ЛАБОРАТОРНА РОБОТА № 7

ДОСЛІДЖЕННЯ ГЕНЕТИЧНИХ ТА МУРАШИНИХ АЛГОРИТМІВ

Мета роботи: *використовуючи спеціалізовані бібліотеки та мову програмування Python дослідити генетичні та мурашині алгоритми*.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

Основні теоретичні відомості подані на лекціях. Також доцільно вивчити матеріал поданий в літературі:

Alberto Artasanchez, Prateek Joshi. Artifcial Intelligence with Python. Second Edition. BIRMINGHAM – MUMBAI: Packt Publishing 2020. – 592 p. ISBN 978-1-83921-953-5.

Додатково деякі теоретичні відомості подано у кожному завданні окремо.

Можна використовувати Google Colab або Jupiter Notebook.

Випадки використання генетичного програмування

Як обговорювалося на лекціях, Генетичні алгоритми (GA) і Генетичне Програмування (GP) є одним із «п’яти племен» машинного навчання.

З самого початку GP створив широкий спектр досягнень. Література, яка охоплює тисячі застосувань GP, містить багато випадків використання, де GP було успішно застосовано. Висвітлення цього списку виходило б за рамки роботи, але ми перерахуємо тут кілька найважливіших.

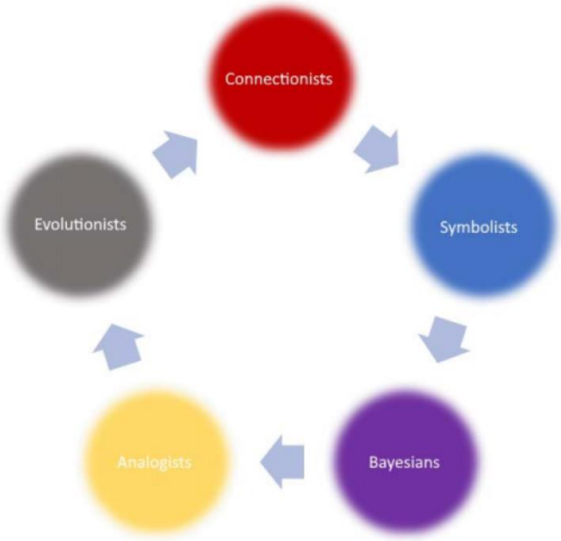


Рис. 1. П'ять племен машинного навчання (Педро Домінгос)

Погано зрозумілі області.

Тут взаємозв'язки між відповідними змінними невідомі або погано зрозумілі (або там, де є підозри, що поточне розуміння може бути неправильним). Однією з переваг GP (та інших еволюційних алгоритмів) є дослідження маловивчених областей. Якщо область проблеми добре зрозуміла, існують інші аналітичні інструменти та методи, які можуть забезпечити якісні рішення без невизначеності, властивої процесу стохастичного пошуку GP.

GP, з іншого боку, дає результати, коли область недостатньо вивчена та зрозуміла. Фахівець загальної практики може допомогти визначити, які атрибути та параметри є релевантними, надати нові та творчі рішення, виявити несподівані зв'язки серед атрибутів і відкрити нові концепції, які потім можна застосувати в інших областях.

Знаходження розміру та форми кінцевого рішення є основною частиною проблеми. Якщо форма рішення відома, то альтернативні механізми пошуку, які працюють із представленнями фіксованого розміру (наприклад, GA), можуть бути більш ефективними, оскільки їм не доведеться виявляти розмір і форму рішення.

Дані доступні та їх багато

GP зокрема, а також методи машинного навчання та пошуку в цілому зазвичай потребують величезних обсягів тестових даних для виконання. Пошук релевантних наборів даних для проблеми може бути великою перешкодою. Але у випадках, коли великі набори даних є легкодоступними, це може бути мрією спеціаліста з даних, і було б гарною ідеєю придумати запитання, які можна поставити просто тому, що дані доступні.

Також корисно, якщо дані тесту будуть максимально чистими та точними. Однак алгоритми GP можуть витончено впоратися з певною кількістю шуму в даних, особливо якщо вжито заходів для мінімізації надмірного розподілу.

Припустимі наближені рішення

GP добре працює в ситуаціях, коли приблизне рішення є прийнятним або приблизне рішення є найкращим із можливих. Еволюція загалом, і GP зокрема, зазвичай полягає в тому, щоб бути «достатньо хорошим», а не «найкращим». Якщо ведмідь женеться за тобою по лісі, тобі не обов'язково бути найшвидшою людиною в світі. Вам просто потрібно бути швидшим за ведмедя або за людину, що біжить поруч. Як наслідок, еволюційні алгоритми, як правило, найкраще працюють у областях, де можливі та прийнятні близькі наближення.

Невеликі, але високо цінні покращення

Було виявлено, що GP добре працює в областях, де технологічні зусилля, як правило, зосереджені в областях високого економічного значення. У цих сферах попередні дослідники, ймовірно, витратили значну кількість часу та зусиль, і «сучасний рівень» має тенденцію бути передовим. У цьому випадку складно вдосконалити існуючі рішення. Однак у цих самих областях

невеликі покращення можуть бути надзвичайно цінними. У подібних випадках фахівець загальної практики іноді може зробити невеликий, але цінний і прибутковий внесок. Прикладами можуть бути розвідка нафти, управління матеріалами та фінансові програми.

Тепер давайте розглянемо деякі галузі застосування GA та GP:

Фільми

Як і деякі інші професії, кінокаскадери скоро можуть бути не потрібні. Стартап під назвою NaturalMotion використовує GP з неймовірно реалістичними результатами для створення людей у русі. Ці віртуальні актори падають, стрибають і виконують інші трюки з реальною точністю. Ці віртуальні актори можуть реагувати, як справжні люди, на силу, яка на них діє, і демонструвати різноманітні реалістичні рухи. Для цього всього потрібна лише потужність настільного ПК. Фільми - це лише початок. У найближчі кілька років NaturalMotion планує розкрити ці реалістичні фігури в наступному поколінні відеоігор.

NaturalMotion — це нова компанія, заснована колишніми оксфордськими дослідниками Торстеном Рейлом і Колмом Массі. Зараз компанія має лише один продукт під назвою Endorphin, який використовує нейронні мережі та штучну еволюцію для створення програмного забезпечення та автоматів, які можуть ходити, бігати, падати та літати з людською точністю.

Ендорфін дебютував у повнометражному фільмі «Повернення короля», де він використовувався, щоб втілити в життя особливо складний трюк. Але це був лише початок. Через кілька місяців роботи компанії билися насмерть кинджали на рівнини Іліуму, у фільмі Вольфганга Петерсена «Троя».

Комп'ютерні ігри

Сьогодні всі закохані в алгоритми глибокого навчання. Вони, безсумнівно, показали вражаючі результати в багатьох сферах і порівняно з багатьма тестами. Але GP розвивається. Завдяки роботі Денніса Вілсона та кількох його колег з Університету Тулузи у Франції було досягнуто вражаючих результатів. Їхня робота над GP змогла перевершити людей у багатьох класичних іграх.

Вілсон і його команда дослідників показали, як GP може зрівнятися з продуктивністю алгоритмів глибокого навчання в знаковому завданні, яке принесло глибокому навчанню популярність у 2013 році – здатність перевершити людей у аркадних відеоіграх, таких як Pong, Breakout і Space Invaders. Вілсон переконливо продемонстрував, що GP може давати вражаючі результати, які можна порівняти з глибоким навчанням і, можливо, краще.

Стиснення файлів

Один із перших методів стиснення без втрат використовував нелінійні предиктори GP для зображень. Алгоритм передбачав рівень сірого, який може прийняти піксель, на основі значень підмножини сірого сусідніх пікселів. Помилки передбачення в поєднанні з описом моделі можуть представляти стиснуту версію зображення. Зображення були стиснуті за допомогою кодування Хаффмана. Результати на широкому колі

різноманітних зображень виявилися багатообіцяючими. У деяких випадках алгоритми GP перевершують деякі з найкращих, розроблених людиною алгоритми стиснення без втрат.

Фінансовий трейдинг

Гіпотеза ефективних ринків є фундаментальним принципом економіки. Вона заснована на ідеї, що кожен учасник ринку володіє досконалою інформацією і діє раціонально. Якщо гіпотеза про ефективні ринки вірна, кожен повинен призначити однакову ціну всім активам на ринку та домовитися про ціну. Якби різниці в ціні не існувало, не було б способу перемогти ринок. Незалежно від того, чи це товарний, валютний чи фондовий ринок, немає рівних учасників ринку, і існують значні сумніви в тому, що ефективні ринки справді існують. Чим більш неліквідним є ринок, тим менш ефективним він буде. Отже, люди продовжують досліджувати фондовий ринок і намагаються знайти способи перемогти це. Є кілька людей і компаній, які на основі свого послужного списку переконливо стверджують, що ринок можна перемогти. Деякі приклади:

• Уоррен Баффет і Berkshire Hathaway
• Пітер Лінч і фонд Fidelity Magellan
• Рей Даліо та Bridgewater Associates
• Джим Саймонс і Renaissance Technologies

Останні два приклади значною мірою покладаються на комп'ютерні алгоритми для досягнення ринкових результатів.

Теорія ігор була стандартним інструментом, який використовували економісти, щоб спробувати зрозуміти ринки, але вона все частіше доповнюється моделюванням як з людьми, так і з комп'ютеризованими агентами. GP все частіше використовується як частина цього моделювання при моделюванні соціальних систем.

Інші застосування

Оптимізація – GA та GP зазвичай використовуються в задачах оптимізації, де значення необхідно максимізувати або мінімізувати, враховуючи цільову функцію в наборі обмежень.

Паралелізація – GA також мають можливості паралельної обробки і виявилися ефективним методом вирішення проблем, які потребують паралельної обробки. Розпаралелювання є активною сферою досліджень у GA та GP.

Нейронні мережі − GA використовуються для навчання нейронних мереж, зокрема рекурентних нейронних мереж (RNN).

Економіка − GA зазвичай використовуються для моделювання економічних систем, таких як:

• Модель павутинки
• Рівноважний дозвіл теорії ігор
• Ціноутворення активів

Обробка зображень – GA також використовуються для різноманітних завдань обробки цифрових зображень (DIP), таких як щільне зіставлення пікселів.

Програми планування – GA можна використовувати для вирішення багатьох проблем планування, зокрема проблеми розкладу. Простіше кажучи, проблема розкладу виникає, коли у нас є набір ресурсів, набір дій і набір залежностей між діяльністю та ресурсами. Прикладом є розклад занять в університеті, де у нас є аудиторії, викладачі та студенти, а в кінці завдання, сподіваюся, що великий відсоток студентів зможе пройти всі заняття, які вони хочуть.

Параметричне проектування – GA використовувалися для проектування транспортних засобів, механізмів і літаків шляхом зміни параметрів і розробки кращих рішень.

Аналіз ДНК – ГА можуть використовуватися і використовувалися для визначення структур ДНК за допомогою спектрометричних даних зразків.

Мультимодальна оптимізація – GA є чудовим підходом для вирішення проблем мультимодальної оптимізації, коли шукаються кілька оптимальних рішень.

Проблема комівояжера (TSP) – GA використовувалися для вирішення TSP і всіх пов'язаних із нею додатків, таких як проблеми маршрутизації транспортного засобу та траєкторії роботів, що є добре відомою комбінаторною проблемою з використанням нових стратегій кросовера та упаковки.

Сподіваюся, вам зрозумілі широкі та різноманітні застосування GP та GA. Можливо, вам вдасться придумати власні унікальні програми та використовувати отримані знання для просування вперед.

Еволюційні та генетичні алгоритми

Автономне вирішення задач комп'ютером є центральною метою штучного інтелекту та машинного навчання. *Генетичні алгоритми (GA)* — це еволюційна техніка обчислень, яка автоматично розв'язує задачі, не вимагаючи від користувача заздалегідь знати чи вказувати форму або структуру рішення. На найбільш абстрактному рівні GA – це систематичний, незалежний від предметної області метод автоматичного розв'язання задач комп'ютерами, починаючи з високорівневої постановки задачі про те, що потрібно зробити.

Генетичний алгоритм є різновидом еволюційного алгоритму. Тож, щоб зрозуміти генетичні алгоритми, нам потрібно обговорити еволюційні алгоритми. Еволюційний алгоритм — це метаевристичний алгоритм оптимізації, який застосовує принципи еволюції для вирішення конкретних задач. Концепція еволюції схожа на ту, яку ми знаходимо в природі: подібно до того, як навколишнє середовище активно стимулює «рішення», отримане в результаті еволюції, ми безпосередньо використовуємо функції та змінні задачі, щоб прийти до оптимального рішення. Однак, у генетичному алгоритмі будь-яка задана задача буде закодована в бітових шаблонах, якими і маніпулює цей алгоритм.

Основні кроки в еволюційних алгоритмах такі:

Крок 1. Випадково згенеруйте початкову популяцію точок даних або індивідуумів. Індивідуум, як визначено GA, є членом популяції, який має певні характеристики чи риси. На наступних етапах алгоритму ми визначимо, чи дозволяють ці риси індивідууму адаптуватися до навколишнього середовища та виживати достатньо довго, щоб мати потомство.

Крок 2. Виконуйте ці кроки в циклі до припинення:

1. Оцініть придатність (fitness) кожного індивідууму в цій популяції.
2. Здійсніть відбір найбільш придатних для розмноження індивідуумів.
3. Виведіть нових індивідуумів шляхом схрещування та мутації для народження потомства.
4. Оцініть індивідуальну придатність нових індивідуумів.
5. Замініть менш придатну популяцію новими індивідуумами.

Придатність окремих осіб визначається за допомогою попередньо визначеної функції пристосованості (fitness function). Тут у гру вступає фраза «виживання найпристосованішого».

Потім ми беремо цих вибраних індивідуумів і створюємо наступне покоління індивідуумів шляхом *рекомбінації* та *мутації*. У наступному розділі ми обговоримо поняття рекомбінації та мутації. Наразі давайте думати про ці техніки як про механізми створити наступне покоління, ставлячись до обраних осіб як до батьків.

Виконавши рекомбінацію та мутацію, ми створюємо нову групу особин, які змагатимуться зі старими за місце в наступному поколінні. Відкидаючи найслабкіших особин і замінюючи їх нащадками, ми підвищуємо загальний рівень пристосованості популяції. Ми продовжуємо повторювати, доки не буде досягнуто бажаної загальної пристосованості.

GA — це еволюційний алгоритм, у якому ми використовуємо евристику, щоб знайти рядок бітів, який вирішує задачу. Ми постійно повторюємо цей алгоритм, щоб знайти найкраще рішення.

Ми робимо це, створюючи нові популяції, що містять більше пристосованих особин. Ми застосовуємо імовірнісні оператори, такі як *відбір*, *схрещування* та *мутація*, щоб створити наступне покоління індивідуумів. Індивідууми представлені у вигляді рядків, де кожен рядок є закодованою (у вигляді «1» та «0») версією потенційного рішення.

Функція пристосованості використовується для оцінки міри придатності кожного рядка, повідомляючи нам, наскільки добре він підходить для вирішення задачі, про яку йде мова. Цю функцію пристосованості також називають функцією оцінювання. GA застосовують оператори, натхненні природою, тому тут термінологія тісно пов’язана зі знайденими термінами в біології.

Фундаментальні поняття в генетичних алгоритмах

Щоб побудувати GA, нам потрібно зрозуміти кілька ключових понять і термінів.

Ці концепції широко використовуються в області GA для створення рішень для різних задач. Одним із найважливіших аспектів GA є випадковість. Для повторення він покладається на випадкову вибірку

індивідуумів. Це означає, що процес є недетермінованим. Отже, якщо ви запустите той самий алгоритм кілька разів, він може закінчитися різними рішеннями.

Давайте тепер дамо визначення терміну популяція. Популяція — це набір індивідуумів, які є можливими кандидатами на рішення. У GA єдине найкраще рішення підтримується не на будь-якій стадії, а скоріше набір потенційних рішень, одне з яких може бути найкращим. Але інші рішення відіграють важливу роль під час пошуку. Оскільки відстежується популяція рішень, менша ймовірність застрягти в локальному оптимумі. Застрягання в локальному оптимумі є класичною проблемою, з якою стикаються інші методи оптимізації.

Тепер, коли ми знаємо про популяції та стохастичну природу GA, давайте поговоримо про оператори. Створюючи наступне покоління індивідуумів, алгоритм намагається переконатися, що вони походять від найпридатніших індивідуумів поточного покоління.

Мутація є одним із шляхів досягнення цього. GA вносить випадкові зміни в один або кілька індивідуумів поточного покоління, щоб отримати нове рішення-кандидат. Ця зміна називається мутацією. Тепер ця зміна може зробити цього індивідуума кращим або гіршим за існуючих осіб.

Наступне поняття, яке необхідно визначити, це рекомбінація, яку також називають кросинговером (кросовером). Це безпосередньо пов'язано з роллю розмноження в процесі еволюції. GA намагається об’єднати індивідуумів із поточного покоління, щоб створити нове рішення. Він поєднує в собі деякі особливості кожного з батьківських індивідуумів для створення потомства. Цей процес називається кросовером. Мета полягає в тому, щоб замінити «менш придатних» особин у поточному поколінні потомством, отриманим від «більш придатних» особин у популяції.

Щоб застосувати кросинговер і мутацію, нам потрібні критерії відбору (селекції). Концепція селекції натхненна теорією природного відбору. Під час кожної ітерації GA виконує процес відбору. За допомогою цього процесу відбору вибираються найпридатніші особини, а слабші особини відкидаються. Ось де у гру вступає концепція виживання найсильнішого. Процес відбору здійснюється за допомогою функції, яка обчислює придатність окремих осіб.

Тепер, коли ми знаємо основні концепції GA, давайте подивимося, як використовувати його для вирішення деяких задач. Ми будемо використовувати пакет Python під назвою DEAP. Ви можете знайти всі подробиці про це на http://deap.readthedocs.io/en/master. Давайте вперед і встановіть його, виконавши таку команду:

```
$ pip3 install deap
```

Якщо ви не бачите повідомлення про помилку, все готово.

2. ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ ТА МЕТОДИЧНІ РЕКОМЕНДАЦІЇ ДО ЙОГО ВИКОНАННЯ

Завдання 2.1. Створення бітового шаблону з попередньо визначеними параметрами

У цьому завданні ми будемо використовувати варіант алгоритму One Max algorithm. One Max намагається створити бітовий рядок, який містить максимальну кількість одиниць. Це простий алгоритм, але корисно ознайомитися з бібліотекою, щоб краще зрозуміти, як реалізовувати рішення за допомогою GA. У цьому випадку може бути згенерований бітовий рядок, який містить заздалегідь визначену кількість одиниць. Ви побачите, що основна структура та частина коду схожі на приклад, використаний у бібліотеці DEAP.

Прочитайте та виконайте дії згідно рекомендацій до виконання.

РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ

Створіть новий файл Python та імпортуйте наступне:

```
import random

from deap import base, creator, tools
```

Скажімо, ми хочемо згенерувати бітовий шаблон довжиною 75, і ми хочемо, щоб він містив 45 одиниць. Нам потрібно визначити функцію пристосованості, яка може бути використана як функція оцінювання досягнення мети:

```
# Функція оцінювання
def eval_func(individual):
    target_sum = 45
    return len(individual) - abs(sum(individual) - target_sum),
```

Формула, використана в попередній функції, досягає свого максимального значення, коли кількість одиниць дорівнює 45. Довжина всіх осіб дорівнює 75. Коли кількість одиниць дорівнює 45, повернене значення буде 75.

Тепер давайте визначимо функцію для створення інструментарію (`toolbox`). По-перше, визначте об’єкт `creator` для функції пристосування, який буде відстежувати окремі індивідууми. Клас Fitness, що використовується тут, є абстрактним класом, і для нього потрібно визначити атрибут ваги `weights`. Ми будемо створювати функцію пристосованості, що максимізується, використовуючи позитивні ваги:

```
# Створіть панель інструментів із правильними параметрами def
def create_toolbox(num_bits):
    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("Individual", list,
fitness=creator.FitnessMax)
```

Перший рядок створює об’єкт пристосовуваності `FitnessMax`, що максимізує одиночну мету. Другий рядок стосується створення індивідууму. Перший індивідуум, який створюється, — це список чисел з плаваючою комою. Щоб створити цей індивідуум, ми повинні створити клас `Individual` за допомогою об’єкта `creator`. Атрибут `fitness` використовуватиме `FitnessMax`, визначений раніше.

Об’єкт `toolbox`— це об’єкт, який зазвичай використовується в DEAP. Він використовується для зберігання різних функцій разом із їхніми аргументами. Давайте створимо цей об'єкт:

```
# Ініціалізація панелі інструментів
toolbox = base.Toolbox()
```

Тепер ми почнемо реєструвати різні функції в цій панелі інструментів. Почнемо з генератора випадкових чисел, який генерує випадкове ціле число від 0 до 1. По суті, це і генерує бітові рядки:

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

Зареєструємо функцію `Individual`. Метод `initRepeat` приймає три аргументи – клас контейнера для індивідуума, функцію, яка використовується для заповнення контейнера, і кількість повторних викликів функції:

```
# Ініціалізація структур
toolbox.register("individual", tools.initRepeat,
creator.Individual,
                 toolbox.attr_bool, num_bits)
```

Нам потрібно зареєструвати функцію `population`. Ми хочемо, щоб популяція була представлена списком осіб:

```
# Визначте сукупність як список індивідуумів
toolbox.register("population", tools.initRepeat, list,
toolbox.individual)
```

Тепер нам потрібно зареєструвати генетичні оператори. Зареєструйте функцію оцінювання, яку ми визначили раніше, і яка діятиме як функція

пристосованості. Ми хочемо щоб індивідуум, який є бітовим шаблоном, мав 45 одиниць:

```
# Реєстрація оператора оцінки
toolbox.register("evaluate", eval_func)
```

Зареєструйте оператор кросинговеру під назвою `mate` за допомогою методу `cxTwoPoint`:

```
# Реєстрація оператора кросовера
toolbox.register("mate", tools.cxTwoPoint)
```

Зареєструйте оператор мутації під назвою `mutate` за допомогою `mutFlipBit`. Нам потрібно вказати ймовірність зміни кожного атрибута за допомогою `indpb`:

```
# Реєстрація оператора мутації
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
```

Зареєструйте оператора відбору за допомогою `selTournament`. У ньому вказано, які особини будуть відібрані для розведення:

```
    # Оператор по відбору особин для розмноження
    toolbox.register("select", tools.selTournament, tournsize=3)
return toolbox
```

Повинен вийти такий об’єкт `toolbox`

```
# Create the toolbox with the right parameters
def create_toolbox(num_bits):
    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("Individual", list, fitness=creator.FitnessMax)

    # Initialize the toolbox
    toolbox = base.Toolbox()

    # Generate attributes
    toolbox.register("attr_bool", random.randint, 0, 1)

    # Initialize structures
    toolbox.register("individual", tools.initRepeat, creator.Individual,
                     toolbox.attr_bool, num_bits)

    # Define the population to be a list of individuals
    toolbox.register("population", tools.initRepeat, list,
toolbox.individual)

    # Register the evaluation operator
    toolbox.register("evaluate", eval_func)

    # Register the crossover operator
    toolbox.register("mate", tools.cxTwoPoint)

    # Register a mutation operator
    toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
```

```
    # Operator for selecting individuals for breeding
    toolbox.register("select", tools.selTournament, tournsize=3)

    return toolbox
```

Це реалізація всіх концепцій, розглянутих раніше. Функція генератора панелі інструментів поширена в DEAP, і ми будемо використовувати її і в подальшому. Тому, важливо витратити трохи часу, щоб зрозуміти, як було створено набор інструментів.

Визначте основну функцію, починаючи з довжини бітового шаблону:

```
if __name__ == "__main__":
    # Визначити кількість бітів
    num_bits = 75
```

Створіть панель інструментів за допомогою функції, яку ми визначили раніше:

```
    # Створіть панель інструментів,
    # використовуючи наведений вище параметр
    toolbox = create_toolbox(num_bits)
```

Встановіть початкове значення для запуску генератора випадкових чисел, щоб забезпечити отримання повторювання результатів:

```
    # Задати генератор випадкових чисел
    random.seed(7)
```

Створіть початкову популяцію, скажімо, з 500 особин за допомогою методу, доступного в об’єкті toolbox. Не соромтеся змінювати це число та експериментувати з ним:

```
# Створіть початкову популяцію з 500 особин
population = toolbox.population(n=500)
```

Визначте ймовірності схрещування та мутації. Знову ж таки, це параметри, які визначає користувач. Отже, ви можете змінити ці параметри та подивитися, як вони вплинуть на результат:

```
# Визначте ймовірність схрещування та мутації
probab_crossing, probab_mutating = 0.5, 0.2
```

Визначте кількість поколінь, необхідних для повторення, доки процес не завершиться. Якщо ви збільшуєте кількість поколінь, ви даєте йому більше циклів для вдосконалення функції пристосованості популяції:

```
# Визначте кількість поколінь
num_generations = 60
```

Оцініть усіх осіб у популяції за допомогою функцій відповідності:

```
print('\n Початок процесу еволюції')

# Оцінити всю сукупність
fitnesses = list(map(toolbox.evaluate, population))
for ind, fit in zip(population, fitnesses):
    ind.fitness.values = fit
```

Почніть ітерацію через покоління:

```
print('\nEvaluated', len(population), 'individuals')

# Ітерація через покоління
for g in range(num_generations):
    print("\n===== Generation", g)
```

У кожному поколінні виберіть особин наступного покоління за допомогою оператора вибору, який ми зареєстрували на панелі інструментів раніше:

```
        # Виберіть індивідуумів наступного покоління
        offspring = toolbox.select(population, len(population))
```

Клонуйте вибраних осіб:

```
        # Клонувати вибраних індивідуумів
        offspring = list(map(toolbox.clone, offspring))
```

Застосуйте кросинговер і мутацію до особин наступного покоління, використовуючи значення ймовірності, визначені раніше. Після цього скиньте значення функції пристосованості:

```
        # Apply crossover and mutation on the offspring
        for child1, child2 in zip(offspring[::2],
offspring[1::2]):
            # Cross two individuals
            if random.random() < probab_crossing:
                toolbox.mate(child1, child2)

            # «Забудьте» значення функції пристосованості дітей
                del child1.fitness.values
                del child2.fitness.values
```

Застосуйте мутацію до особин наступного покоління, використовуючи відповідне значення ймовірності, визначене раніше. Після цього скиньте значення функції пристосованості:

```
        # Застосування мутації
        for mutant in offspring:
            # Мутація індивідуума
            if random.random() < probab_mutating:
                toolbox.mutate(mutant)
                del mutant.fitness.values
```

Оцініть осіб із недійсними значеннями міцності:

```
        # Оцініть індивідуумів з поганою придатністю
        invalid_ind = [ind for ind in offspring if not
ind.fitness.valid]
        fitnesses = map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        print('Evaluated', len(invalid_ind), 'individuals')
```

Замініть популяцію наступним поколінням особин:

```
        # Популяція повністю замінюється потомством
        population[:] = offspring
```

Роздрукуйте статистику для поточного покоління, щоб побачити, як воно розвивається:

```
        # Зберіть усі дані про фітнес в один список і
роздрукуйте статистику
        fits = [ind.fitness.values[0] for ind in population]

        length = len(population)
        mean = sum(fits) / length
        sum2 = sum(x * x for x in fits)
        std = abs(sum2 / length - mean ** 2) ** 0.5

        print('Min =', min(fits), ', Max =', max(fits))
        print('Average =', round(mean, 2), ', Standard deviation
=',
              round(std, 2))

    print("\n==== End of evolution")
```

Надрукуйте кінцевий результат:

```
    best_ind = tools.selBest(population, 1)[0]
    print('\nBest individual:\n', best_ind)
    print('\nNumber of ones:', sum(best_ind))
```

Повний код основної функції наведено далі.

```
if __name__ == "__main__":
    # Define the number of bits
    num_bits = 75

    # Create a toolbox using the above parameter
    toolbox = create_toolbox(num_bits)

    # Seed the random number generator
    random.seed(7)

    # Create an initial population of 500 individuals
    population = toolbox.population(n=500)

    # Define probabilities of crossing and mutating
    probab_crossing, probab_mutating = 0.5, 0.2

    # Define the number of generations
    num_generations = 60

    print('\nStarting the evolution process')

    # Evaluate the entire population
    fitnesses = list(map(toolbox.evaluate, population))
    for ind, fit in zip(population, fitnesses):
        ind.fitness.values = fit

    print('\nEvaluated', len(population), 'individuals')

    # Iterate through generations
    for g in range(num_generations):
        print("\n===== Generation", g)

        # Select the next generation individuals
        offspring = toolbox.select(population, len(population))

        # Clone the selected individuals
        offspring = list(map(toolbox.clone, offspring))

        # Apply crossover and mutation on the offspring
        for child1, child2 in zip(offspring[::2], offspring[1::2]):
            # Cross two individuals
            if random.random() < probab_crossing:
                toolbox.mate(child1, child2)

                # "Forget" the fitness values of the children
                del child1.fitness.values
                del child2.fitness.values

        # Apply mutation
        for mutant in offspring:
            # Mutate an individual
            if random.random() < probab_mutating:
                toolbox.mutate(mutant)
                del mutant.fitness.values

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        print('Evaluated', len(invalid_ind), 'individuals')
```

```
        # The population is entirely replaced by the offspring
        population[:] = offspring

        # Gather all the fitnesses in one list and print the stats
        fits = [ind.fitness.values[0] for ind in population]

        length = len(population)
        mean = sum(fits) / length
        sum2 = sum(x * x for x in fits)
        std = abs(sum2 / length - mean ** 2) ** 0.5

        print('Min =', min(fits), ', Max =', max(fits))
        print('Average =', round(mean, 2), ', Standard deviation =',
              round(std, 2))

    print("\n==== End of evolution")

    best_ind = tools.selBest(population, 1)[0]
    print('\nBest individual:\n', best_ind)
    print('\nNumber of ones:', sum(best_ind))
```

Якщо ви з'єднаєте та запустите код, то побачите роздруковані ітерації.

Наприкінці ви побачите щось подібне до наступного, що вказує на кінець еволюції:

```
===== Generation 57
Evaluated 306 individuals
Min = 68.0 , Max = 75.0
Average = 74.02 , Standard deviation = 1.27

===== Generation 58
Evaluated 276 individuals
Min = 69.0 , Max = 75.0
Average = 74.15 , Standard deviation = 1.18

===== Generation 59
Evaluated 288 individuals
Min = 69.0 , Max = 75.0
Average = 74.12 , Standard deviation = 1.24

==== End of evolution

Best individual:
 [1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1
, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0
, 1, 0, 0, 0, 1]

Number of ones: 45
```

Ваш рисунок подібний до цього занесіть у звіт.

Процес еволюції закінчується після 60 поколінь (з нульовим індексом). Після цього вибирається найкращий індивідуум, який друкується на виході. Він має 45 одиниць, що є підтвердженням результату, оскільки цільова сума становить 45 у функції оцінювання.

Збережіть код робочої програми під назвою LR_7_task_1.py
Код програми, результати виведення занесіть у звіт.
Зробіть висновок

Завдання 2.2. Візуалізація процесу еволюції

Візуалізуйте процес еволюції за допомогою методу *Стратегія еволюції шляхом адаптації коваріаційної матриці* (Covariance Matrix Adaptation Evolution Strategy – CMA-ES). Це еволюційний алгоритм, який використовується для вирішення нелінійних задач у неперервній області. Техніка CMA-ES є надійною, добре вивченою та вважається «найсучаснішою» в еволюційних алгоритмах. Наступний код є невеликою зміною прикладу показано в бібліотеці DEAP.

РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ

Створіть новий файл Python та імпортуйте такі пакети.

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

Визначте функцію для створення панелі інструментів. Ми визначимо функцію `FitnessMin`, що матиме негативні вагові коефіцієнти:

```
# Функція створення панелі інструментів
def create_toolbox(strategy):
    creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
    creator.create("Individual", list,
fitness=creator.FitnessMin)
```

Створіть набір інструментів і зареєструйте функцію оцінювання, як показано нижче:

```
    toolbox = base.Toolbox()
    toolbox.register("evaluate", benchmarks.rastrigin)

    # Задати генератор випадкових чисел
    np.random.seed(7)
```

Зареєструйте методи `generate` та `update`. Це необхідно для роботи парадигми «генерація – оновлення», у відповідності до якої ми генеруємо популяцію, що базується на стратегії, а потім ця стратегія оновлюється на основі популяції:

```
    toolbox.register("generate", strategy.generate,
creator.Individual)
    toolbox.register("update", strategy.update)

    return toolbox
```

Визначте головну функцію. Почніть із визначення кількості індивідуумів і кількість поколінь:

```
if __name__ == "__main__":
    # Розмір проблеми
    num_individuals = 10
    num_generations = 125
```

Визначте стратегію перед початком процесу:

```
    # Створення стратегії за допомогою алгоритму CMA-ES
    strategy = cma.Strategy(centroid=[5.0]*num_individuals,
sigma=5.0,
            lambda_=20*num_individuals)
```

Створіть інструментарій на основі стратегії:

```
    # Створіть панель інструментів на основі наведеної вище
стратегії
    toolbox = create_toolbox(strategy)
```

Створіть об’єкт `HallOfFame`. Об’єкт `HallOfFame` містить найкращих індивідуумів, що коли-небудь існували в популяції. Цей об’єкт завжди зберігається у відсортованому форматі. Таким чином, першим елементом у цьому об’єкті є індивідуум, який має найкраще значення пристосованості на протязі всього процесу еволюції:

```
     # Створіть обєкт HallOfFame
    hall_of_fame = tools.HallOfFame(1)
```

Зареєструвати статистику за допомогою методу `Statistics`:

```
    # Зареєструйте відповідну статистику
    stats = tools.Statistics(lambda x: x.fitness.values)
    stats.register("avg", np.mean)
    stats.register("std", np.std)
    stats.register("min", np.min)
    stats.register("max", np.max)
```

Визначте журнал для відстеження еволюційних записів. Він являє собою хронологічний список словників:

```
    logbook = tools.Logbook()
    logbook.header = "gen", "evals", "std", "min", "avg", "max"
```

Визначте об’єкти для компіляції всіх даних:

```
    # Об'єкти, які збиратимуть дані
```

```
    sigma = np.ndarray((num_generations, 1))
    axis_ratio = np.ndarray((num_generations, 1))
    diagD = np.ndarray((num_generations, num_individuals))
    fbest = np.ndarray((num_generations,1))
    best = np.ndarray((num_generations, num_individuals))
    std = np.ndarray((num_generations, num_individuals))
```

Виконуємо ітерацію по поколінням:

```
    for gen in range(num_generations):
        # Створити нову популяцію
        population = toolbox.generate()
```

Оцініть індивідуумів за допомогою функції пристосованості:

```
        # Оцініть індивідуумів
        fitnesses = toolbox.map(toolbox.evaluate, population)
        for ind, fit in zip(population, fitnesses):
            ind.fitness.values = fit
```

Оновіть стратегію на основі популяції:

```
         # Оновіть стратегію за допомогою оцінених індивідуумів
        toolbox.update(population)
```

Оновіть `HallOfFame` та статистику поточним поколінням людей:

```
         # Оновіть HallOfFame та статистику для
         # розраховуємої в даний момент популяції

        hall_of_fame.update(population)
        record = stats.compile(population)
        logbook.record(evals=len(population), gen=gen, **record)

        print(logbook.stream)
```

Збережіть дані для побудови:

```
        # Збережіть данні для побудови графіків
        sigma[gen] = strategy.sigma
        axis_ratio[gen] =
max(strategy.diagD)**2/min(strategy.diagD)**2
        diagD[gen, :num_individuals] = strategy.diagD**2
        fbest[gen] = hall_of_fame[0].fitness.values
        best[gen, :num_individuals] = hall_of_fame[0]
        std[gen, :num_individuals] = np.std(population, axis=0)
```

Визначте вісь x і побудуйте статистичні графіки:

```
     # На осі абсцисс буде відкладено кількість оцінок
```

```
    x = list(range(0, strategy.lambda_ * num_generations,
strategy.lambda_))
    avg, max_, min_ = logbook.select("avg", "max", "min")
    plt.figure()
    plt.semilogy(x, avg, "--b")
    plt.semilogy(x, max_, "--b")
    plt.semilogy(x, min_, "-b")
    plt.semilogy(x, fbest, "-c")
    plt.semilogy(x, sigma, "-g")
    plt.semilogy(x, axis_ratio, "-r")
    plt.grid(True)
    plt.title("Синій: f-значення, зелений: sigma, червоний: axis
ratio")
```

Побудуємо графік ходу процесу:

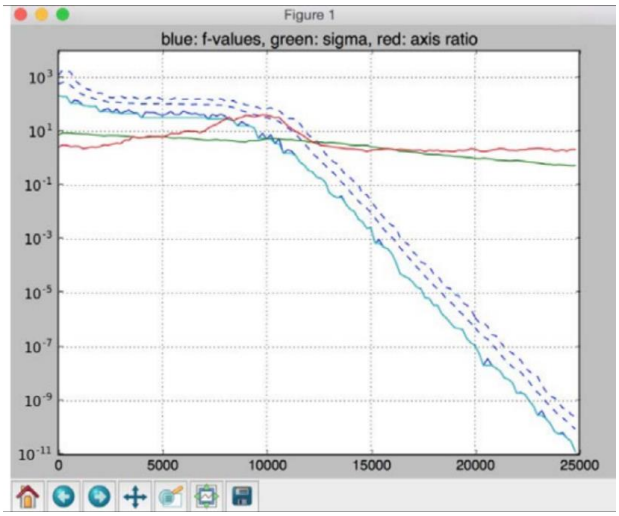
```
    plt.figure()
    plt.plot(x, best)
    plt.grid(True)
    plt.title("Обєктні змінні")

    plt.figure()
    plt.semilogy(x, diagD)
    plt.grid(True)
    plt.title("Масштабування (Всі основні осі)")

    plt.figure()
    plt.semilogy(x, std)
    plt.grid(True)
    plt.title("Стандартне відхилення по всіх координатах")

    plt.show()
```

Якщо ви запустите код, ви отримаєте чотири рисунки. Перший рисунок показує різні параметри:



На другому рисунку показано змінні об’єкта. На третьому рисунку показано масштабування. Четвертий рисунок показує стандартні відхилення.

Зробіть скріншоти всіх 4-х рисунків та вставте їх у звіт.

У вікні терміналу також ви побачите інформацію про хід процесу.

Зробіть скріншоти початку та кінця інформації з терміналу та вставте їх у звіт.

На початку ви побачите щось на зразок наступного:

gen	evals	std	min	avg	max
0	200	188.36	217.082	576.281	1199.71
1	200	250.543	196.583	659.389	1869.02
2	200	273.081	199.455	683.641	1770.65
3	200	215.326	111.298	503.933	1579.3
4	200	133.046	149.47	373.124	790.899
5	200	75.4405	131.117	274.092	585.433
6	200	61.2622	91.7121	232.624	426.666
7	200	49.8303	88.8185	201.117	373.543
8	200	39.9533	85.0531	178.645	326.209
9	200	31.3781	87.4824	159.211	261.132
10	200	31.3488	54.0743	144.561	274.877
11	200	30.8796	63.6032	136.791	240.739
12	200	24.1975	70.4913	125.691	190.684
13	200	21.2274	50.6409	122.293	177.483
14	200	25.4931	67.9873	124.132	199.296
15	200	26.9804	46.3411	119.295	205.331
16	200	24.8993	56.0033	115.614	176.702
17	200	21.9789	61.4999	113.417	170.156
18	200	21.2823	50.2455	112.419	190.677
19	200	22.5016	48.153	111.543	166.2
20	200	21.1602	32.1864	106.044	171.899
21	200	23.3864	52.8601	107.301	163.617
22	200	23.1008	51.1226	109.628	185.777
23	200	22.0836	51.3058	106.402	179.673

У кінці ви побачите наступне:

100	200	2.38865e-07	1.12678e-07	5.18814e-07	1.23527e-06
101	200	1.49444e-07	5.56979e-08	3.3199e-07	7.98774e-07
102	200	1.11635e-07	2.07109e-08	2.41361e-07	7.96738e-07
103	200	9.50257e-08	3.69117e-08	1.94641e-07	5.75896e-07
104	200	5.63849e-08	2.09827e-08	1.26148e-07	2.887e-07
105	200	4.42488e-08	1.64212e-08	8.6972e-08	2.58639e-07
106	200	2.34933e-08	1.28302e-08	5.47789e-08	1.54658e-07
107	200	1.74434e-08	7.13185e-09	3.64705e-08	9.88235e-08
108	200	1.17157e-08	6.32208e-09	2.54673e-08	7.13075e-08
109	200	8.73027e-09	4.60369e-09	1.79681e-08	5.88066e-08
110	200	6.39874e-09	1.92573e-09	1.43229e-08	4.00087e-08
111	200	5.31196e-09	2.05551e-09	1.13736e-08	3.16793e-08
112	200	3.15607e-09	1.72427e-09	7.28548e-09	1.67727e-08
113	200	2.3789e-09	1.01164e-09	5.01177e-09	1.24541e-08
114	200	1.38424e-09	6.43112e-10	2.94696e-09	9.25819e-09
115	200	1.04172e-09	2.87571e-10	2.06068e-09	7.90436e-09
116	200	6.08685e-10	4.32905e-10	1.4704e-09	3.80221e-09
117	200	4.51515e-10	2.1538e-10	9.23627e-10	2.2759e-09
118	200	2.77204e-10	1.46869e-10	6.3507e-10	1.44637e-09
119	200	2.06475e-10	7.54881e-11	4.41427e-10	1.33167e-09
120	200	1.3138e-10	5.97282e-11	2.98116e-10	8.60453e-10
121	200	9.52385e-11	6.753e-11	2.32358e-10	5.45441e-10
122	200	7.55001e-11	4.1851e-11	1.72688e-10	5.05054e-10
123	200	5.52125e-11	3.2216e-11	1.23505e-10	3.10081e-10
124	200	4.38068e-11	1.32871e-11	8.94929e-11	2.57202e-10

На скріншотах ви повинні побачити, що усі значення продовжують зменшуватися в міру просування. Це означає, що процес збігається (наближається до точки оптимума).

Збережіть код робочої програми з обов'язковими коментарям під назвою LR_7_task_2.py

Код робочої програми занесіть у звіт

Зробіть висновок

Завдання 2.3. Розв'язування задачі символьної регресії

Давайте подивимося, як використовувати генетичне програмування для вирішення проблеми символьної регресії. Важливо розуміти, що генетичне програмування – це не те саме, що GA.

Генетичне програмування — це тип еволюційного алгоритму, у якому рішення виступають у формі комп'ютерних програм. По суті індивудууми в кожному поколінні будуть комп'ютерні програмами, і рівень їхньої пристосованості відповідатиме їхній здатності вирішувати проблеми. Ці програми модифікуються на кожній ітерації за допомогою GA. Підсумовуючи можна сказати, що генетичне програмування - це застосування GA.

Переходячи до проблеми символьної регресії, ми маємо поліноміальний вираз, який тут потрібно апроксимувати. Це класична задача регресії, де ми намагаємося оцінити базову функцію. У цьому прикладі ми використаємо вираз: f(x) = 2x^3 – 3x^2 + 4x – 1

Код, який тут обговорюється, є варіантом задачі символьної регресії, наведеної в Бібліотека DEAP.

РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ

Створіть новий файл Python та імпортуйте такі пакети.

```
import operator
import math
import random

import numpy as np
from deap import algorithms, base, creator, tools, gp
```

Створіть оператор ділення, який може обробити помилку ділення на нуль:

```
# Визначення нових функцій
def division_operator(numerator, denominator):
    if denominator == 0:
```

```
        return 1

    return numerator / denominator
```

Визначте функцію оцінки, яка використовуватиметься для обчислення функції пристосованості. Ми потребуємо визначити функцію, що викликається, для виконання обчислень на вхідних даних:

```
# Визначте функцію оцінювання
def eval_func(індивідуум, бали):

# Define the evaluation function
def eval_func(individual, points):
    # Transform the tree expression in a callable function
    func = toolbox.compile(expr=individual)
```

Обчисліть середню квадратичну помилку (MSE) між функцією, визначеною раніше, і оригінальний вираз:

```
    # Оцініть середню квадратичну помилку
    mse = ((func(x) - (2 * x**3 - 3 * x**2 + 4 * x - 1))**2 for
x in points)

    return math.fsum(mse) / len(points),
```

Визначте функцію для створення панелі інструментів. Щоб створити тут панель інструментів, необхідно створити набір примітивів. Ці примітиви є операторами, які будуть використовувався під час еволюції. Вони служать будівельними блоками для індивідуумів. У якості примітивів будемо користуватися базовими арифметичними функціями:

```
# Функція для створення панелі інструментів
def create_toolbox():
    pset = gp.PrimitiveSet("MAIN", 1)
    pset.addPrimitive(operator.add, 2)
    pset.addPrimitive(operator.sub, 2)
    pset.addPrimitive(operator.mul, 2)
    pset.addPrimitive(division_operator, 2)
    pset.addPrimitive(operator.neg, 1)
    pset.addPrimitive(math.cos, 1)
    pset.addPrimitive(math.sin, 1)
```

Далі оголосимо «ефемерну» константу. Ефемерна константа - це спеціальний термінальний тип, який не має фіксованого значення. Коли дана програма додає таку ефемерну константа до дерева, функція виконується. Потім вставляється результат у дерево як термінальна константа. Ці термінальні константи можуть приймати значення -1, 0 або 1:

```
    pset.addEphemeralConstant("rand101", lambda:
random.randint(-1,1))
```

Іменами для аргументів за замовчуванням є - ARGx. Давайте перейменуємо його на `x`:

```
    pset.renameArguments(ARG0='x')
```

Нам потрібно визначити два об'єктних типи – `ftness` та `Individual`. Давайте зробимо це за допомогою `creator`:

```
    creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
    creator.create("Individual", gp.PrimitiveTree,
fitness=creator.FitnessMin)
```

Створіть панель інструментів і зареєструйте функції. Процес реєстрації завершено як і в попередніх завданнях:

```
    toolbox = base.Toolbox()

    toolbox.register("expr", gp.genHalfAndHalf, pset=pset,
min_=1, max_=2)
    toolbox.register("individual", tools.initIterate,
creator.Individual, toolbox.expr)
    toolbox.register("population", tools.initRepeat, list,
toolbox.individual)
    toolbox.register("compile", gp.compile, pset=pset)
    toolbox.register("evaluate", eval_func, points=[x/10. for x
in range(-10,10)])
    toolbox.register("select", tools.selTournament, tournsize=3)
    toolbox.register("mate", gp.cxOnePoint)
    toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
    toolbox.register("mutate", gp.mutUniform,
expr=toolbox.expr_mut, pset=pset)

    toolbox.decorate("mate",
gp.staticLimit(key=operator.attrgetter("height"), max_value=17))
    toolbox.decorate("mutate",
gp.staticLimit(key=operator.attrgetter("height"), max_value=17))

    return toolbox
```

Визначте головну функцію та почніть із визначення початкового положення генератора випадкових чисел:

```
if __name__ == "__main__":
    random.seed(7)
```

Створіть об'єкт панелі інструментів:

```
toolbox = create_toolbox()
```

Визначте початкову популяцію за допомогою методу, доступного в об’єкті `toolbox` з використанням 450 осіб. Кількість осіб може бути змінена. Не соромтеся експериментуйте з ним. Також визначте об’єкт `hall_of_fame`:

```
population = toolbox.population(n=450)
hall_of_fame = tools.HallOfFame(1)
```

Під час створення GA корисно використовувати статистику. Визначте об’єкти статистики:

```
stats_fit = tools.Statistics(lambda x: x.fitness.values)
stats_size = tools.Statistics(len)
```

Зареєструйте статистику за допомогою об’єктів, визначених раніше:

```
mstats = tools.MultiStatistics(fitness=stats_fit,
size=stats_size)
mstats.register("avg", np.mean)
mstats.register("std", np.std)
mstats.register("min", np.min)
mstats.register("max", np.max)
```

Визначте ймовірність кросинговеру, ймовірність мутації та кількість покоління:

```
# Визначаємо параметри
probab_crossover = 0.4
probab_mutate = 0.2
num_generations = 60
```

Запустіть еволюційний алгоритм, використовуючи наведені вище параметри:

```
population, log = algorithms.eaSimple(population, toolbox,
        probab_crossover, probab_mutate, num_generations,
        stats=mstats, halloffame=hall_of_fame, verbose=True)
```

Якщо ви запустите код, ви побачить наступне на початку еволюції:

		fitness				size			
gen	nevals	avg	max	min	std	avg	max	min	std
0	450	18.6918	47.1923	7.39087	6.27543	3.73556	7	2	1.62449
1	251	15.4572	41.3823	4.46965	4.54993	3.80222	12	1	1.81316
2	236	13.2545	37.7223	4.46965	4.06145	3.96889	12	1	1.98861
3	251	12.2299	60.828	4.46965	4.70055	4.19556	12	1	1.9971
4	235	11.001	47.1923	4.46965	4.48841	4.84222	13	1	2.17245
5	229	9.44483	31.478	4.46965	3.8796	5.56	19	1	2.43168
6	225	8.35975	22.0546	3.02133	3.40547	6.38889	15	1	2.40875
7	237	7.99309	31.1356	1.81133	4.08463	7.14667	16	1	2.57782
8	224	7.42611	359.418	1.17558	17.0167	8.33333	19	1	3.11127
9	237	5.70308	24.1921	1.17558	3.71991	9.64444	23	1	3.31365
10	254	5.27991	30.4315	1.13301	4.13556	10.5089	25	1	3.51898

У кінці ви побачите наступне:

44	236	9.30096	3481.25	0.0277886	163.888	26.9622	55	1	6.27169
45	231	1.73196	86.7372	0.0342642	6.8119	27.4711	51	2	5.27807
46	227	1.86086	185.328	0.0342642	10.1143	28.0644	56	1	6.10812
47	216	12.5214	4923.66	0.0342642	231.837	29.1022	54	1	6.45898
48	232	14.3469	5830.89	0.0322462	274.536	29.8244	58	3	6.24093
49	242	2.56984	272.833	0.0322462	18.2752	29.9267	51	1	6.31446
50	227	2.80136	356.613	0.0322462	21.0416	29.7978	56	4	6.50275
51	243	1.75099	86.0936	0.0322462	5.70833	29.8089	56	1	6.62379
52	253	10.9184	3435.84	0.0227048	163.602	29.9911	55	1	6.66833
53	243	1.80265	48.0418	0.0227048	4.73856	29.88	55	1	7.33084
54	234	1.74487	86.0936	0.0227048	6.0249	30.6067	55	1	6.85782
55	220	1.58888	31.094	0.0132398	3.82809	30.5644	54	1	6.96669
56	234	1.46711	103.287	0.00766444	6.81157	30.6689	55	3	6.6806
57	250	17.0896	6544.17	0.00424267	308.689	31.1267	60	4	7.25837
58	231	1.66757	141.584	0.00144401	7.35306	32	52	1	7.23295
59	229	2.22325	265.224	0.00144401	13.388	33.5489	64	1	8.38351
60	248	2.60303	521.804	0.00144401	24.7018	35.2533	58	1	7.61506

Ми бачимо, що значення в стовпці min стає все меншим, що вказує на це похибка апроксимації розв'язку рівняння стає меншою. Тобто функція апроксимується краще.

Код програми та результати занесіть у звіт.

Програмний код збережіть під назвою LR_6_task_3.py

Завдання 2.4. Дослідження мурашиного алгоритму на прикладі рішення задачі комівояжера

У файлі Відстані між обласними центрами України.docx міститься таблиця відстаней між обласними центрами України. Міста пронумеровані по алфавіту. ***Ваш номер за журналом групи (або номер за табличкою у рейтингу) повинен відповідати місту з якого ви будете починати виїзд.*** Наприклад: за рейтинговою таблицею - № 6 Драк Тарас Сергійович починає з № 6. Івано-Франківськ. Виїзд починається і закінчується в цьому місті.

Використовуючи мову Python, розробити програму, що реалізує метод мурашиних колоній, для рішення задачі комівояжера, який їздить по містах України.

ТЕОРЕТИЧНА ЧАСТИНА

1. Принципи поведінки мурах

Мурахи належать до так званих «соціальних комах», тобто комах, що живуть в межах певного колективу — сім'ї або колонії. На Землі біля двох відсотків комах є «соціальними», половина з яких припадає на мурах. Поведінка мурах під час транспортування їжі, обминання перешкод, побудови мурашника тощо наближається до теоретично оптимальної. Основу «соціальної» поведінки мурах складає самоорганізація — сукупність динамічних механізмів, за допомогою яких система досягає глобальної мети в результаті взаємодії елементів на низькому рівні. Принциповою особливістю такої низькорівневої взаємодії є використання елементами системи лише локальної інформації, без будь-якого централізованого управління та звернення до глобального образу, який репрезентує систему у зовнішньому світі. Самоорганізація є результатом взаємодії таких чотирьох компонентів: 1) випадковість; 2) додатний зворотний зв'язок; 3) від'ємний зворотний зв'язок; 4) багатократність взаємодій.

Мурахи використовують два способи передачі інформації:

прямий — обмін харчами, мандибулярний, візуальний та хімічний контакти тощо;

непрямий — стигмержі (stigmergy). Стигмержі — це рознесений у часовому просторі тип взаємодії між елементами системи, коли один суб'єкт взаємодії змінює деяку частину оточуючого середовища, а решта використовує інформацію про її стан пізніше, коли знаходяться в її околі. Біологічно, стигмержі здійснюється через феромон (pheromone) — спеціальний секрет, який відкладається як слід під час руху мурахи. Чим вища концентрація феромону на стежці, тим більше мурах буде рухатись по ній. Феромон з часом випаровується, що дозволяє мурахам адаптувати свою поведінку до зміни оточуючого середовища. Розподіл феромону по простору пересування мурах є своєрідною глобальною пам'яттю, яка динамічно змінюється. Кожна мураха може сприймати та змінювати лише локальну чарунку цієї глобальної пам'яті мурашника.

Розглянемо, як колективна поведінка біологічних мурах забезпечує знаходження найкоротшого шляху до їжі на прикладі експериментів на асиметричному мості [6]. Асиметричний міст (рис. 1) з'єднує гніздо мурах з джерелом їжі двома гілками різної довжини. Експерименти [6] проводилися за такою схемою: 1) будувався міст A-B-C-D; 2) відчинялися дверцята в точці A; 3) фіксувалась кількість мурах, які обрали довгий (A-C-D) та короткий (A-B-D) шляхи. На початку експериментів мурахи обирали обидві гілки з однаковою ймовірністю тому, що на мості не було феромонів. Через деякий

час майже усі мурахи пересувались найкоротшим маршрутом A-B-D, що пояснюється таким чином. Мурахи, які обрали короткий маршрут A-B-D-B-A, скоріше повертались з їжею в гніздо, залишаючи феромонні сліди на короткій гілці мосту.

При наступному виборі маршруту, мурахи віддавали перевагу короткій гілці мосту, тому що на ній вища концентрація феромонів. Таким чином, феромон швидше накопичується на гільці A-B-D, що підштовхує мурах до вибору найкоротшого маршруту.

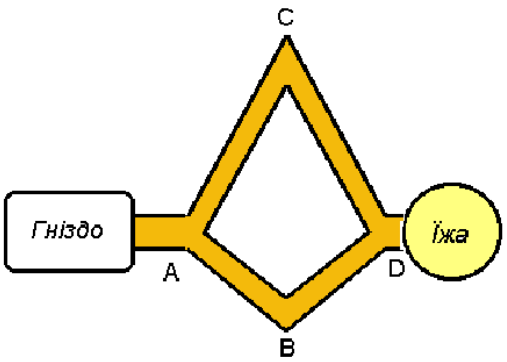


Рис. 1. Асиметричний міст

2. Основні поняття

Мурашині алгоритми серйозно досліджуються європейськими вченими із середини 1990-х років. На сьогоднішній день вже отримано гарні результати для оптимізації таких складних комбінаторних задач,як задача комівояжера, задача оптимізації маршрутів вантажівок, задача розфарбування графа, квадратична задача про призначення, задача оптимізації сіткових графіків, задача календарного планування і т. ін. Особливо ефективні мурашині алгоритми при динамічній оптимізації процесів у розподілених нестаціонарних системах, наприклад, графіків у телекомунікаційних мережах.

Метод мурашиних колоній заснований на взаємодії декількох *мурах* (програмних агентів, що є членами великої колонії) і використовується для вирішення різних оптимізаційних задач. Агенти спільно вирішують проблему й допомагають іншим агентам у подальшій оптимізації розв’язку.

Методи мурахи (Ant algorithms), або оптимізація за принципом мурашиної колонії, мають специфічні особливості, властиві мурахам, і використовують їх для орієнтації у фізичному просторі. Природа пропонує різні методики для оптимізації деяких процесів. Методи мурашиних колоній особливо цікаві тому, що їх можна використовувати для вирішення не тільки статичних, але й динамічних задач, наприклад, задач маршрутизації в мінливих мережах.

Базова ідея методу мурашиних колоній полягає у вирішенні оптимізаційної задачі шляхом застосування непрямого зв'язку між автономними агентами.

У методі мурашиних колоній передбачається, що навколишнє середовище являє собою закриту двовимірну мережу – групу вузлів, з'єднаних за допомогою граней. Кожна грань має вагу, що позначається як відстань між двома вузлами, з'єднаними нею. Граф – двонаправлений, тому агент може подорожувати по грані в будь-якому напрямку.

Агент забезпечується набором простих правил, які дозволяють йому вибирати шлях у графі. Він підтримує список табу *tList* – список вузлів, які він вже відвідав. Таким чином, агент повинен проходити через кожний вузол тільки один раз.

Вузли в списку "поточної подорожі" *tList* розташовуються в тому порядку, у якому агент відвідував їх. Пізніше список використовується для визначення довжини шляху між вузлами.

Справжня мураха під час переміщення по шляху залишає за собою деяку кількість *феромону*. У методі мурашиних колоній агент залишає феромон на гранях мережі після завершення подорожі.

3. Мурашиний підхід до розв'язання задачі комівояжера

Задача комівояжера полягає у виборі найкоротшого замкнутого шляху, що проходить через усі міста рівно один раз. Розглянемо, як реалізувати чотири основних компоненти самоорганізаційної поведінки мурах під час оптимізації маршруту комівояжера.

Багатократність взаємодії реалізується ітераційним пошуком маршруту комівояжера одночасного декількома мурахами.

Додатний зворотний зв'язок реалізується як імітація природної поведінки мурах типу «залишання слідів – пересування по слідах». Чим більше слідів залишено на стежці — ребрі графу, тим більше мурах буде рухатись по ній. При цьому на стежці з'являються нові сліди, які приваблюють додаткових мурах. Для задачі комівояжера додатний зворотний зв'язок реалізується таким стохастичним правилом: «ймовірність включення ребра графу в маршрут мурахи пропорційна кількості феромону на ній».

Використання цього стохастичного правила забезпечує реалізацію і іншого компоненту поведінки мурах – випадковості. Кількість феромону, який відкладає мураха на ребрі графа, є зворотно пропорційною величиною до довжини відповідного маршруту комівояжера. Чим коротший маршрут комівояжера знайшла мураха, тим більше феромону буде відкладене на відповідних ребрах графу.

Використання лише додатного зворотного зв'язку призводить до передчасної збіжності алгоритму, тобто до випадку, коли усі мурахи рухаються одним і тим же субоптиальним маршрутом. Для уникнення цього використовується від'ємний зворотний зв'язок – випаровування феромону. Час випаровування феромону не повинен бути дуже великим, бо при цьому

виникає загроза збігання маршрутів усіх мурах до одного субоптимального розв'язку. З іншого боку, час випаровування не повинен бути і малим, щоб не призвести до некооперативної поведінки мурах через втрату пам'яті колонії.

Перехід мурахи з міста i в місто j на ітерації t алгоритму залежить від трьох складових: табу-списка, видимості та віртуального сліду феромону.

Табу-список (tList) — це перелік міст, які вже відвідані мурахою і заходити в які ще раз заборонено. Табу-список збільшується зі здійсненням маршруту та заповнюється нулями на початку кожної ітерації алгоритму. Позначимо через J_i^k список міст, які ще потрібно відвідати мурасі k, що перебуває у місті i. Зрозуміло, що об'єднання цих списків дає множину усіх міст з маршруту комівояжера.

Видимість – це величина обернена до відстані: $\eta_{ij} = 1/D_{ij}$, де D_{ij} — відстань між містами i та j. Видимість базується тільки на локальній інформації і являє собою «евристичну бажаність» вибору міста j, під час перебування у місті i. Чим ближче міста i та j, тим більше бажання відвідати їх.

Віртуальний слід феромону на ребрі (i – j) являє собою «бажаність, підкріплену досвідом» переходу в місто i з міста j. Інформація, яку несе слід феромону, змінюється підчас оптимізації і відображає набутий мурахами досвід. Кількість віртуального феромону на ребрі (i – j) на t-й ітерації алгоритму позначимо як $\tau_{ij}(t)$.

Ймовірність переходу k-ї мурахи з міста i у місто j на t-й ітерації розраховується за випадково-пропорційним правилом:

$$\begin{cases} P_{ij}^k(t) = \dfrac{\left[\tau_{ij}(t)\right]^\alpha \left[\eta_{ij}\right]^\beta}{\sum\limits_{l \in J_i^k} \left[\tau_{il}(t)\right]^\alpha \left[\eta_{il}\right]^\beta}, & \text{якщо } j \in J_i^k; \\ P_{ij}^k(t) = 0, & \text{якщо } j \notin J_i^k. \end{cases}$$

де α і β – два регульовані параметри, які є вагами інтенсивності сліду феромону та видимості. Якщо α = 0, то найвірогіднішим буде перехід у найближчі міста. У класичній теорії оптимізації це відповідає, так званому, пожадливому алгоритму. Якщо β = 0, тоді працює лише феромоне підсилення, що призводить до швидкого завершення роботи алгоритму через збігання маршрутів усіх мурах до одного субоптимального розв'язку.

Після завершення маршруту кожна мураха k відкладає на ребро (i – j) таку кількість феромону:

$$\Delta\tau_{ij}^k(t) = \begin{cases} \dfrac{Q}{L^k(t)}, & \text{якщо } (i,j) \in T^k(t) \\ 0, & \text{якщо } (i,j) \notin T^k(t) \end{cases},$$

де $T^k(t)$ – маршрут, зроблений мурахою k на ітерації t; $L^k(t)$ — його довжина; Q — регульований параметр, значення якого обирають одного порядку з довжиною оптимального маршруту.

Для забезпечення можливості експлуатації простору рішень потрібно забезпечити випаровування феромону — зменшення кількості відкладеного

на попередніх ітераціях феромону. Інтенсивність випаровування феромону задається за допомогою коефіцієнта випаровування $p \in [0, 1]$. Кінцеве правило оновлення феромону, яке стосується всіх ребер, приймає вигляд

$$\tau_{ij}(t+1) \leftarrow (1-p)\ \tau_{ij}(t) + \Delta\tau_{ij}(t),$$

де $\Delta\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t)$; m – кількість мурах в колонії.

На початку оптимізації кількість феромону на ребрах приймається за малу додатну константу τ_0. Загальна кількість мурах в колонії приймається постійною на весь час розв'язання задачі. Забагато мурах призводить до швидкого підсилення субоптимальних маршрутів. Коли ж мурах замало, виникає небезпека втрати кооперативної поведінки через швидке випаровування феромону. Зазвичай кількість мурах приймають рівною числу міст — кожна мураха починає маршрут з окремого міста.

4. Послідовність виконання методу

Метод мурашиних колоній виконується в наступній послідовності кроків.

Крок 1. Задати параметри методу: α – коефіцієнт, що визначає відносну значимість шляху (кількість феромона на шляху); β – параметр, що означає пріоритет відстані над кількістю феромона; ρ – коефіцієнт кількості феромона, що агент залишає на шляху, де $(1–\rho)$ показує коефіцієнт випару феромона на шляху після його завершення; Q– константу, що відноситься до кількості феромона, яку було залишено на шляху.

Крок 2. Ініціалізація методу. Створення популяції агентів. Після створення популяція агентів нарівно розподіляється по вузлах мережі. Необхідно рівномірний розподіл агентів між вузлами, щоб всі вузли мали однакові шанси стати відправною точкою. Якщо всі агенти почнуть рух з однієї точки, це буде означати, що дана точка вважається оптимальною для старту, а насправді вона такою може і не бути.

Крок 3. Рух агентів. Якщо агент ще не закінчив шлях, тобто не відвідав всі вузли мережі, для визначення наступної грані шляху використовується формула:

$$P = \frac{\tau_{ru}(t)^{a} \cdot \eta_{ru}(t)^{\beta}}{\sum_{k \in j} \tau_{ru}(t)^{a} \cdot \eta_{ru}(t)^{\beta}},$$

де J – множина граней, ще не відвіданих агентом; $\tau_{ru}(t)$ – інтенсивність феромона на грані між вузлами r і u, які утворять k-y грань, у момент часу t; $\eta_{ru}(t)$ – функція, що представляє собою зворотну величину відстані грані.

Агент подорожує тільки по вузлах, які ще не були відвідані ним,тобто по вузлах, яких не має у списку табу *tList*. Тому ймовірність розраховується тільки для граней, які ведуть до ще не відвіданих вузлів.

Крок 4. Пройдений агентом шлях відображається, коли агент відвідає всі вузли мережі. Цикли заборонені, оскільки в метод включений список табу *tList*. Після завершення може бути розрахована *довжина шляху*. Вона дорівнює сумі всіх граней, по яких подорожував агент. *Кількість феромону*, що було залишено на кожній грані шляху *i*-го агенту, визначається за формулою:

$$\Delta\tau_{ru}^{i}(t) = \frac{Q}{L^{i}(t)},$$

де $L^{i}(t)$ – довжина шляху *i*-го агенту.

Результат є засобом вимірювання шляху: короткий шлях характеризується високою концентрацією феромону, а більш довгий шлях – більш низкою. Потім отриманий результат використовується для збільшення кількості феромону уздовж кожної грані пройденого *i*-им агентом шляху:

$$\tau_{ru}(t+1) = \tau_{ru}(t) + (\Delta\tau_{ru}^{i}(t) \cdot \rho),$$

де *r*, *u* – вузли, що утворюють грані, які відвідав *i*-ий агент.

Дана формула застосовується до всього шляху, при цьому кожна грань позначається феромоном пропорційно довжині шляху. Тому варто дочекатися, поки агент закінчить подорож і тільки потім оновити рівні феромону, у противному випадку дійсна довжина шляху залишиться невідомою. Константа ρ приймає значення між 0 і 1.

$$\tau_{ru}(t) = \tau_{ru}(t) \cdot (1-\rho)$$

Тому для випаровування феромону використовується зворотний коефіцієнт відновлення шляху $(1-\rho)$.

Крок 5. Перевірка на досягнення оптимального результату. Перевірка може виконуватися для постійної кількості шляхів або до моменту, коли протягом декількох запусків не було отримано повторних змін у виборі найкращого шляху. Якщо перевірка дала позитивний результат, то відбувається закінчення роботи методу (перехід до кроку 7), у противному випадку – перехід до кроку 6.

Крок 6. Повторний запуск. Після того як шлях агента завершений, грані оновлені відповідно до довжини шляху, й відбулося випаровування феромону на всіх гранях, метод виконується повторно. Список табу очищується, і довжина шляху прирівнюється нулю. Після цього виконується перехід до кроку 3.

Крок 7. Кінець. Визначається кращий шлях, що і є розв’язком.

РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ

Створіть новий файл Python та імпортуйте такі пакети.

```
import numpy as np
import matplotlib.pyplot as plt
```

Створіть клас який буде формувати карту відстаней між містами з нанесеними на кожну відстань феромонами.

```
# Карта відстаней з феромонами
class CityMap:
    def __init__(self, distances_matrix, cities_count):
        self.distances = distances_matrix
        self.numberOfCities = cities_count
        self.pheromones = [[np.random.rand() for j in
range(cities_count)] for i in range(cities_count)]
```

Створіть функцію для оновлення значення феромонів.

```
    # Оновлення значення феромонів
    def upd_pheromones(self, evaporation_rate, pheromone_delta):
        for i, row in enumerate(self.pheromones):
            for j, col in enumerate(row):
                self.pheromones[i][j] *= (1 - evaporation_rate)
                self.pheromones[i][j] += pheromone_delta[i][j]
```

Створіть клас що описує мураху та її дії .

```
class Ant:
    def __init__(self, city_start):
        self.startingCity = city_start
        self.currentCity = city_start
        self.distance = 0
        self.visitedCities = [city_start]
```

Створіть функцію для переміщення мураи в нове місто.

```
    # Переміщення мурахи в нове місто

    def move(self, city_new, distance):
        self.currentCity = city_new
        self.visitedCities.append(city_new)
        self.distance += distance
```

Створіть клас що описує колонію мурах .

```
class Colony:
    maxColonyCycles = 50
    pheromoneAddition = 0.0005
    pheromoneEvaporationRate = 0.2
    pheromoneImportance = 0.01
    distanceImportance = 9.5
    antCanVisitPreviousCities = False

    def __init__(self, ants_num):
        self.numberOfAnts = ants_num
```

Створіть функцію для пошуку найкоротшого шляху.

```
    # Пошук найкоротшого шляху
    def find_route(self, city_map, city_num):
        min_dist = float('inf')
        route = []
        for cycle in range(self.maxColonyCycles):
            pheromones_delta = [[0.0 for i in
range(city_map.numberOfCities)] for j in
range(city_map.numberOfCities)]
            for antNumber in range(self.numberOfAnts):
                ant = Ant(city_num)
                while len(ant.visitedCities) <
city_map.numberOfCities:
                    next_city = self.get_next_city(ant,
city_map)
                    ant.move(next_city,
city_map.distances[ant.currentCity][next_city])
                ant_dist = ant.distance +
city_map.distances[ant.currentCity][ant.startingCity]
                if ant_dist < min_dist:
                    min_dist = ant_dist
                    route = ant.visitedCities
                for city in range(len(ant.visitedCities) - 1):
                    pheromones_delta[ant.visitedCities[city]][
                        ant.visitedCities[city + 1]] +=
self.pheromoneAddition / ant_dist

city_map.upd_pheromones(self.pheromoneEvaporationRate,
pheromones_delta)

        return min_dist, route
```

Створіть функцію для розрахунку ймовірностей переміщення мурахи в місто.

```
    # Формування списку ймовірностей переміщення в місто для
мурахи
    def get_probabilities(self, ant, city_map):
        result = [0 for i in range(city_map.numberOfCities)]
        total_probability = 0
        for newCity in range(city_map.numberOfCities):
            if (newCity != ant.currentCity) and
(self.antCanVisitPreviousCities or newCity not in
ant.visitedCities):
                probability =
pow(city_map.pheromones[ant.currentCity][newCity],
self.pheromoneImportance) * pow(
                    1 /
city_map.distances[ant.currentCity][newCity],
self.distanceImportance)
                result[newCity] = probability
                total_probability += probability
```

```
        result = [result[i] / total_probability for i in
range(city_map.numberOfCities)]
        return result
```

Створіть функцію вибору наступного міста для мурахи.

```
    # Вибір наступного міста для мурахи
    def get_next_city(self, ant, city_map):
        probabilities = self.get_probabilities(ant, city_map)
        random_value = np.random.rand()
        for i in range(city_map.numberOfCities):
            if probabilities[i] > random_value:
                return i
            else:
                random_value -= probabilities[i]
        return -1
```

Створіть словник де зберігаються відстані між містами.

```
# Відстані між містами
distance = [
    [0, 645, 868, 125, 748, 366, 256, 316, 1057, 382, 360, 471,
428, 593, 311, 844, 602, 232, 575, 734, 521, 120,
     343, 312, 396],
    [645, 0, 252, 664, 81, 901, 533, 294, 394, 805, 975, 343,
468, 196, 957, 446, 430, 877, 1130, 213, 376, 765,
     324, 891, 672],
    [868, 252, 0, 858, 217, 1171, 727, 520, 148, 1111, 1221,
611, 731, 390, 1045, 591, 706, 1100, 1391, 335, 560,
     988, 547, 1141, 867],
    [125, 664, 858, 0, 738, 431, 131, 407, 1182, 257, 423, 677,
557, 468, 187, 803, 477, 298, 671, 690, 624, 185,
     321, 389, 271],
    [748, 81, 217, 738, 0, 1119, 607, 303, 365, 681, 833, 377,
497, 270, 925, 365, 477, 977, 1488, 287, 297, 875,
     405, 957, 747],
    [366, 901, 1171, 431, 1119, 0, 561, 618, 1402, 328, 135,
747, 627, 898, 296, 1070, 908, 134, 280, 1040, 798,
     246, 709, 143, 701],
    [256, 533, 727, 131, 607, 561, 0, 298, 811, 388, 550, 490,
489, 337, 318, 972, 346, 427, 806, 478, 551, 315,
     190, 538, 149],
    [316, 294, 520, 407, 303, 618, 298, 0, 668, 664, 710, 174,
294, 246, 627, 570, 506, 547, 883, 387, 225, 435,
     126, 637, 363],
    [1057, 394, 148, 1182, 365, 1402, 811, 668, 0, 1199, 1379,
857, 977, 474, 1129, 739, 253, 1289, 1539, 333, 806,
     1177, 706, 1292, 951],
    [382, 805, 1111, 257, 681, 328, 388, 664, 1199, 0, 152, 780,
856, 725, 70, 1052, 734, 159, 413, 866, 869, 263,
     578, 336, 949],
    [360, 975, 1221, 423, 833, 135, 550, 710, 1379, 152, 0, 850,
970, 891, 232, 1173, 896, 128, 261, 1028, 1141,
```

```
     240, 740, 278, 690],
    [471, 343, 611, 677, 377, 747, 490, 174, 857, 780, 850, 0,
120, 420, 864, 282, 681, 754, 999, 556, 51, 590, 300,
     642, 640],
    [428, 468, 731, 557, 497, 627, 489, 294, 977, 856, 970, 120,
0, 540, 741, 392, 800, 660, 1009, 831, 171, 548,
     420, 515, 529],
    [593, 196, 390, 468, 270, 898, 337, 246, 474, 725, 891, 420,
540, 0, 665, 635, 261, 825, 1149, 141, 471, 653,
     279, 892, 477],
    [311, 957, 1045, 187, 925, 296, 318, 627, 1129, 70, 232,
864, 741, 665, 0, 1157, 664, 162, 484, 805, 834, 193,
     508, 331, 458],
    [844, 446, 591, 803, 365, 1070, 972, 570, 739, 1052, 1173,
282, 392, 635, 1157, 0, 896, 1097, 1363, 652, 221,
     964, 696, 981, 1112],
    [602, 430, 706, 477, 477, 908, 346, 506, 253, 734, 896, 681,
800, 261, 664, 896, 0, 774, 1138, 190, 732, 662,
     540, 883, 350],
    [232, 877, 1100, 298, 977, 134, 427, 547, 1289, 159, 128,
754, 660, 825, 162, 1097, 774, 0, 338, 987, 831, 112,
     575, 176, 568],
    [575, 1130, 1391, 671, 1488, 280, 806, 883, 1539, 413, 261,
999, 1009, 1149, 484, 1363, 1138, 338, 0, 1299,
     1065, 455, 984, 444, 951],
    [734, 213, 335, 690, 287, 1040, 478, 387, 333, 866, 1028,
556, 831, 141, 805, 652, 190, 987, 1299, 0, 576, 854,
     420, 1036, 608],
    [521, 376, 560, 624, 297, 798, 551, 225, 806, 869, 1141, 51,
171, 471, 834, 221, 732, 831, 1065, 576, 0, 641,
     351, 713, 691],
    [120, 765, 988, 185, 875, 246, 315, 435, 1177, 263, 240,
590, 548, 653, 193, 964, 662, 112, 455, 854, 641, 0,
     463, 190, 455],
    [343, 324, 547, 321, 405, 709, 190, 126, 706, 578, 740, 300,
420, 279, 508, 696, 540, 575, 984, 420, 351, 463,
     0, 660, 330],
    [312, 891, 1141, 389, 957, 143, 538, 637, 1292, 336, 278,
642, 515, 892, 331, 981, 883, 176, 444, 1036, 713,
     190, 660, 0, 695],
    [396, 672, 867, 271, 747, 701, 149, 363, 951, 949, 690, 640,
529, 477, 458, 1112, 350, 568, 951, 608, 691, 455,
     330, 695, 0]
]
```

Створіть список міст.

```
# Список міст

cities = [
    'Вінниця', 'Дніпро', 'Донецьк', 'Житомир', 'Запоріжжя',
'Івано-Франківськ', 'Київ', 'Кропивницький',
```

```
    'Луганськ', 'Луцьк', 'Львів', 'Миколаїв', 'Одеса',
'Полтава', 'Рівне', 'Сімферополь', 'Суми', 'Тернопіль',
    'Ужгород', 'Харків', 'Херсон', 'Хмельницький', 'Черкаси',
'Чернівці', 'Чернігів'
]
```

Створіть основну функцію програми.

```
if __name__ == '__main__':
    # Пошук відповіді задачі
    cityMap = CityMap(distance, len(distance[0]))
    colony = Colony(len(distance[0]))
    result = colony.find_route(cityMap, 6)
    print(f"Отриманий найкоротший шлях: {result[0]} км")
```

Створіть функцію для виведення на друк оптимального маршруту.

```
    # Вивід отриманого маршруту
    cityRoutes = "Отриманий маршрут: "
    for i in result[1]:
        cityRoutes += cities[i]
        if i != result[1][-1]:
            cityRoutes += "->"
    print(cityRoutes)
```

Побудуйте графічне відображення отриманих даних.

```
    # Графічне відображення отриманих даних
    fig = plt.figure(figsize=(13, 13))
    plt.xticks([i + 1 for i in range(25)])
    plt.yticks([i for i in range(25)], cities)
    plt.xlabel("Номери міст")
    plt.ylabel("Назви міст")
    plt.title("Маршрут пройдений комівояжером")
    plt.plot([i + 1 for i in range(25)], result[1], ms=12,
marker='*', mfc='r',
             mec='black', mew=2, color='black', ls="--")
    plt.grid()
    plt.show()
```

Увага!

Врахуйте, що старт та фініш мурахи повинен бути у вашому місті.

Запускайте декілька разів (не менше 5) вашу програму і ви отримаєте різні результати. Результати (мін. відстань та маршрут у текстовому вигляді) та рисунок з маршрутом копіюйте собі у проміжний документ. ***Потім з цього переліку оберіть оптимальний, там де відстань мінімальна. (Вона обов'язково понинна бути < 5000 км) та занесіть маршрут і рисунок у звіт.***

Збережіть код робочої програми з обов'язковими коментарям під назвою LR_7_task_4.py

Зробіть висновок. Висновок занесіть у звіт.

Коди комітити на GitHab. У кожному звіті повинно бути посилання на GitHab.

Назвіть бланк звіту ШІ-ЛР-7-NNN-XXXXX.doc
де NNN – позначення групи
XXXXX – позначення прізвища студента.

Переконвертуйте файл звіту в ШІ-ЛР-7-NNN-XXXXX.pdf
Надішліть чи представте звіт викладачу.