *n_pop* – кількість особин в популяції
- у випадку, якщо користувач обрав задачу пошуку екстремуму функції, необхідно ввести також такі додаткові дані як:
    - *function* – функція у вигляді рядка, з дотриманням усіх синтаксичних правил мови python; за допомогою методу eval ми переведемо її в лямбда-функцію, і передаємо в атрибут *fitness_func.* Треба зауважити, що нам потрібна додаткова *transitional_func* – вона зберігатиме саму функцію від змінних x та y (або від x), а *fitness_func* буде приймати список [x;y] (або [x]) та до нього застосовувати *transitional_func.* Це робиться тому, що нам необхідно передавати саме список у атрибут *gene_list* хромосоми
    - *goal* – дозволяє обрати максимум чи мінімум
    - *dimensions* – розмірність функції, доступні тільки значення 2 або 3

    - *boundaries* – в залежності від розмірності, це буде список відповідно на 4 чи 6 елементів – максимальні та мінімальні допустимі значення координат
- якщо ж було обрано задачу комівояжера, то необхідно лише ввести параметр *points* – список точок (кожна точка представляє собою список з 2 елементами – x та y координатами на площині), для яких треба створити оптимальний маршрут. При цьому перша точка цього списку вважається початковою (та кінцевою) для замкненого маршруту. При цьому *fitness_func* буде собою являти сумарну відстань для подолання обраного маршруту, це реалізовано через додаткову функцію *total_distance*, яка приймає список координат точок, і рахує сумарну відстань для їх поступового обходу (з останньої точки списку переходимо в першу).

Розділ 2.1.3. Клас Population

Як було вказано вище, популяція – множина особин. Але в даному контексті особина буде представлена однією хромосомою, тобто це будуть взаємозамінні поняття. При ініціалізації об’єкту, окрім кількості осіб в популяції та даних по задачі, введемо наступні атрибути:

- *best_chromosome_info* – це буде список генів та значення фітнес-функції для найкращої особини поточної популяції, отримаємо її тільки в кінці обробки популяції, тому зараз це None;
- *chromosome_list* – список хромосом поточної популяції. За замовчуванням (і на початку) порожній список, але під час роботи циклу ми будемо передавати наступній популяції об’єкт попередньої популяції;
- *parents_list* – список особин-батьків, відібраних після селекції
- *children_list* – список утворених особин-нащадків після рекомбінації

- *mutation_list* – список особин, які пройдуть через мутацію (забираємо зі списку нащадків)

Розділ 2.1.4. Клас Chromosome

Об’єкт класу *Chromosome* матиме такі атрибути як *gene_list* та *fitness_func*, а також під час ініціалізації визначимо *fitness_value* як результат застосування функції пристосованості до набору генів даної особини. Створимо метод *get_info* – він виводитиме список генів та значення фітнес-функції.

Розділ 2.2. Методи створення початкової популяції

Тепер можна переходити до реалізації методу *create* класу *Population*, який в залежності від параметра *create_population_method* виконає поставлене завдання. Методи обирались найефективніші, тому, наприклад, формування повної множини, що містить всі можливі рішення, не бралось до уваги, оскільки така популяція не матиме змоги розвиватися.

При формуванні заданого числа особин, почнемо взаємодіяти з класом *Chromosome.* Зрозуміло, що для пошуку екстремуму функції треба буде працювати з дійсними числами на деякому проміжку, тому для генерації особин використовуватимемо функцію з пакету numpy - random.uniform. В той же час, для комівояжера ситуація зовсім інша – ми маємо «пропонувати» маршрути, тому створимо для цього спеціальну функцію *mix_points*, яка на вхід приймає список точок, і видає перемішаний список, при цьому перший елемент залишається на тому ж місці.

Розділ 2.2.1. Стратегія «рушниці»

Ідея полягає в тому, щоб створити досить велику випадкову підмножину рішень. Один з найкращих способів, оскільки в результаті

еволюції є можливості перейти в інші області пошуку (не зациклюючись на якихось конкретних). Оскільки користувач в будь-якому випадку вводить базові обмеження на числові значення генів – *boundaries,* то в цих межах і будемо створювати початкових особин.

Розділ 2.2.2. Стратегія фокусування

На відміну від попереднього методу, використовується тільки тоді, коли вже наявне припущення щодо правильного рішення – тоді можна за меншу кількість кроків прийти до результату, генеруючи початкові особини лише в чітких заданих межах, які зберігаються в атрибуті *focus_boundaries.*

Розділ 2.3. Методи відбору особин-батьків

Коли популяція вже готова, можна перейти до наступного кроку – на основі функції пристосованості, обираємо певну кількість найкращих особин, які зможуть поділитися своїми генами для формування якісних нащадків.

Клас *Population* також має атрибут *parents_list* – це буде поточний список батьків, який для кожного етапу буде оновлюватись, на момент створення об’єкту він порожній.

Розділ 2.3.1. Метод «рулетки» (пропорційний відбір)

Він є одним з найбільш практичних методів відбору найякісніших особин, але використовується лише в задачах максимізації цільової функції. Згідно методу, особини (рішення) відображаються як відрізки на лінії (або як сектора рулетки) таким чином, щоб їх розмір був пропорційний значенню цільової функції [3]. Це показано в наступному прикладі, представленому в таблиці:

Особина	1	2	3	4	5	6	7	8	9	10	11
Функція	2,0	1,8	1,6	1,4	1,2	1,0	0,8	0,6	0,4	0,2	0,0
Ймовірність	0,18	0,16	0,15	0,13	0,11	0,09	0,07	0,06	0,03	0,02	0,0

На відрізку [0,1] для кожної особи будуються відрізки, довжини яких пропорційні ймовірностям вибору особин, як це показано на наступному рисунку:

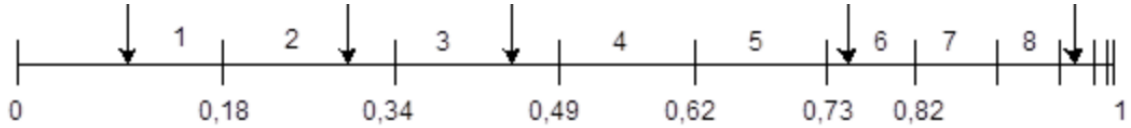


Рис. 2.1. Ймовірності вибору особин методом рулетки [3]

Далі випадково генеруються числа з діапазону [0,1] і в проміжну популяцію обираються ті особини, в чий відрізок потрапляють ці випадкові числа. Таким чином, кожна спроба є випадковим числом з відрізка [0,1], в результаті чого вибирається особина, що відповідає обраному відрізку. В даному прикладі в результаті п'яти спроб в проміжну популяцію були обрані особини 1, 2, 3, 6 і 9.

Метод має свої недоліки [3]:

- залежність від позитивних значень цільової функції (функція повинна для абсолютно всіх особин набувати позитивного значення, що є проблематичним для задачі пошуку екстремуму);
- метод без модифікацій може використовуватися лише у завданнях максимізації функції. Тобто для задачі комівояжера, у якій цільова функція завжди більше нуля, цей спосіб також не підходить, оскільки перед нами стоїть мета мінімізувати пройдений шлях;
- просте додавання великої константи до цільової функції може звести метод фактично до випадкового вибору;
- особини з дуже маленькими значеннями фітнес-функції дуже швидко виключаються з популяції, що може призвести до передчасної збіжності

генетичного алгоритму, тобто не всі множини рішень будуть враховані під час пошуку найоптимальнішого.

У нашому випадку метод реалізується наступним чином:

1) Створюємо об’єкт DataFrame, який міститиме хромосоми та відповідні дані про них – значення фітнес-функції (абсолютне та пропорційне), кумулятивні пропорційні ймовірності (як вказано вище) та флаг ‘parent’, що вказуватиме, чи увійде в *parents_list* хромосома;
2) Генеруємо множину випадкових значень від 0 до 1, кількість – половина особин в популяції;
3) Проходимося по DataFrame, знаходимо, на які хромосоми попали ці випадкові числа, заносимо їх до списку, в кінці прибираємо дублікати.

Розділ 2.3.2. Метод ранжування

Ранжування підходить як при максимізації, так і при мінімізації цільової функції. Також цей метод не потребує масштабування для запобігання занадто ранньої збіжності, на відміну від попереднього.

Спочатку сортуємо особин відповідно до значень їх функцій пристосованості. Необхідно зауважити, що саме значення не впливає на ймовірність селекції особини – важлива лише позиція в упорядкованому списку, що дає більше стійкості. Найбільш часто застосовують лінійне ранжування, яке задається наступною формулою:

$$P_s(a_i) = \frac{1}{N}(a - (a - b)\frac{i - 1}{N - 1})$$

При цьому a – випадкове число між 1 та 2 включно, $b = 2 - a$, N – кількість особин в популяції, i – номер особини в упорядкованому списку.

Програмна реалізація методу аналогічна попередньому, відрізняється лише формулою пошуку вірогідності, так саме і в наступному пункті.

Розділ 2.3.3. Метод імітації відпалу

Його також називають методом Больцмана. В цьому випадку використовується такий же підхід, як і в популярному методі оптимізації «моделювання відпалу», якій ґрунтується на природному процесі – кристалізації речовини при зміні температури.

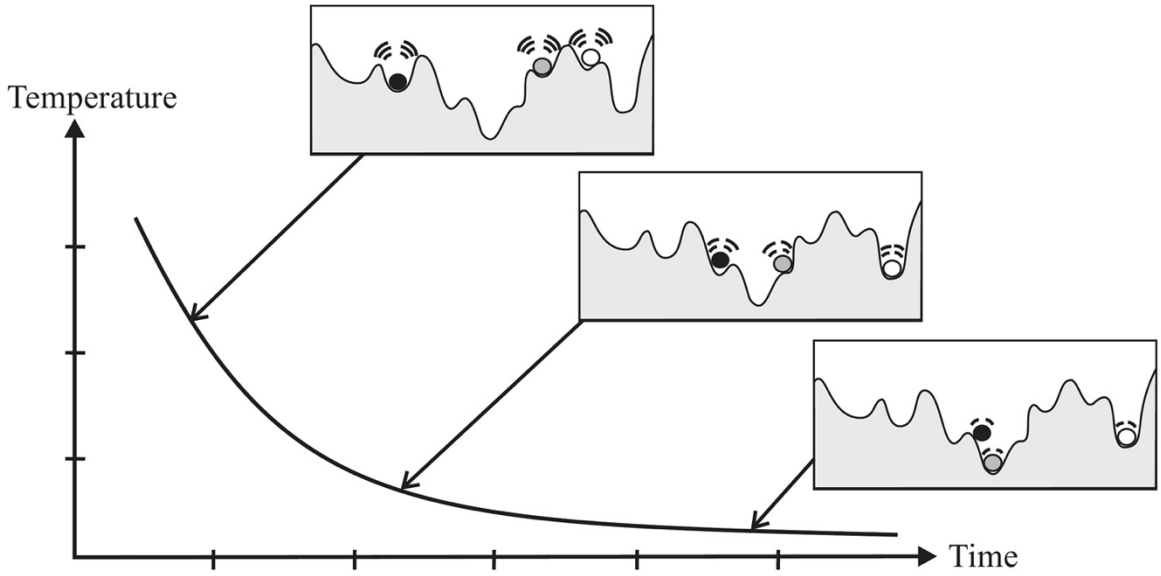


Рис. 2.2. Демонстрація пошуку мінімуму функції в залежності від «температури» при імітації відпалу [5]

Процесом відбору керує «штучна температура». Задля цього вводимо змінну T, яка, починаючи з деякого досить великого значення буде поступово зменшуватись згідно деякого закону, і таким чином змінюватиме вірогідність селекції особин.

Ймовірність відбору особи, що має значення фітнес-функції $f(i)$ рахується наступним чином:

$$P_s(a_i) = \frac{1}{N}\left(\frac{e^{f(a_i)/T}}{e^{f(a)/T}_{avg}}\right)$$

При цьому $e_{avg}^{f(a)/T}$ – середнє значення $e^{f(a_i)/T}$ для поточної популяції. Зрозуміло, що з пониженням температури різниця значень $P_s(a_i)$ між найкращими та найгіршими особинами збільшується. Це дає змогу на кінцевому етапі зменшити область пошуку до найбільш перспективної частини простору рішень, при цьому буде збережена достатня різноманітність в популяції. На багатьох класах задач цей метод дає набагато більш надійні результати ніж «рулетка» чи ранжування.

Розділ 2.4. Методи визначення пар для схрещування

Коли ми визначились із множиною особин-батьків, можна перейти до створення пар для схрещування. Існує два основних варіанта:

- випадковий вибір батьківської пари, при цьому обидві особини обираються із проміжної популяції. Одна особина може бути залучена в декількох парах. Зазвичай використовується саме цей метод – він універсальний для більшості класів задач, але його ефективність знижується із зростом кількості особин в популяції;
- селективний вибір допускає у батьки лише тих особин, що мають *fitness_value* більше за середнє по популяції при рівній вірогідності цих кандидатів утворити пару. Такий підхід дає швидшу збіжність алгоритму, але не підходить для мультимодальних задач – наприклад, коли є декілька можливих екстремумів у задачі їх пошуку – селективний вибір може просто пропустити найоптимальніше рішення та прийти до некоректного локального екстремуму;
- іноді використовуються такі підходи як інбридинг та аутбридинг. У них формування пари відбувається з урахуванням близької чи дальньої спорідненості відповідно. Під спорідненістю мається на увазі відстань між особинами популяції у просторі параметрів. У зв'язку з цим розрізняють генотипний та фенотипний інбридинг та

аутбридинг. В інбридингу перший член пари обирається випадково, а другим є найближча до нього особина. Аутбридинг же навпаки, формує пари з дуже далеких по спорідненості особин [4]. Комбінація цих методів добре себе показує під час вирішення мультимодальних задач.

Розділ 2.5. Методи схрещування

Ці методи можна розділити на два класи – оператори двійкової та дійсної рекомбінації.

Розділ 2.5.1. Одно точковий кросинговер

Найбільш простий та звичний варіант – обирається точка для схрещування, і відбувається обмін генами батьківських хромосом після цієї точки. Приклад можна зобразити у вигляді наступної таблиці:

O_1	1	0	1	0	1	0	1	0	1
O_2	0	0	1	1	0	0	1	1	0
H_1	1	0	1	0	0	0	1	1	0
H_2	0	0	1	1	1	0	1	0	1

Де перші дві особини – батьківські, останні дві особини – нащадки, обмін відбувся після 4-го гену.

Розділ 2.5.2. Багато точковий кросинговер

Також є варіант, коли обирається декілька точок, і по цим межам діляться хромосоми батьків для передачі генів. Візьмемо для прикладу тих же батьків як і в одно точковому кросинговері, але точки поділу буть 2, 4 та 6, 8:

O_1	1	0	1	0	1	0	1	0	1
O_2	0	0	1	1	0	0	1	1	0

H_1	1	0	1	1	1	0	1	1	1
H_2	0	0	1	0	0	0	1	0	0

Розділ 2.5.3. Однорідний кросинговер

Цей метод відрізняється від попередніх, оскільки створює суміш генів кожного із батьків. Кожен ген для дитини – копія відповідного гена одного із батьків, тобто потенційно між будь-якими генами може бути точка кросинговеру.

Для цього необхідно випадковим чином згенерувати двійкову маску схрещування тієї ж довжини що і кількість генів у хромосомах батьків. Далі створюємо правило – якщо на певній позиції в масці 0 – то для цього гена беремо відповідний ген першого з батьків, якщо в масці 1 – то ген другого з батьків, і таким чином заповнюємо хромосому нащадку. Знову візьмемо той же приклад:

Маска	1	1	1	0	0	1	0	1	1
O_1	1	0	1	0	1	0	1	0	1
O_2	0	0	1	1	0	0	1	1	0
H	1	0	1	1	1	0	1	1	1

Розділ 2.5.4. Обмежений кросинговер

В цьому методі точки схрещування обираються лише там, де у батьків відрізняються гени. Наприклад:

O_1	1	0	1	0	1	0	1	0	1
O_2	0	0	1	1	0	0	1	1	0
H_1	1	0	1	1	0	0	1	0	1
H_2	0	0	1	0	1	0	1	1	0

Розділ 2.5.5. Дискретна рекомбінація

Схожа на однорідний двійковий кросинговер, використовується для генів будь-якого типу. Тут аналогічно випадковим чином генерується маска, яка показує, який із батьків має передати ген на певній позиції. Розглянемо приклад з дійсними числами, створимо дві окремі маски для двох потомків:

$О_1$	11,5	4	1,6	2,25
$О_2$	8,9	3,1	5	7,6
$Маска_1$	1	2	2	1
$Н_1$	11,5	3,1	5	2,25
$Маска_2$	2	1	1	1
$Н_2$	8,9	4	1,6	2,25

Розділ 2.5.6. Проміжна рекомбінація

Ця рекомбінації застосовується лише для дійсних чисел. Під час неї гарантовано буде створено два нащадки. Гени нащадків будуються на основі генів обох батьків – береться значення в околицях (або між їх генами) за принципом:

$$\begin{cases} c_1 = p_1 + \beta_1(p_2 - p_1) \\ c_2 = p_1 + \beta_2(p_2 - p_1) \end{cases}$$

В цьому випадку p_1 та p_2 – значення певного гена 1-го та 2-го батьків відповідно; c_1 та c_2 – значення цього ж гена для двох нащадків; β_1 та β_2 – множники, що обираються випадковим чином з проміжку $[-\sigma; 1+\sigma]$. Для звичайної проміжної рекомбінації беруть $\sigma = 1$, а множники для кожного з генів $\beta_i \in [0,1]$. Для узагальненого варіанту $\sigma = 0{,}25$. Також, якщо є мета обирати гени нащадків між генами батьків, достатньо взяти $\sigma = 0$. Тоді c_1 та c_2 для довільного гена будуть в межах $[p_1, p_2]$. Розглянемо на минулому прикладі:

p_1	11,5	4	1,6	2,25
p_2	8,9	3,1	5	7,6
β_1	1,1	0,5	0,7	0,1
c_1	8,64	3,55	3,98	2,785
β_2	0,4	1,3	0,35	0,8
c_2	10,46	2,83	2,79	6,53

Для наочності, приклад обчислення першого гена для обох нащадків:

$$\begin{cases} c_1 = 11{,}5 + 1{,}1 * (8{,}9 - 11{,}5) = 8{,}64 \\ c_2 = 11{,}5 + 0{,}4 * (8{,}9 - 11{,}5) = 10{,}46 \end{cases}$$

Розділ 2.5.7. Лінійна рекомбінація

Аналогічно проміжній рекомбінації, але множники β_i беруться однакові для всіх генів i-того нащадка. Приклад:

p_1	11,5	4	1,6	2,25
p_2	8,9	3,1	5	7,6
β_1	0,5			
c_1	10,2	3,55	3,3	4,925
β_2	0,1			
c_2	11,24	3,91	1,94	2,785

Розділ 2.6. Методи мутації

Коли схрещування завершено, утворені нащадки з певною ймовірністю P_m проходять через мутацію.

Розділ 2.6.1. Класична двійкова мутація

Цей метод актуальний для бінарних даних. Для кожної нової особини обирається певний ген, і з невеликою ймовірністю (від 0,01 до 0,001) він змінюється на інверсійний, наприклад:

H_1	1	0	0	1	1	1
$H_1^{'}$	1	0	0	1	0	1

Розділ 2.6.2. Оператор інверсії

Також різновид мутації, схожий на минулий, але обирається не одна позиція, а послідовність генів:

H_1	1	0	0	1	1	1
$H_1^{'}$	1	1	0	0	1	1

Розділ 2.6.3. Мутації над дійсними числами

У випадку, коли гени представлені дійсними числами, мутація відбувається шляхом додавання до гена (або генів) особини деякого випадкового значення, яке називають кроком мутації. Він не обов'язково стабільний, іноді його змінюють в процесі вирішення задачі. Замалий крок – безпечний варіант, що веде до високої точності, але це уповільнює збіжність алгоритму. Однорідна мутація – та, що має сталий крок і ймовірність.

Загалом оператор виражається таким чином:

$$V_m = V \pm \mu * \delta$$

Де V – значення дійсного гена до мутації, V_m – після; μ – діапазон зміни гена; δ – випадкове число від 0 до 1.

Задля більшої ефективності, на етапі, коли генетичний алгоритм вже прямує до збіжності, вірогідність мутації краще зменшувати. На початкових етапах достатньо задати P_m^0 від 0,05 до 0,1.

Щоб обрати закон зменшення ймовірності мутації, можна знову повернутись до методу моделювання відпалу, тоді якщо t – номер покоління, то можна виразити:

$$P_m = P_m^0 * e^{-\frac{1}{t}}$$

Мутації будемо проводити не на рівні класу *Chromosome,* а в *Population* – достатньо згенерувати необхідні гени, і з них створити нову хромосому, додавши її у список.

Розділ 2.7. Скорочення поточної популяції

Один з останній кроків під час кожного циклу роботи алгоритму – очищення популяції від неякісних особин. Так як ми з самого початку роботи алгоритму задаємо число особин в популяції – *n_pop*, а після роботи попередніх етапів на поточному циклі, утворилась певна кількість нащадків шляхом схрещування і мутації, треба видалити особин на це ж число.

По-перше визначимось з тим, які особини складають поточну повну популяцію:

- *parents_list* – всі обрані батьківські особини;
- *children_list* – всі нащадки, які не пройшли через мутації;
- *mutation_list* – нащадки, які прошли мутацію;
- *population_remainder* – особини, які не були обрані під час селекції, але вони не обов’язково мають погану фітнес-функцію – ми могли їх не обрати просто через вірогідності. Саме тому не є раціональним відкидання всієї цієї частини, бо тоді з’явиться ризик передвчасної збіжності алгоритму. Цю множину ми будемо враховувати на одному рівні з іншими особами.

Розділ 2.7.1. Чиста заміна

Якщо при схрещуваннях та мутаціях особин-нащадків додалось стільки, скільки було під час селекції та парувань обрано батьків, одна з найпростіших опцій – видалити всіх особин-батьків. Але при цьому є ризик викинути хороші рішення, і залишити неоптимальні. Це сильно сповільнить роботу алгоритму.

Розділ 2.7.2. Елітарна схема

У випадку, якщо нащадків згенеровано менше, ніж батьків, можна замінювати потомками найгірших (по фітнес-функції) батьків, таким чином в наступній популяції будуть наявні всі потомки з попередньої, та найкращі батьки. Але при такій схемі є ризик занадто ранньої збіжності до одного з локальних екстремумів.

Розділ 2.7.3. Рівномірна випадкова заміна

Даний метод аналогічний попередньому, тобто потомків згенеровано менше ніж батьків, але видаляти особин-батьків будемо випадково, незалежно від їх фітнес-функції. Таким чином ми можемо позбутися ризику застрягнути біля одного локального екстремуму – паралельно будемо розглядати також і інші області рішень.

Розділ 2.7.4. Пропорційна редукція

В даному випадку потомків генерується більше, ніж необхідно для заміни. Тоді задану кількість батьків замінюють виключно найкращими нащадками відповідно до їх функції пристосованості.

Розділ 2.7.5. Селекційна схема

В цій схемі батьки з нащадками – на рівних правах, вони попадають в одну репродукційну групу, в якій всі особини проходять ранжування відповідно до їх фітнес-функції, і в наступне покоління йде лише певна кількість найкращих особин – наприклад, у яких значення *fitness_value* вище ніж середнє по групі.

Розділ 2.8. Зупинка роботи алгоритму

Як було зазначено раніше, у користувача є два варіанти, коли зупиниться цикл. В залежності від цього по-різному працює атрибут *stop_value* та змінна *stop_condition*.

Одразу після створення початкової популяції задаються змінні *generation_num* = 0 (номер поточного покоління) та *accuracy* = 1 (різниця між функціями пристосованості найкращих особин останніх трьох поколінь), далі йде перевірка:

- коли *stop_criterion* = 'generations_limit', цикл зупиниться при досягненні *generation_num* значення *stop_value;*
- коли *stop_criterion* = 'fitness_stability', цикл припинить роботу на моменті становлення *accuracy* нижче *stop_value.*

Після зупинки роботи циклу залишається обрати найкращу особину з останніх популяцій – це і буде оптимальний результат. Далі питання постає в тому, як його вивести користувачу. Для екстремуму достатньо вказати координати та значення функції, а от для задачі комівояжера необхідно продемонструвати графік шляху.

РОЗДІЛ 3. ВИКОРИСТАННЯ АЛГОРИТМУ ДЛЯ ЗАДАЧ

Розділ 3.1. Задача пошуку екстремуму функції

Найпростіший приклад, на якому можна продемонструвати роботу генетичного алгоритму – пошук мінімуму або максимуму заданої функції. Для прикладу візьмемо наступну функцію в просторі, нам необхідно знайти її мінімум:

$$z = -x * sin\sqrt{|x|} + y * cos\sqrt{|y|}$$

Також задаємо обмеження для пошуку:

- x ∈ [-30; 30]
- y ∈ [-10; 10]
- потужність популяції - 1000

Графік виглядає наступним чином (програму можна знайти у Додатку Б):

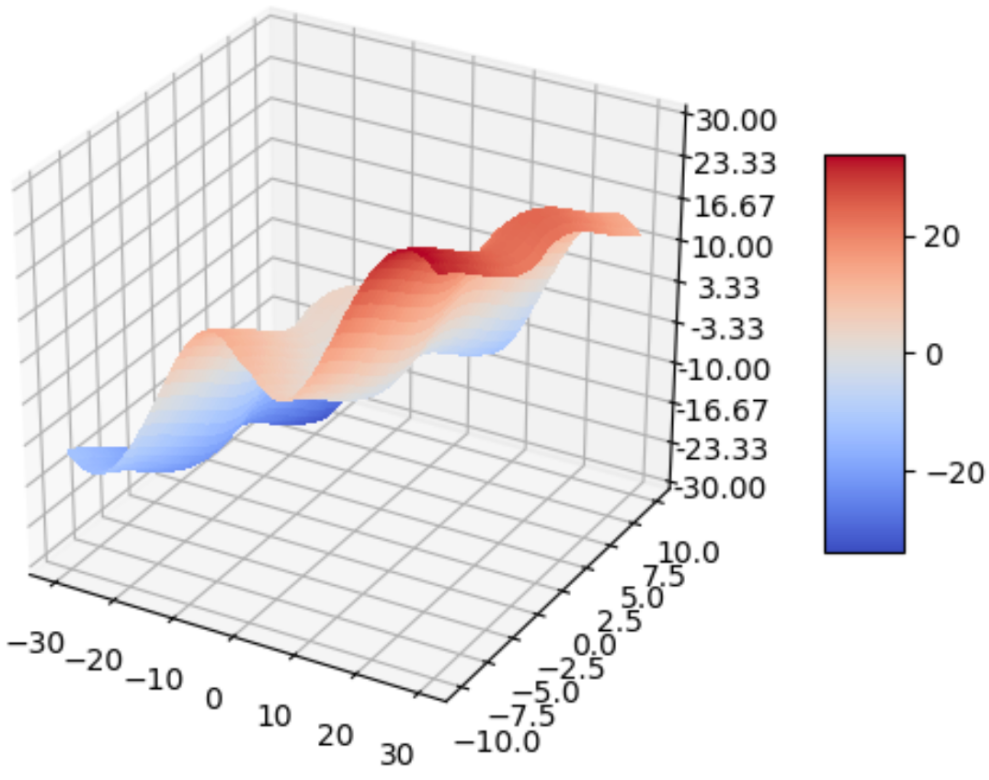


Рис. 3.1. Графік функції для пошуку мінімуму

Створимо об’єкт класу Solver, передавши йому наступні параметри:

- *func_extremum* – тип задачі
- *shotgun* – метод генерації початкової популяції
- *annealing* – метод селекції
- *transitional* – метод рекомбінації
- *real_number* – метод мутації
- *uniform_random_replacement* – метод редукції
- *n_pop* = 1000
- *function* – передаємо вищезгадану функцію у вигляді рядку, з дотриманням усіх правил синтаксису мови python
- *goal* = ‘min’
- *dimensions* = 3
- *boundaries* = [-30, 30, -10, 10]
- *focus_boundaries* = None – оскільки обрали метод «рушниці» для генерації початкової популяції, а не метод фокусування

Маємо такий результат: мінімум функції z = -34.008, він досягається в точці (-26.092; 9.932):

```
The results of Genetic Algorithm:     (-34.00810062387917, -26.092160278525693, 9.932246296008586)
```

Розділ 3.2. Пошук екстремуму іншими алгоритмами

Разом з генетичним, застосували також алгоритм випадкового пошуку та імітації відпалу, результати вийшли наступні:

```
The results of Random Search:         (-33.357702364035404, -27.197605675865656, 9.538858315038862)
The results of Simulated Annealing:   (-33.55382098496566, -26.46783085317394, 9.534693334365237)
The results of Genetic Algorithm:     (-34.00810062387917, -26.092160278525693, 9.932246296008586)
```

Тобто, генетичний алгоритм вийшов до оптимальнішого розв’язку, ніж ці два алгоритми, при цьому за ту ж кількість кроків. Для того щоб впевнитись в коректності і стабільності, запустимо ці три алгоритми 10 разів та порівняємо результати:

```
Comparison table:

Random Search                Simulated Annealing          Genetic Algorithm
-34.045817820382             -32.717878668476             -33.383880091578
-34.043166495641             -33.858270351333             -32.412483587045
-33.535536844422             -24.682160066141             -33.769195422480
-33.184101983140             -33.826990977136             -33.584619534444
-34.061386457416             -32.257328166524             -33.579696672915
-23.386752498169             -33.765173277021             -33.842291395600
-32.773115661859             -27.013350100552             -33.940861556569
-22.913013386748             -31.843987193419             -33.245565083494
-33.690117559660             -33.320208939916             -33.771118875946
-34.062635404867             -32.645396263731             -34.058007149107
```

Можна впевнитись в тому, що наш алгоритм найчастіше видає оптимальніший результат, ніж *random_search* та *simulated_annealing*.

Розділ 3.3. Задача комівояжера

Задача комівояжера (TSP – traveling salesman problem) – алгоритмічна задача, що полягає в пошуку найкоротшого шляху між набором точок, кожну з яких необхідно обійти, при цьому повернувшись в початкове місце. Зазвичай, точки вважаються містами, як комівояжер має відвідати, його мета – пройти мінімальну дистанцію і витратити найменш можливу кількість ресурсів.

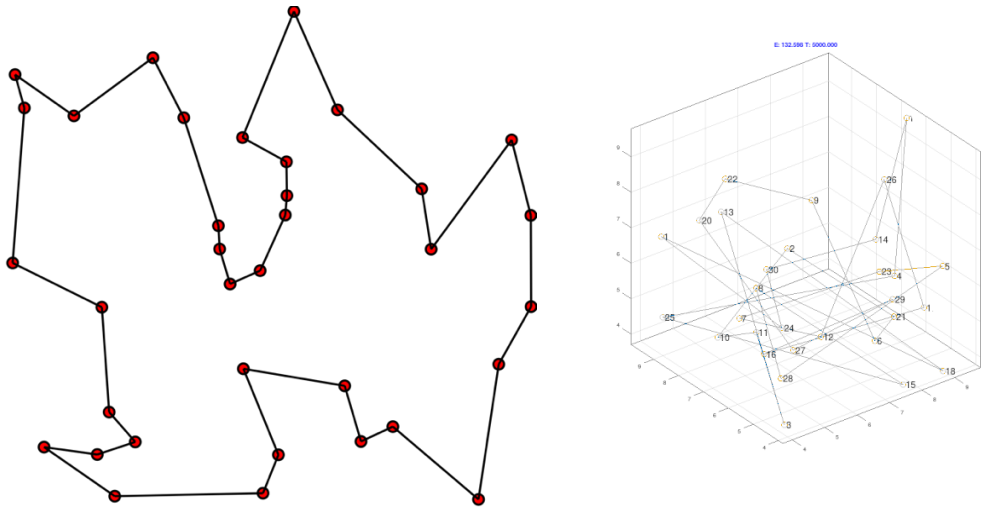


Рис. 3.2. Приклади рішень задачі комівояжера в площині та в просторі

Також цю задачу можна інтерпретувати в комп’ютерних науках як пошук найбільш ефективного шляху для даних між декількома вузлами. Більш того, її застосовують для визначення мережевих або апаратних методів оптимізації. Вперше це було описано ірландським математиком У. Р. Гамільтоном та британським математиком Т. Кіркманом 1800-х роках за допомогою створення гри, яка вирішувалась знаходження циклу Гамільтона, який є неперекриваючим маршрутом між усіма вузлами [6].

Ця задача вивчалась десятиліттями, за цей час була запропонована величезна кількість рішень. Найпростіше і інтуїтивно зрозуміло – спробувати всі шляхи, рахуючи при цьому сумарну відстань, і обрати оптимальний варіант. Але очевидно, що це займе багато часу і ресурсів, особливо зі зростанням кількості міст. Багато вирішень застосовують евристики, що надають результати ймовірності. Але від цього результати стають приблизними і не обов’язково оптимальними. Також деяка множина рішень зав’язана на алгоритмах розгалуження, наприклад Монте-Карло.

Через всі фактори, TSP зазвичай фокусується не на пошуку найбільш оптимального шляху, а на знаходженні найдешевшого вирішення поставленого завдання – саме тому перевага надається наближеним та швидким алгоритмам. Одним із таких рішень є генетичний алгоритм, який дасть не лише прийнятну швидкість виконання, але і досить точний результат.

Отже, користувач задає список координат [x; y] – точок на площині. Візьмемо для прикладу 27 точок, згенеровані координати яких по осі X будуть від 20,000 до 28,000, по осі Y від 9,000 до 18,000, на малюнку видно їх розташування:

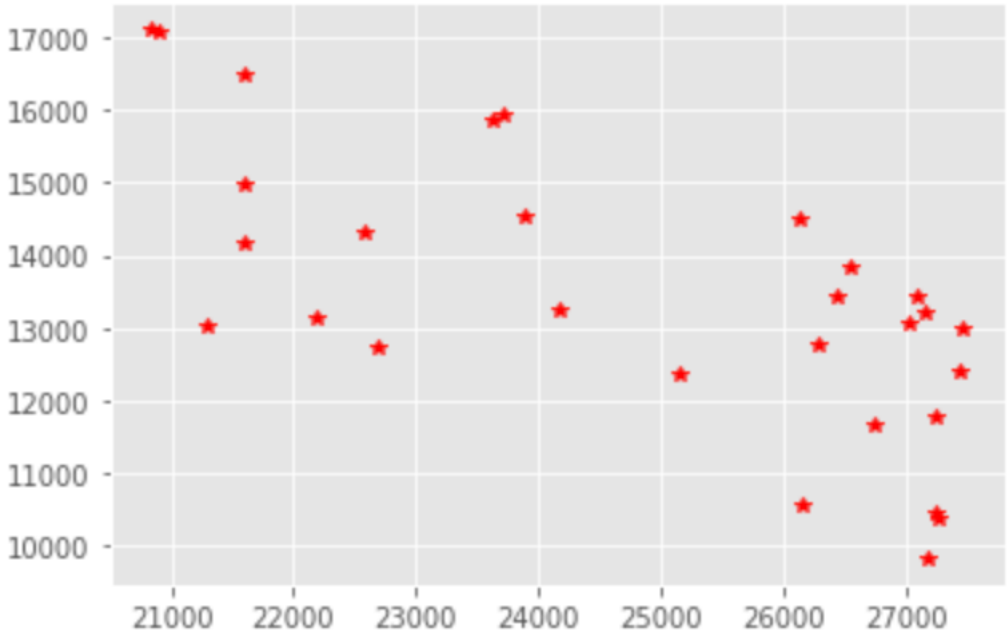


Рис. 3.3. Візуалізація вхідних даних для задачі комівояжера

При цьому перша точка (в даному випадку – у лівому верхньому куті) зі списку вважається базовим містом, і у будь-якому маршруті буде початковою та кінцевою одночасно. Очевидно, не треба обирати ціль задачі – це буде мінімізація. Створюємо об’єкт класу *Solver*, передавши такі атрибути:

- *TSP* – тип задачі
- *shotgun* – метод генерації початкової популяції
- *ranging* – метод селекції
- *multi_point* – метод рекомбінації
- *inversion* – метод мутації
- *elite_scheme* – метод редукції
- *n_pop* = 1000
- *goal* = ‘min’
- *points* – список точок, візьмемо 27, згенеровані координати по осі X будуть від 20,000 до 28,000, по осі Y від 9,000 до 18,000

Застосувавши метод solve, маємо результуючий маршрут у вигляді списку точок, візуалізувавши його, маємо наступне:

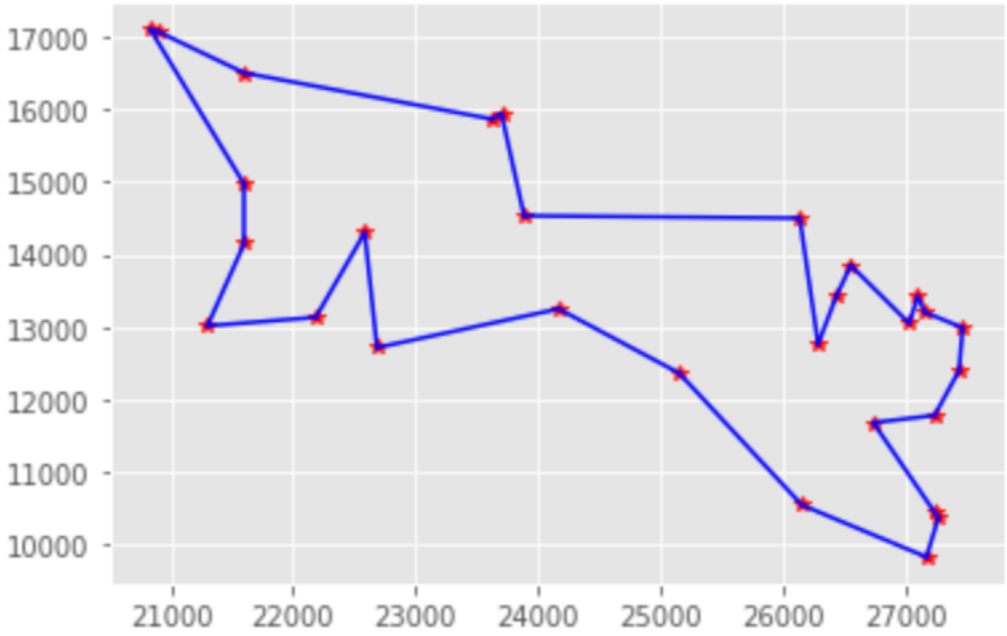


Рис. 3.4. Результат роботи генетичного алгоритму для задачі комівояжера

Як бачимо, обраний маршрут не має явних недоліків та локальних неоптимальних частин. Тим паче, ми задали невелике число потужності популяцій, при його збільшенні алгоритм матиме кращі результати виконання.

Розділ 3.4. Інші алгоритми для задачі комівояжера

Одним з найпопулярніших методів для розв’язання TSP вважається жадібний алгоритм – він полягає в обранні найоптимальнішого варіанту на поточному етапі, тобто найближчої точки тут і зараз. Але проблема в тому, що таким чином є ризик пропустити деякі точки, які потім все одно необхідно буде обходити, що в сумі дасть більшу загальну відстань. Результат для нашого прикладу буде такий:

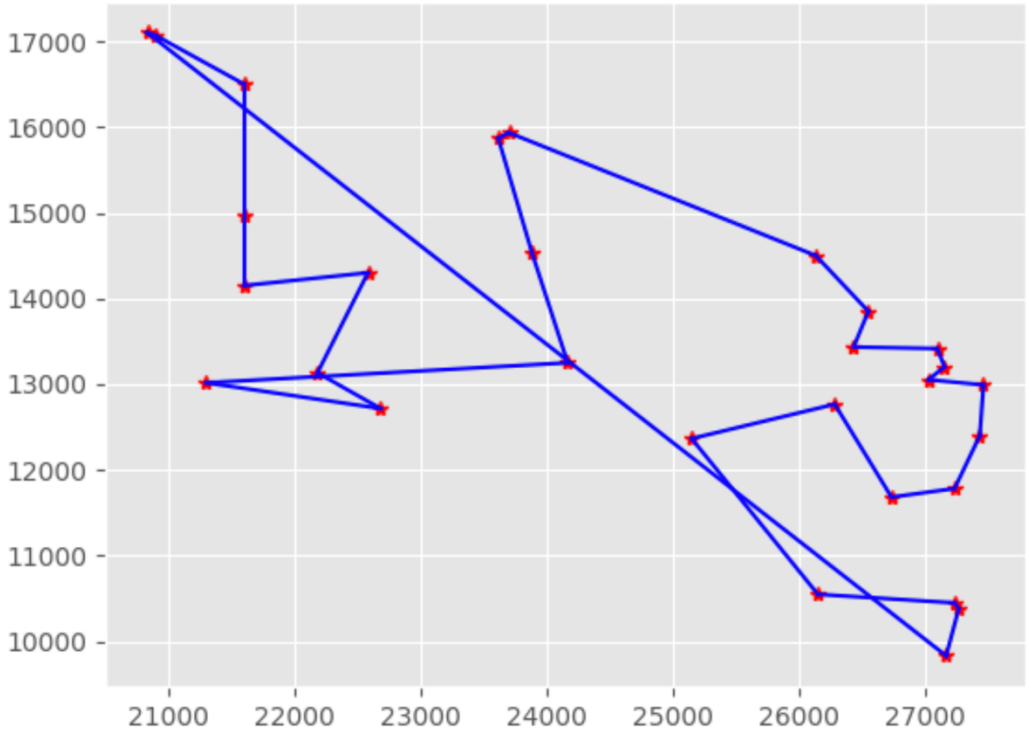


Рис. 3.5. Результат роботи жадібного алгоритму

Бачимо, що за рахунок того, що на кожному етапі обиралась найближча точка, не врахувалось те, що до останньої (тобто нульової) треба пройти досить великий шлях. Також помітні локальні неоптимальні частини маршруту.

Наступний метод – випадкового пошуку. Він ґрунтується на створенні випадкових шляхів, модифікації найкращих з них, і після проходження певного числа таких циклів отримаємо найбільш оптимальний розв’язок. На нашому наборі вхідних даних маємо:

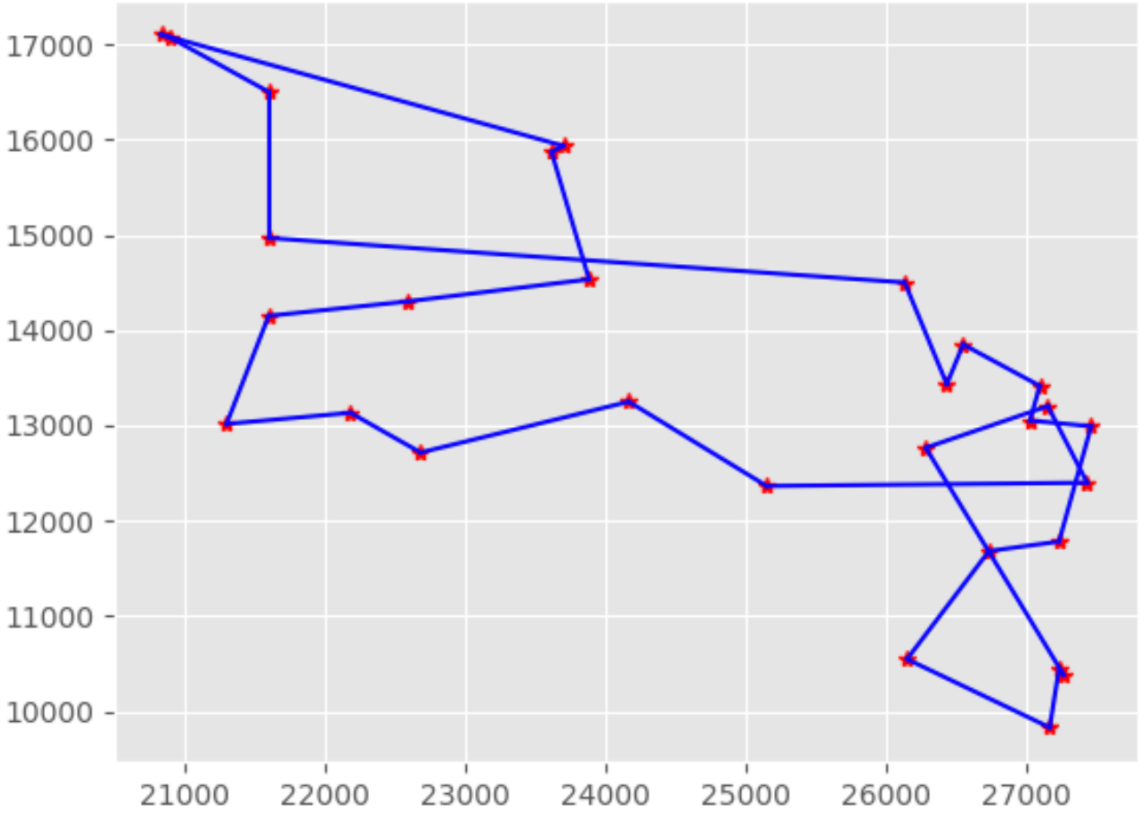


Рис. 3.6. Результат роботи алгоритму випадкового пошуку

Одразу можна помітити локальний неоптимальний під-маршрут у правій частині графіку. Цей метод працює досить нестабільно з очевидних причин. Але для невеликої кількості точок і великій кількості циклів є висока вірогідність дійсно згенерувати оптимальний розв’язок.

РОЗДІЛ 4. ОФОРМЛЕННЯ БІБЛІОТЕКИ

Розділ 4.1. Викладення в PyPI

PyPI – потужний каталог python-пакетів, дає можливість знаходити, завантажувати та публікувати пакети. Саме на ньому ми викладемо кінцеву версію бібліотеки. Виглядатиме наступним чином:

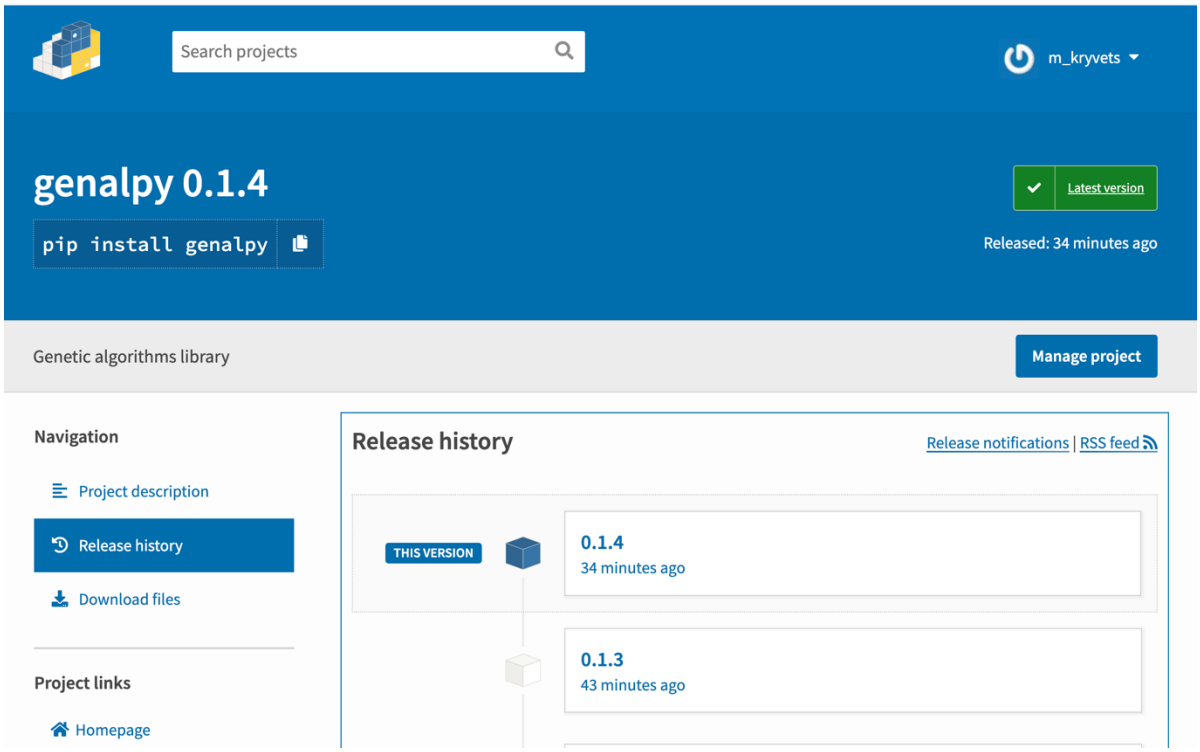
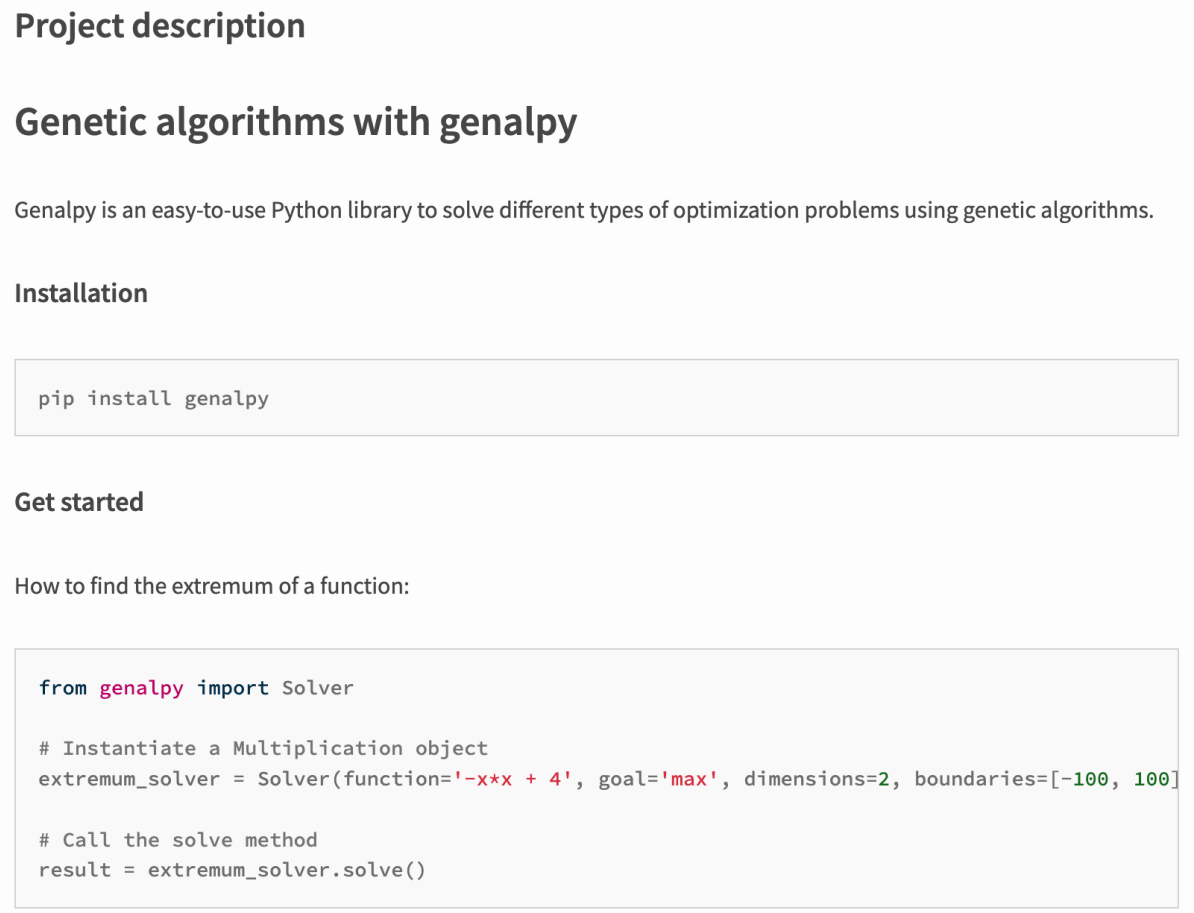


Рис. 4.1. Остання версія нашого пакету genalpy

Тепер будь-який користувач зможе завантажити наш пакет, якщо введе команду в консолі “pip install genalpy”. Але є деякі вимоги:

- встановлений python 3.6 версії і вище
- бібліотека numpy
- бібліотека pandas
- встановлений система для управління бібліотеками та залежностями в python – pip

Також додали файл Readme.md з коротким описом проекту:



Project description

Genetic algorithms with genalpy

Genalpy is an easy-to-use Python library to solve different types of optimization problems using genetic algorithms.

Installation

```
pip install genalpy
```

Get started

How to find the extremum of a function:

```
from genalpy import Solver

# Instantiate a Multiplication object
extremum_solver = Solver(function='-x*x + 4', goal='max', dimensions=2, boundaries=[-100, 100]

# Call the solve method
result = extremum_solver.solve()
```

Рис. 4.2. Опис проекту на головній сторінці

Посилання на дану сторінку, а також на GitHub проекту та документацію, описану нижче, можна знайти в Додатку В.

Розділ 4.2. Створення документації

Окрім цього, було автоматизовано оновлення документації через GitHub. Вона має вкладену структуру на два рівні:

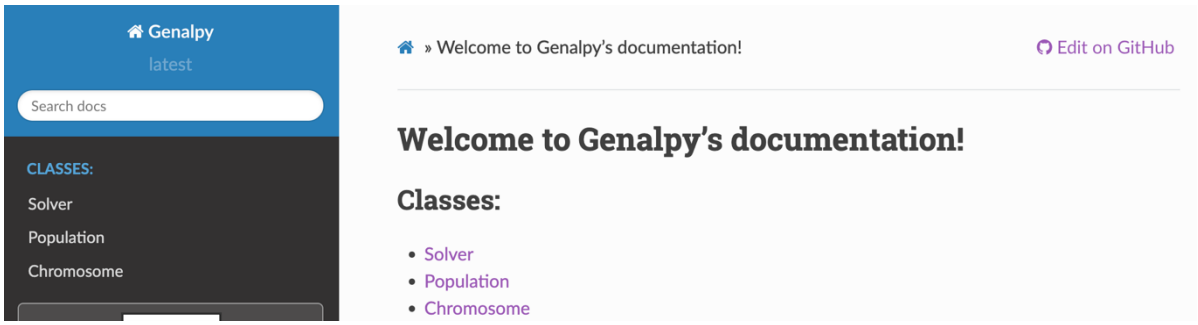


Рис. 4.3. Головна сторінка сайту з документацією genalpy

ВИСНОВКИ

У даній роботі було досліджено генетичний алгоритм - один з найбільш перспективних та потужних пошукових алгоритмів для розв’язку оптимізаційних задач.

Головною метою було створення саме універсального інструменту для вирішення різноманітних класів оптимізаційних задач – тобто розглядались не конкретні типи завдань і найкращі для них підходи, а будь-які модифікації та методи алгоритму загалом. Такий підхід дав можливість для реалізації рішень двох абсолютно різних за змістом та вхідними даними задач – пошуку екстремуму функції (в площині та в просторі) та задачі комівояжера. Таким чином, можна бути впевненими в тому, що розширення бібліотеки у майбутньому не створить жодних проблем – достатньо буде прописати логіку внесення вхідних даних в алгоритм та при потребі створити функції для нових методів (наприклад, для селекції чи мутації).

При роботі алгоритму на різних задачах було помічено як його переваги, так і недоліки. По-перше, він досить гнучкий і має багато вільних параметрів - це дає простір для експериментування та створення евристик. По-друге, генетичний алгоритм можна зручно розпаралелити на різних етапах, наприклад, при схрещуваннях та мутаціях різних частин популяції. Також, він ефективно справляється з досить складними цільовими функціями, на відміну від своїх конкурентів. Але в той же час, треба бути дуже обережним з обранням методів селекції, схрещування, мутації та рекомбінації, оскільки є ризик передчасної збіжності, наприклад, фокусування алгоритму навколо одного з локальних екстремумів. Треба також розуміти, що якщо цільова функція не складна, то краще обрати більш простий алгоритм пошуку – це буде вигідніше по часу.

Було розглянуто майже всі доступні на сьогоднішній день модифікації алгоритму, виявлені їх особливості та задачі, з якими вони

найкраще справляються. Для етапу створення початкової популяції було ретельно досліджено та реалізовано стратегії «рушниці» та фокусування. Для селекції – методи ранжування, імітації відпалу та пропорційний. Для рекомбінації (кросинговеру, схрещування) було розглянуто одно- та багато-точковий методи, однорідний, дискретний, проміжний, лінійний. Для двійкової мутації було реалізовано класичний та інверсійний методи, також окремо дослідили мутацію дійсних чисел. Щодо рекомбінації – порівняли чисту заміну, елітарну схему, рівномірну випадкову заміну, пропорційну та селективну схему.

Для порівняння принципів та результатів роботи генетичного алгоритму з іншими, також було реалізовано наступні пошукові алгоритми:

- імітації відпалу (для пошуку екстремуму функції);
- жадібний (для задачі комівояжера);
- випадковий пошук (для обох задач).

Після порівняння прийшли до висновку, що при однакових умовах (кількість циклів та час) генетичний алгоритм показує значно кращі результати. При цьому цільові функції в обох задачах були досить простими. Якщо ж їх ускладнити, то розрив між результатами роботи буде більш помітним.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. List of genetic algorithm applications [Електронний ресурс]: Natural Science, Mathematics, Computer Science - режим доступу до ресурсу: https://en.wikipedia.org/wiki/List_of_genetic_algorithm_applications
2. Goldberg D. E., Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, reading, 1989 – с. 25
3. Еволюційні обчислення [Електронний ресурс]: Лекція 3. Модифікації генетичних алгоритмів - режим доступу до ресурсу: https://intuit.ru/studies/professional_skill_improvements/14221/courses/1284/lecture/24172?page=1
4. The Computer Science of evolution [Електронний ресурс]: An introduction to genetic algorithms - режим доступу до ресурсу: https://www.freecodecamp.org/news/the-computer-science-of-evolution-an-introduction-to-genetic-algorithms-b3871286c7e7/
5. Simulated Annealing Evolution [Електронний ресурс]: Figure 5. Cooling: three ANNs learning - режим доступу до ресурсу: https://www.intechopen.com/chapters/38520
6. G. Gutin, A.P. Punnen, The Traveling Salesman Problem and Its Variations, Springer Science & Business Media, 2002 – с. 44

ДОДАТОК А

Код проекту мовою python:

- Клас Solver

```
import math
import numpy as np
from genalpy.utils import total_distance, stop_condition
from genalpy.population import Population

class Solver:
    """
    Instantiates a solving operator for optimization problems.
    Uses the selected methods of Genetic Algorithm.

    :param task_type: problem to solve, available - func_extremum, TSP
    :type task_type: string

    :param create_population_method: method of creating initial population, available -
        shotgun, focusing (additional parameter - focus_boundaries)
    :type create_population_method: string

    :param selection_method: method of parent selection, available - proportional, ranging, annealing
    :type selection_method: string

    :param recombination_method: method of recombination, available - discrete, transitional, linear
    :type recombination_method: string

    :param mutation_method: method of mutation, available - real_number
    :type mutation_method: string

    :param reduction_method: method of reduction, available - uniform_random_replacement, selective_scheme
    :type reduction_method: string

    :param stop_criterion: condition to stop the cycle, available - generations_limit, fitness_stability
    :type stop_criterion: string
```

```
    :param stop_criterion_value: implication depends on stop_criterion:
            - 'generations_limit' -> max number of generations
            - 'fitness_stability' -> max difference between the best fitness values in the last generations
    :type stop_criterion_value: int/float

    :param n_pop: number of elements in population, default - 1000
    :type n_pop: int

    In case of task_type=='func_extremum':

        :param function: represented as a string, examples:
            - 2dim: 'x**2 + x - 1'
            - 3dim: 'x*y - 2*(np.cos(y)+1)'
        :type function: string

        :param goal: max or min, default - min
        :type goal: string

        :param dimensions: number of dimensions (currently available 2 or 3), default - 3
        :type dimensions: int

        :param boundaries: min and max possible values of variable coordinates:
            - 2dim: [Xmin, Xmax]
            - 3dim: [Xmin, Xmax, Ymin, Ymax]
        :type boundaries: list of floats

        :param focus_boundaries: parameter for create_population_method=='focusing':
            - 2dim: [Xmin, Xmax]
            - 3dim: [Xmin, Xmax, Ymin, Ymax]
        :type focus_boundaries: list of floats

    In case of task_type=='TSP':
```

```
    :param points: cities to visit, the first city must be the starting
        and simultaneously the ending point of the route
    :type points: list of 2-element lists [float, float], X and Y coordinates respectively
    """
    def __init__(self, task_type='func_extremum', create_population_method='shotgun',
                 selection_method='ranging', recombination_method='transitional', mutation_method='real_number',
                 reduction_method='selective_scheme', stop_criterion='generations_limit', stop_value=20,
                 n_pop=50, function=None, goal='min', dimensions=3,
                 boundaries=None, focus_boundaries=None, points=None):
        try:
            self.task_type = task_type
            self.create_population_method = create_population_method
            self.selection_method = selection_method
            self.recombination_method = recombination_method
            self.mutation_method = mutation_method
            self.reduction_method = reduction_method
            self.stop_criterion = stop_criterion
            self.stop_value = stop_value
            self.n_pop = n_pop
            self.goal = goal
            self.dimensions = dimensions
            self.boundaries = boundaries
            self.focus_boundaries = focus_boundaries
            self.points = points
            # definition of the fitness_func
            if task_type == 'func_extremum':
                if self.dimensions == 2:
                    self.transitional_func = lambda x: eval(function)
                    self.fitness_func = lambda coords: self.transitional_func(coords[0])
                elif self.dimensions == 3:
                    self.transitional_func = lambda x, y: eval(function)
                    self.fitness_func = lambda coords: self.transitional_func(coords[0], coords[1])
```

```
                else:
                    print("Change the number of dimensions to the following: 2/3")
            elif task_type == 'TSP':
                self.fitness_func = lambda p: total_distance(p)
            else:
                print('There is no solution for your task... Try the following: func_extremum/TSP')
        except:
            print('Check the available parameters')

    def solve(self):
        """
        Solves the optimization problems based on parameters using Genetic Algorithm.

        :return: The optimum solution.
        :rtype: float
        """
        population = Population(self.task_type, self.n_pop, self.fitness_func, self.goal, self.dimensions,
                                self.boundaries, self.points)
        population.create(self.create_population_method, self.focus_boundaries)
        generation_num = 0
        accuracy = 1
        # temperature for simulated annealing
        t = 20.0
        while not stop_condition(self.stop_criterion, accuracy, generation_num, self.stop_value):
            # best fitness of the previous population
            old_best_fitness = population.best_fitness()
            # clean parents, children, mutation lists for the new population
            population.clean()
            # select even number of parents for recombination
            population.selection(self.selection_method, t)
            # lower the temperature for the next cycle
            if self.selection_method == 'annealing':
                t *= 0.5
            # create children
```

```
            population.recombination(self.recombination_method)
            # implement random mutations
            population.mutation(self.mutation_method, generation_num)
            # reducing the number of individuals to n_pop
            population.reduction(self.reduction_method)
            # calculate the best fitness of current population
            new_best_fitness = population.best_fitness()
            # compare results for possible cycle condition
            accuracy = abs(new_best_fitness - old_best_fitness)
            generation_num += 1
        print("The optimum value is:", population.final_chromosome().fitness_value)
        print("It is reached at the point:", population.final_chromosome().gene_list)
        return population.final_chromosome().fitness_value
```

- Клас Chromosome

```
class Chromosome:
    """
    Instantiates the chromosome - an individual's set of genes.

    :param gene_list: set of genes (mostly floats)
    :type gene_list: list

    :param fitness_func: lambda function to maximize/minimize
    :type fitness_func: function
    """
    def __init__(self, gene_list, fitness_func):
        self.gene_list = gene_list
        self.fitness_func = fitness_func
        # calculate fitness_value of the individual
        self.fitness_value = self.fitness_func(self.gene_list)

    def get_info(self):
        """
        Returns list of genes and the value of chromosome's fitness function
        """
        return self.gene_list, self.fitness_value
```

- Клас Population

```python
import numpy as np
import pandas as pd
from genalpy.utils import mix_points, ranging, euler, simulated_annealing
from genalpy.chromosome import Chromosome
import random


class Population:
    """
    Instantiates the population - set of individuals. Each individual has one chromosome.

    :param chromosome_list: current individuals
    :type chromosome_list: list of objects (Chromosomes)
    :param parents_list: selected parents
    :type parents_list: list of objects (Chromosomes)
    :param children_list: non-mutated children
    :type children_list: list of objects (Chromosomes)
    :param mutation_list: mutated children
    :type mutation_list: list of objects (Chromosomes)
    """
    def __init__(self, task_type, n_pop, fitness_func, goal, dimensions,
                 boundaries, points):
        self.task_type = task_type
        self.n_pop = n_pop
        self.fitness_func = fitness_func
        self.goal = goal
        self.dimensions = dimensions
        self.boundaries = boundaries
        self.points = points
        self.chromosome_list = []
        self.parents_list = []
        self.children_list = []
        self.mutation_list = []
```

```python
    def clean(self):
        """
        Prepare population to the next step (empty parents, children, mutation lists).
        """
        self.parents_list = []
        self.children_list = []
        self.mutation_list = []

    def create(self, create_population_method, focus_boundaries):
        """
        Generates random chromosomes and fills the first population with them.
        """
        if create_population_method == 'shotgun':
            bound = self.boundaries
        elif create_population_method == 'focusing':
            bound = focus_boundaries
        else:
            print('Try available methods')
        for i in range(self.n_pop):
            if self.task_type == 'func_extremum':
                if self.dimensions == 2:
                    new_chromosome = Chromosome([np.random.uniform(bound[0], bound[1])],
                                                self.fitness_func)
                elif self.dimensions == 3:
                    new_chromosome = Chromosome([np.random.uniform(bound[0], bound[1]),
                                                 np.random.uniform(bound[2], bound[3])],
                                                self.fitness_func)
            elif self.task_type == 'TSP':
                new_chromosome = Chromosome(mix_points(self.points), self.fitness_func)
            self.chromosome_list.append(new_chromosome)
```

```python
def selection(self, selection_method, t):
    """
    Selects mostly the best individuals for recombination and puts them in parents_list.
    Number of parents is the half of n_pop.
    Proportional method is used only for maximization and positive fitness values.
    Ranging and simulated annealing for both max & min, and any fitness values.

    :param t: temperature for the simulated annealing
    :param t: float
    """
    # create DataFrame with chromosomes and their fitness_values to operate
    df_proportions = pd.DataFrame({'chromosome': self.chromosome_list,
                                   'fitness': [ch.fitness_value for ch in self.chromosome_list]})
    if selection_method == 'proportional':
        sum_fitness = sum(df_proportions['fitness'])
        # get the proportional fitness -> probability
        df_proportions['probability'] = df_proportions.apply((lambda row: (row['fitness'] / sum_fitness)), axis=1)
    elif selection_method == 'ranging':
        df_proportions.sort_values(by='fitness', ascending=False, inplace=True)
        # get the i - index
        df_proportions.reset_index(drop=False, inplace=True)
        df_proportions['probability'] = df_proportions.apply(lambda row: (ranging(row['index'] + 1, self.n_pop)),
                                                             axis=1)
        # get rid of extra column
        df_proportions.drop(columns='index', inplace=True)
    elif selection_method == 'annealing':
        df_proportions['euler'] = df_proportions.apply(lambda row: (euler(row['fitness'], t)),
                                                       axis=1)
        euler_avg = float(df_proportions['euler'].mean())
        df_proportions['probability'] = df_proportions.apply(lambda row: (simulated_annealing(euler_avg, row['euler'],
                                                                                              self.n_pop)),
                                                             axis=1)
```

```python
    # sort and cumsum the probabilities to 'place' them on the line segment
    df_proportions.sort_values(by='probability', ascending=False, inplace=True)
    df_proportions['probability_cumulative'] = df_proportions['probability'].cumsum()
    df_proportions.reset_index(drop=True, inplace=True)
    # get the start of each segment (it will be the end of the previous segment)
    probs_offset = list(df_proportions['probability_cumulative'])
    probs_offset = [0.0] + probs_offset[:-1]
    df_proportions['probability_lower_limit'] = probs_offset
    # generate numbers on the line [0;1] and sort them asc
    random_list = [np.random.uniform(0, 1) for i in range(int(self.n_pop / 2))]
    random_list.sort()
    # get the parent based on generated number
    for num in random_list:
        # num must be between chromosomes boundaries
        df_proportions['parent'] = np.where((df_proportions['probability_lower_limit'] <= num) &
                                            (df_proportions['probability_cumulative'] > num), True, False)
        # add parent chromosome to the list
        try:
            parent_ch = df_proportions[df_proportions.parent].reset_index(drop=True)['chromosome'].iloc[0]
            self.parents_list.append(parent_ch)
        except:
            pass
    # drop duplicates
    self.parents_list = list(set(self.parents_list))
    # parents must form pairs -> get even number of them
    if len(self.parents_list) % 2 != 0:
        self.parents_list = self.parents_list[:-1]
```

```python
def recombination(self, recombination_method):
    """
    Generates new individuals based on chosen parents and puts them in children_list.
    Discrete - get the genes from each parent based on mask.
    """
    num_of_parents = len(self.parents_list)
    pairs_list = []
    selected_parents_list = []
    while len(selected_parents_list) < num_of_parents:
        # get parents from non-selected
        first_parent = random.choice([parent for parent in self.parents_list if parent not in selected_parents_list])
        selected_parents_list.append(first_parent)
        second_parent = random.choice([parent for parent in self.parents_list if parent not in selected_parents_list])
        selected_parents_list.append(second_parent)
        # create and add pair to list
        pair = [first_parent, second_parent]
        pairs_list.append(pair)
    chromosome_len = len(self.chromosome_list[0].gene_list)
    if recombination_method == 'discrete':
        for parent_pair in pairs_list:
            chromosome_genes = []
            for i in range(chromosome_len):
                # choose the gene, 0 - first parent, 1 - second parent
                mask = random.randint(0, 1)
                if mask == 0:
                    chromosome_genes.append(parent_pair[0].gene_list[i])
                else:
                    chromosome_genes.append(parent_pair[1].gene_list[i])
            self.children_list.append(Chromosome(chromosome_genes, self.fitness_func))
    elif (recombination_method == 'transitional') or (recombination_method == 'linear'):
        for parent_pair in pairs_list:
            chromosome_genes_first = []
            chromosome_genes_second = []
            if recombination_method == 'linear':
                beta1 = random.uniform(0, 1.25)
                beta2 = random.uniform(0, 1.25)
            for i in range(chromosome_len):
                if recombination_method == 'transitional':
                    beta1 = random.uniform(0, 1.25)
                    beta2 = random.uniform(0, 1.25)
                p1 = parent_pair[0].gene_list[i]
                p2 = parent_pair[1].gene_list[i]
                chromosome_genes_first.append(p1 + beta1 * (p2 - p1))
                chromosome_genes_second.append(p1 + beta2 * (p2 - p1))
            self.children_list.append(Chromosome(chromosome_genes_first, self.fitness_func))
            self.children_list.append(Chromosome(chromosome_genes_second, self.fitness_func))

def mutation(self, mutation_method, generation_num):
    """
    Randomly picks up some individuals from children_list and implements the certain type of gene mutation.
    """
    mutation_probability = 0.1 - (1 / (self.n_pop * 10)) * generation_num
    if mutation_method == 'real_number':
        for child in self.children_list:
            mutate = (random.random() <= mutation_probability)
            if mutate:
                self.children_list.remove(child)
                chromosome_len = len(child.gene_list)
                chromosome_genes = []
                for i in range(chromosome_len):
                    modified_gene = random.uniform(self.boundaries[i*2], self.boundaries[i*2 + 1])
                    chromosome_genes.append(modified_gene)
                self.mutation_list.append(Chromosome(chromosome_genes, self.fitness_func))
```

```
def reduction(self, reduction_method):
    """
    Chooses some individuals and deletes them from the current population.
    """
    # select the rest part of individuals (not parents)
    population_remainder = [ch for ch in self.chromosome_list if ch not in self.parents_list]
    all_population = population_remainder + self.parents_list + self.children_list + self.mutation_list
    if reduction_method == 'selective_scheme':
        df_population = pd.DataFrame({'chromosome': all_population,
                                      'fitness_value': [ch.fitness_value for ch in all_population]})
        if self.goal == 'min':
            df_population.sort_values(by='fitness_value', ascending=True, inplace=True)
        elif self.goal == 'max':
            df_population.sort_values(by='fitness_value', ascending=False, inplace=True)
        df_population.reset_index(drop=True, inplace=True)
        df_population = df_population[:self.n_pop]
        clean_population = list(df_population['chromosome'])
    elif reduction_method == 'uniform_random_replacement':
        num_to_delete = len(all_population) - self.n_pop
        items_to_delete = set(random.sample(range(len(all_population)), num_to_delete))
        clean_population = [x for i, x in enumerate(all_population) if not i in items_to_delete]
    self.chromosome_list = clean_population

def best_fitness(self):
    """
    Get the best fitness value among all current chromosomes.

    :return: float
    """
    # in case of the first generation - empty list
    if not self.chromosome_list:
        return 0.0
    fitness_values_list = [ch.fitness_value for ch in self.chromosome_list]
    if self.goal == 'min':
        return min(fitness_values_list)
    elif self.goal == 'max':
        return max(fitness_values_list)


def final_chromosome(self):
    """
    Get the result chromosome with the optimum fitness.

    :return: Chromosome
    """
    if not self.chromosome_list:
        return None
    df_final_population = pd.DataFrame({'chromosome': self.chromosome_list,
                                        'fitness_value': [ch.fitness_value for ch in self.chromosome_list]})
    if self.goal == 'min':
        df_final_population.sort_values(by='fitness_value', inplace=True, ascending=True)
    elif self.goal == 'max':
        df_final_population.sort_values(by='fitness_value', inplace=True, ascending=False)
    df_final_population.reset_index(drop=True, inplace=True)
    return df_final_population['chromosome'].iloc[0]
```

- utils.py (other functions)

```python
import random
import math

def stop_condition(stop_criterion, accuracy, generation_num, stop_value):
    """
    Based on stop_criterion, creates and checks the condition for the main cycle.

    :type stop_criterion: string
    :type accuracy: float
    :type generation_num: int
    :type stop_value: float
    :return: bool
    """
    if stop_criterion == 'generations_limit':
        return generation_num >= stop_value
    elif stop_criterion == 'fitness_stability':
        return accuracy < stop_value

def distance(p1, p2):
    """
    Returns distance between 2 points.

    :param p1: point [X1;Y1]
    :type p1: list of floats
    :param p2: point [X2;Y2]
    :type p2: list of floats
    :return: distance in float
    """
    x_diff = p2[0] - p1[0]
    y_diff = p2[1] - p1[2]
    return math.sqrt(x_diff**2 + y_diff**2)
```

```python
def total_distance(points):
    """
    Returns the route length, where route is determined by the order of points in a given list.
    The first point is also the last one.

    :param points: coordinates [X;Y] of cities to visit
    :type points: list of lists [float; float]
    :return: distance in float
    """
    n = len(points)
    res = 0
    for i in range(n - 1):
        d = distance(points[i], points[i+1])
        res += d
    res += distance(points[n-1], points[0])
    return res
```

```python
def mix_points(points):
    """
    Changes the order of points (except the first point).

    :param points: coordinates [X;Y] of cities to visit
    :type points: list of lists [float; float]
    :return: list of lists [float; float]
    """
    res_list = [points[0]]
    elements_to_shuffle = points[1:]
    random.shuffle(elements_to_shuffle)
    for i in range(len(points) - 1):
        res_list.append(elements_to_shuffle[i])
    return res_list

def ranging(i, N):
    """
    Formula for the ranging selection to get the probability for each individual.

    :param i: index (starting from 1) of the individual in sorted df (by fitness)
    :type i: int
    :param N: number of individuals in population
    :type N: int
    :return: float
    """
    a = random.uniform(1, 2)
    b = 2 - a
    return (1 / N) * (a - (a - b) * ((i - 1) / (N - 1)))
```

```python
def euler(fitness_value, T):
    """
    Part of the simulated annealing selection formula.

    :param fitness_value: individual's fitness value
    :type fitness_value: float
    :param T: temperature
    :type T: float
    :return: float
    """
    return pow(math.e, (fitness_value / T))

def simulated_annealing(euler_avg, euler_current, N):
    """
    Formula for the simulated annealing selection to get the probability for each individual.

    :param euler_avg: average value of euler function for the population
    :type euler_avg: float
    :param euler_current: individual's current value of euler function
    :type euler_current: float
    :param N: number of individuals in population
    :type N: int
    :return: float
    """
    return (1 / N) * (euler_current / euler_avg)
```

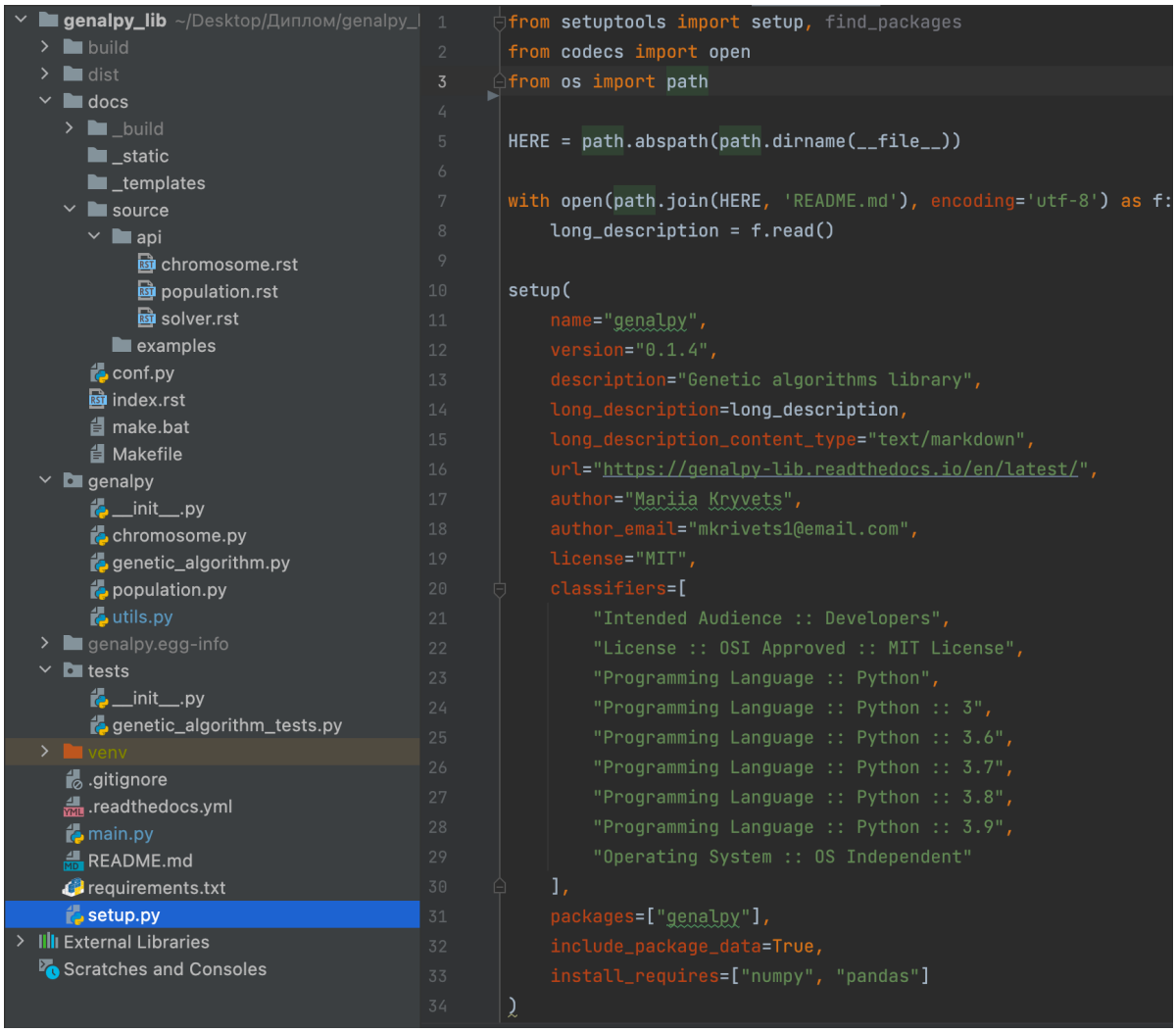
- main.py

```python
from genalpy import Solver


solver_3d_min = Solver(task_type='func_extremum', create_population_method='shotgun',
                       selection_method='ranging', recombination_method='transitional', mutation_method='real_number',
                       reduction_method='selective_scheme', stop_criterion='generations_limit', stop_value=20,
                       n_pop=100, function='-x * (np.sin(np.sqrt(abs(x)))) + y * np.cos(np.sqrt(abs(y)))',
                       goal='min', dimensions=3, boundaries=[-30, 30, -10, 10])
solver_3d_min.solve()

solver_2d_max = Solver(task_type='func_extremum', create_population_method='shotgun',
                       selection_method='ranging', recombination_method='transitional', mutation_method='real_number',
                       reduction_method='selective_scheme', stop_criterion='generations_limit', stop_value=20,
                       n_pop=50, function='-x*x + 4',
                       goal='max', dimensions=2, boundaries=[-100, 100])
solver_2d_max.solve()
```

- setup.py and project structure



ДОДАТОК Б

Створення візуалізації для функції у прикладі:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')

X = np.arange(-30, 30, 0.01)
Y = np.arange(-10, 10, 0.01)
X, Y = np.meshgrid(X, Y)
Z = -X * (np.sin(np.sqrt(abs(X)))) + Y * np.cos(np.sqrt(abs(Y)))

surface = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, linewidth=0, antialiased=False)

ax.set_zlim(-30, 30)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

fig.colorbar(surface, shrink=0.5, aspect=5)

plt.savefig('vis.png')
plt.show()
```

ДОДАТОК В

Посилання:

- на github проекту: https://github.com/mkryvets/genalpy_lib
- на бібліотеку в PyPi: https://pypi.org/project/genalpy/
- на документацію: https://genalpy-lib.readthedocs.io/en/latest/