МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ
КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ МАТЕМАТИКИ

Звіт до лабораторної роботи №2 на тему «Генетичний алгоритм»

Виконав студент групи ОМ-3
Скибицький Нікіта

Київ — 2019

Зміст

1 Неформальний опис алгоритму

1.1 Власне опис

У генетичному алгоритмі популяція розв'язків-кандидатів (яких будемо називати особинами) оптимізаційної задачі еволюціонує до кращого розв'язку. Кожен кандидат характеризується множиною властивостей (це його хромосоми або генотип), які можуть мутувати. За традицією генотипи позначаються як бінарні рядки з 0 і 1, але інші кодування також можливі.

1.2 Історія винаходу методу

Перші спроби симуляції еволюції були проведені у 1954 році Нільсом Баричеллі на комп'ютері, встановленому в Інституті перспективних досліджень Принстонського університету. Його робота, що була опублікована у тому ж році, привернула увагу громадськості. З 1957 року, австралійський генетик Алекс Фразер опублікував серію робіт з симуляції штучного відбору серед організмів з множинним контролем вимірюваних характеристик. Це дозволило комп'ютерній

симуляції еволюційних процесів та методам, які були описані у книгах Фразера та Барнела(1970) та Кросбі(1975), з 1960-х років стати більш розповсюдженим видом діяльності серед біологів. Симуляції Фразера містили усі найважливіші елементи сучасних генетичних алгоритмів. До того ж, Ганс-Іоахім Бремерман в 1960-х опублікував серію робіт, які також приймали підхід використання популяції рішень, що піддаються відбору, мутації та рекомбінації, в проблемах оптимізації. Дослідження Бремермана також містили елементи сучасних генетичних алгоритмів. Також варто відмітити Річарда Фрідберга, Джоржа Фрідмана та Майкла Конрада.

Хоча Баричеллі у своїй роботі 1963 р. займався симуляцією можливості машини грати у просту гру, штучна еволюція стала загальновизнаним методом оптимізації після роботи Інго Рехенберга та Ханса-Пауля Шверeля у 60-х та на початку 70-х років двадцятого століття — група Рехенсберга змогла вирішити складні інженерні проблеми згідно зі стратегіями еволюції. Іншим підходом була техніка еволюційного програмування Лоренса Дж. Фогеля, яка була запропонована для створення штучного інтелекту. Еволюційне програмування, яке спочатку використовувало кінцеві автомати для передбачення обставин, та використовували різноманіття та відбір для оптимізації логіки передбачення. Генетичні алгоритми стали особливо відомі завдяки роботі Дж. Холанда на початку 70-х років та його книзі "Адаптація у природніх та штучних системах" (1975). Його дослідження були основані на експериментах з клітинними автоматами та на його роботах, що були написані в університеті Мічигану. Він ввів формалізований підхід для передбачення якості наступного покоління, відомий як Теорема схем. Дослідження в області генетичних алгоритмів залишалось більше теоретичним до середини 80-х років, коли була, нарешті, проведена Перша міжнародна конференція з генетичних алгоритмів (Піттсбург, Пенсильванія (США)).

З ростом дослідницького інтересу суттєво виросла обчислювальна потужність комп'ютерів, що дозволило використовувати нову обчислювальну техніку на практиці. Наприкінці 80-х років, компанія General Electric почала продаж першого в світі продукту, який працював з використанням генетичного алгоритму. Це був набір промислових обчислювальних засобів. В 1989 інша компанія Axcelis, Inc. випустила Evolver — перший у світі комерційний продукт на генетичному алгоритмі для персональних комп'ютерів. Журналіст The New York Times в технологічній сфері Джон Маркофф писав про Evolver у 1990 році.

1.3 Сфери застосування

Задачі до яких особливо часто застосовується генетичний алгоритм включають складання розкладів, інженерські задачі [1], розробка схеми сонячної електростанції [2], дизайн форми космічної антени [3], розробка штучного опорно-рухового апарату для роботів [4], і, звичайно, аеродинамічний дизайн [5].

1.4 Можливі модифікації

1.4.1 Елітаризм

Ця модифікація передбачає, що найкращі особини переходять до нового покоління без мутації. Вона гарантує монотонне незростанняя результату генетичного алгоритму, але ускладнює вихід з локальних оптимумів.

1.4.2 Паралельна реалізація

Цілком очевидно, що алгоритм гарно паралелиться, для цього кожному комп'ютері в обчислювальному кластері доручають моделювання якоїсь малої частини усієї популяції, які мутують поодинці, схрещуються всередині комп'ютера. Щодо генерації нового покоління то зазвичай раз на кілька ітерацій нове покоління генерується одним центральним комп'ютером на основі результатів всіх інших комп'ютерів, а не просто всередині кожного окремого комп'ютеру.

2 Формалізований опис алгоритму

Нехай є певна фітнес-функція $f\colon \mathbb{R}^m \to \mathbb{R}$ і ставиться оптимізаційна задача

$$f(x) \underset{x\in\mathcal{C}}{\longrightarrow} \min, \tag{2.1}$$

де $\mathcal{C} \subset \mathbb{R}^m$ — опукла і замкнена допустима множина, наприклад $\mathcal{C} = [x_{\min}, x_{\max}]^m$.

Розглянемо популяцію з n особин які протягом M поколінь розв'язують цю оптимізаційну задачу (пристосовуються під задану фітнес-функцію). Кожну особину будемо описувати двійковим вектором достатньо довгим для того щоб кодувати десяткові значення аргументів функції x_i з потрібною точністю ε.

Перше покоління не має досвіду предків і генерується випадковим чином, рівномірно на $\mathcal{C}$.

Далі з кожним поколінням відбуваються наступні речі:

- воно кодується з десяткового представлення у двійкове;
- у ньому із заданою ймовірністю p відбувають бітові мутації;
- покоління розбивається на пари, між якими відбувається кроссовер (обмін генами);
- на основі покоління генерується нове таким чином що найкращі особини мають більшу ймовірність мати нащадка.

3 Код програмного продукту

3.1 GenerationDec

Призначення

Ця процедура заповнює матрицю із заданою кількістю рядків і стовпчиків випадковими десятковими числами із заданого діапазону.

Діапазон значень фіксований для усіх елементів кожного стовпчика матриці.

Елементи останнього стовпчика матриці не заповнюються випадковими числами і призначені для обчислення значень фітнес-функції, аргументами якої є всі попередні елементи рядка матриці.

Вхідні параметри

- N, M — цілі невід'ємні числа
- $X_{\min}(1..M), X_{\max}(1..M)$ — масиви дійсних чисел, $X_{\min}[i] < X_{\max}[i]$, $i = \overline{1..M}$;

Вихідні параметри

- $G(1..N, 1..M+1)$ — матриця випадкових значень.

Обчислення

Заповнює елементи матриці $G[i,j]$, $i = \overline{1..N}$, $j = \overline{1..M}$, $X_{min}[j] \leq G[i,j] \leq X_{max}[j]$.

Вказівки

Для генерування випадкових чисел з заданого діапазону використовувати функцію Generate пакета Random Tools.

Власне реалізація

```
def __generation_dec(n: int, m: int, x_min: np.array, x_max: np.array) -> np.matrix:
    """
    :param n: num rows in returned matrix
    :param m: num cols in returned matrix
    :param x_min: float array, min possible nums in cols of returned matrix
    :param x_max: float array, max possible nums in cols of returned matrix
    :return: n times m float matrix with nums in col number i in [x_min[i], x_max[i])
    """
    assert n > 0, "n should be positive"
    assert m > 0, "m should be positive"

    assert x_min.shape == (m, ), "x_min should be of shape (m, )"
    assert x_max.shape == (m, ), "x_max should be of shape (m, )"

    return np.random.uniform(low=x_min, high=x_max, size=(n, m))


def generation_dec(n: int, x_min: np.array, x_max: np.array) -> np.matrix:
    """
    :param n: num rows to return
    :param x_min: float arr, min possible nums in cols to return
    :param x_max: float arr, max possible nums in cols to return
    :return: float mat with n rows and nums in col number i in [x_min[i], x_max[i])
    """
    assert n > 0, "n should be positive"

    assert x_min.ndim == 1, "x_min should have dimension 1"
    assert x_max.ndim == 1, "x_max should have dimension 1"
    assert x_min.shape == x_max.shape, "x_min and x_max shapes mismatch"
    m: int = len(x_min)

    assert np.all(x_min < x_max), "x_min should be element-wise less than x_max"
```

```
    return __generation_dec(n, m, x_min, x_max)
```

3.2 Mutation

Призначення

Опрацьовує в циклі кожен елемент прямокутної матриці заданого розміру з елементами 0 або 1 за наступним правилом:

- якщо згенероване випадкове число менше заданого порогового значення, то відповідний елемент матриці інвертується (0 в 1 або 1 в 0);
- інакше елемент матриці не змінюється.

Результат зберігається у новій матриці.

Вхідні параметри

- G — прямокутна матриця зі значеннями 0 або 1;
- p — дійсне число $0 < \mathsf{p} \ll 1$ — ймовірність мутації.

Вихідні параметри

- G_{mut} — прямокутна матриця зі значеннями 0 або 1, розмірність якої дорівнює розмірності G;
- S_{mut} — лічильних загальної кількості мутацій, що були виконані для матриці G.

Обчислення

- В циклі опрацювати кожен елемент матриці G, перевіряючи умову, що згенероване випадкове число менше заданого параметра p.
- При виконанні умови виконати мутацію, інакше елемент матриці не змінюється.
- При кожній мутації збільшувати лічильник на одиницю.

Вказівки

Для генерування випадкових чисел з заданого діапазону можна використовувати функцію Generate з пакету Random Tools.

Власне реалізація

```
def mutation(g: np.matrix, p: float) -> Tuple[np.matrix, int]:
        """
        :param g: bin mat to mutate
        :param p: prob of mutation
        :return:
                g_mut - mutated matrix
```

```
            s_mut - number of mutations
    """
    mask = np.random.uniform(size=g.shape) < p
    return g ^ mask, np.sum(np.sum(mask))
```

3.3 Crossover

Призначення

Процедура опрацьовує вхідну прямокутну матрицю з парною 2N кількістю рядків і довільною кількістю стовпчиків, елементами якої є числа 0 або 1.

Рядки матриці об'єднуються в пари, пари утворюються за значенням номерів рядків матриці, вказаними у двох списка розмірності N.

Кожна пара рядків матриці опрацьовується наступним чином:

- Генерується випадкове ціле число $1 \le cros \le L$ — кількість стовпчиків матриці.
- Пара рядків матриці, яка зараз опрацьовується, обмінюється між собою елементами стовпчиків з номерами від 1 до cros.
- Результати записуються у нову матрицю, розмірність якої дорівнює розмірності вхідної матриці.

Вхідні параметри

- N — ціле додатнє число;
- $G(1..2N, 1..L)$ — матриця з елементами 0 або 1;
- $M_{list}(1..N), F_{list}(1..N)$ — списки цілих чисел зі значеннями в інтервалі від 1 до 2N.

Вихідні параметри

- Матриця $G_{cros}(1..2N, 1..L)$ з елементами 0 або 1 як результат процедури кроссовера (схрещування).

Обчислення

- Організується цикл по елементам списків $M_{list}[i], F_{list}[i]$, $i = \overline{1..N}$.
- Послідовно опрацьовується кожна пара рядків $G[M_{list}[i], 1..L]$ и $G[F_{list}[i], 1..L]$.
- Для кожної такої пари рядків генерується випадкове число cros від 1 до L.
- Ці рядки обмінюються між собою елементами від 1 до cros.
- Результат змінених рядків записується у нову матрицю $G_{cros}(1..2N, 1..L)$.

Вказівки

- Для генерування випадкових чисел з заданого діапазону можна використовувати функцію Generate з пакету Random Tools.
- Для обчислення кількості рядків або стовпчиків матриці можна використовувати функції RowDimension і ColumnDimension з пакету LinearAlgebra.

Власне реалізація

```
def __crossover(n: int, g: np.matrix, m_list: np.array, f_list: np.array) -> np.matrix:
        """
        :param n: half of g.shape[0]
        :param g: bin mat of genes
        :param m_list: male nums
        :param f_list: female nums
        :return: crossed-over bin mat of genes
        """
        cros = np.random.randint(low=0, high=g.shape[1], size=n)

        g_cros = np.copy(g)

        for m, f, c in zip(m_list, f_list, cros):
                g_cros[[m, f], :c] = g_cros[[f, m], :c]

        return g_cros


def crossover(g: np.matrix, m_list: np.array, f_list: np.array) -> np.matrix:
        """
        :param g: bin mat of genes
        :param m_list: male nums
        :param f_list: female nums
        :return: crossed-over bin mat of genes
        """
        return __crossover(g.shape[0] >> 1, g, m_list, f_list)
```

3.4 Parents

Призначення

послідовність натуральних чисел від 1 до 2N розбивається на два списки натуральних чисел таким чином, що елементи кожгого списку вибираються випадковим чином з діапазону [1..2N] з умовою, що елементи списків з одинаковими номерами різні.

Вхідні параметри

- N — кількість елементів списків.

Вихідні параметри

- Списки $m_{list}(1..N), f_{list}(1..N)$.

Обчислення

- Організується циклічний процес по всім елементам списку.
- Елементам першого списку значення присвоють випадковим чином, а елементам другого списку випадковим чином, але з умовою, що $m_{list}[i] \neq f_{list}[i]$.
- При рівності робиться ще спроба присвоєння випадково числа до виконання умови.

Вказівки

Для генерування випадкових чисел з заданого діапазону можна використовувати функцію Generate з пакету Random Tools.

Власне реалізація

```
def parents(n: int) -> Tuple[np.array, np.array]:
        """
        :param n: half of nums to div into lists
        :return: two lists with permutation of [0, 2n)
        """
        p = np.random.permutation(np.arange(n << 1))
        return p[:n], p[n:]
```

3.5 BinDecParam

Призначення

Це допоміжна процедура для CodBinary і CodDecimal, обчислює параметри для прямого і зворотнього перетворення із заданою точністю із заданого інтервалу дійсних чисел у бінарну послідовність.

Вхідні параметри

- M — розмірності масивів.
- $X_{min}(1..M)$ — масив, де $X_{min}[j]$ — мінімальне значення діапазону j;
- $X_{max}(1..M)$ — масив, де $X_{max}[j]$ — максимальне значення діапазону j;
- ε — точність представлення десяткових чисел двійковим кодом.

Вихідні параметри

- $nn[i]$, $i = \overline{1..M}$ — список цілих чисел, містить значення кількості бінарних розрядів які необхідні для кодування довільного дійсного числа з діапазону $[X_{min}[i], X_{max}[i]]$ з точністю ε.

- $dd[i],\ i = \overline{1..M}$ — дійсне число, містить значення дискретності для представлення дійсного числа із заданого діапазону цілим числом.
- $NN[j+1] = \sum_{i=1}^{j} nn[i],\ j = \overline{1..M},\ NN[1] = 0$ — список цілих чисел.

Вказівки

Всі вихідні параметри описуються з атрибутом global.

Обчислення

$$nn[i] = \left\lceil \log_2 \left(\frac{X_{max}[i] - X_{min}[i]}{\varepsilon} \right) \right\rceil + 1, \quad i = \overline{1..M}, \tag{3.1}$$

$$dd[i] = \frac{X_{max}[i] - X_{min}[i]}{2^{nn[i]}} \leq \varepsilon, \quad i = \overline{1..M}, \tag{3.2}$$

$$NN[j+1] = \sum_{i=1}^{j} nn[i], \quad j = \overline{1..M}, \quad NN[1] = 0. \tag{3.3}$$

Власне реалізація

```
def __bin_dec_param(m: int, x_min: np.array, x_max: np.array,
                    eps: float) -> Tuple[np.array, np.array, np.array]:
    """
    :param m: dimen of x_min and x_max
    :param x_min: float array, min possible nums
    :param x_max: float array, max possible nums
    :param eps: desired precision
    :return:
        nn - min num bin digits to encode nums in [x_min, x_max) with precision eps
        dd - discretionaries of obtained encodings, < eps
        NN - prefix sums of nn
    """
    assert m > 0, "m should be positive"

    assert x_min.shape == (m, ), "x_min should be of shape (m, )"
    assert x_max.shape == (m, ), "x_max should be of shape (m, )"

    nn = (np.floor(np.log2((x_max - x_min) / eps)) + 1).astype(int)
    assert np.all(nn > 0), "desired precision should be smaller than |x_max - x_min|"

    dd = np.round((x_max - x_min) / (np.power(2, nn) - 1), 7)
    NN = np.array([0,] + list(accumulate(nn)))

    return nn, dd, NN


def bin_dec_param(x_min: np.array, x_max: np.array,
```

```
                eps: float) -> Tuple[np.array, np.array, np.array]:
    """
    :param x_min: float array, min possible nums
    :param x_max: float array, max possible nums
    :param eps: desired precision
    :return:
        nn - min num bin digits to encode nums in [x_min, x_max) with precision eps
        dd - discretionaries of obtained encodings, < eps
        NN - prefix sums of nn
    """
    assert x_min.ndim == 1, "x_min should have dimension 1"
    assert x_max.ndim == 1, "x_max should have dimension 1"
    assert x_min.shape == x_max.shape, "x_min and x_max shapes mismatch"
    m: int = len(x_min)

    return __bin_dec_param(m, x_min, x_max, eps)
```

3.6 CodBinary

Призначення

Процедура виконує кодування довільного дійсного числа $\mathrm{x}_{\mathrm{dec}}$ з заданого діапазону $[\mathrm{x}_{\min}..\mathrm{x}_{\max}]$ з заданою точністю ε у послідовність з 0 і 1 фіксованої довжини.

Процедура працює у парі з процедурою CodDecimal, яка виконує зворотнє перетворення.

Допоміжні параметри обчислюються процедурою BinDecParam.

Вхідні параметри

- $\mathrm{x}_{\mathrm{dec}}$ — десяткове число;
- $\mathrm{x}_{\min}$ — мінімальне значення числа, що кодується;
- l — ціле число, максимальна кількість двійкових розрядів для представлення довільного числа із заданого діапазону із заданою точністю;
- d — дискретність кодування дійсного числа $\mathrm{x}_{\mathrm{dec}}$ цілим числом.

Вихідні параметри

- $\mathrm{X}_{\mathrm{bin}}$ — список з l розрядів двійкового числа, молодші розряди йдуть спочатку.

У разі потреби старші розряди дозаповнюються нулями.

Обчислення

Ціле число частин величини d для заданого числа $\mathrm{x}_{\mathrm{dec}}$ можно обчислити як

$$\mathrm{xx} = \left[\frac{\mathrm{x}_{\mathrm{dec}} - \mathrm{x}_{\min}}{\mathrm{d}}\right]. \tag{3.4}$$

Ціле число xx записуємо у двійковій формі і доповнюємо старші розряди нулями, якщо їхня кількість менше l.

Вказівки

Значення l і d обчислюються процедурою BinDecParam і не можуть задаватися довільно.

Перетворення цілого десяткового числа у двійковий код (список 0 і 1) можна виконати функцією convert(xx, base, 2).

Власне реалізація

```
def __cod_binary(x_dec: float, x_min: float, l: int, d: float) -> np.array:
    """
    :param x_dec: dec num to encode
    :param x_min: min num possible to encode
    :param l: num of bin digits to return
    :param d: discretionary of encoding to return
    :return: encoded num as np.array of digits
    """
    assert l > 0, "num digits in encoding should be positive"
    assert d > 0, "discretionery should be positive"

    # either this assertion or the following assertion about xx should be removed
    if x_dec < x_min: x_dec = x_min
    if x_dec > x_min + d * ((1 << l) - 1): x_dec = x_min + d * ((1 << l) - 1)
    assert x_min <= x_dec <= x_min + d * 2**l, "x_dec cannot be encoded"
    xx = int(np.floor((x_dec - x_min) / d))
    assert xx <= 2**l - 1, "x_dec cannot be encoded"
    xx ^= xx >> 1

    digits = list(map(int, bin(xx)[:1:-1]))

    return np.array(digits + [0 for _ in range(l - len(digits))])
```

3.7 CodDecimal

Призначення

Процедура виконує перетворення закодованого послідовністю з 0 і 1 двійкового представлення числа з заданого діапазону $[x_{min}..x_{max}]$ із заданою точністю ε у його звичайне десяткове представлення.

Процедура працює в парі з процедурою CodBinary, яка виконує зворотнє перетворення.

Вхідні параметри

- x_{bin} — послідовність розрядів двійкового числа;
- x_{min} — мінімальне значення десяткового числа;
- d — дискретність кодування дійсного чиисла x_{dec} цілим числом.

Вихідні параметри

- x_{dec} — десяткове число.

Обчислення

- Послідовність розрядів x_{bin} перетворюється у ціле число x'_{dec}.
- Далі за допомогою цілого числа d відновлюємо десяткове число $x_{dec} = x_{min} + d \cdot x'_{dec}$.

Вказівки

Значення l і d обчислюються процедурою BinDecParam і не можуть задаватися довільно.

Перетворення двійкового коду (список 0 і 1) у послідовність десяткових розрядів цілого десяткового числа можна виконати функцією convert(xbin, base, 2, 10).

Власне реалізація

```
def __cod_decimal(x_bin: np.array, x_min: float, d: float) -> float:
    """
    :param x_bin: bin num to decode as np.array of digits
    :param x_min: min num possible to encode
    :param d: discretionary of encoded number
    :return: decoded num as float
    """
    assert d > 0, "discretionary should be positive"

    x_dec_1 = reduce(lambda _1, _2: _1 * 2 + _2, map(int, x_bin[::-1]))

    x_bin_1 = 0
    while x_dec_1 != 0:
        x_bin_1 ^= x_dec_1
        x_dec_1 >>= 1
    return x_min + d * x_bin_1
```

3.8 ACodBinary

Призначення

Послідовно перетворює дійсні числа, а саме елементи матриці $G_{dec}(1..N, 1..M+1)$, яка складається з N рядків і $M+1$ стовпчиків у двійковий код.

Перетворення виконуються над елементами тільки M перших стовпчиків.

Для перетворення кожного елемента матриці використовуються процедури CodBinary і BinDecParam.

Результати перетворення дійсних чисел зберігаються у матрицю Gbin з елементами 0 або 1.

Кожному стовпчику матриці G_{dec} відповідає фіксована кількість стовпчиків матриці G_{bin}, яка визначається діапазоном дійсних значень які кодуються і точністю їхнього представлення (див. процедуру CodBinary, параметр l).

Вхідні параметри

- N, M — розмірності матриці $G_{dec}(1..N, 1..M+1)$;
- Матриця $G_{dec}(1..N, 1..M+1)$;
- $X_{min}(1..M)$ — масив, де $X_{min}[j]$ — мінімальне значення елементів стовпчика j;
- Глобальні параметри процедури BinDecParam: nn, dd, NN.

Вихідні параметри

- Матриця G_{bin} з елементами 0 або 1, яка складається з N рядків, кількість стовпчиків матриці визначається значенням $NN \cdot (M+1)$.

Обчислення

В циклі опрацьовуємо кожний елемент матриці G_{dec}, використовуємо процедуру CodBinary, записуємо результати перетворення у матрицю G_{bin} розмірності $(1..N, 1..NN \cdot [M+1])$.

Вказівки

Використовувати для обробки кожного елемента вихідної матриці процедуру CodBinary.

Власне реалізація

```
def __a_cod_binary(n: int, m: int, g_dec: np.matrix, x_min: np.array,
                   nn: np.array, dd: np.array) -> np.matrix:
    """
    :param n: num rows in g_dec
    :param m: num cols in g_dec
    :param g_dec: dec nums matrix to encode
    :param x_min: min nums possible to encode
    :param nn: num bin digits to encode nums
    :param dd: discretionaries of encodings
    :return: encoded nums as np.matrix of np.arrays of digits
    """
    assert x_min.shape == (m, ), "x_min should be of shape (m, )"
    assert nn.shape == (m, ), "nn should be of shape (m, )"
    assert dd.shape == (m, ), "dd should be of shape (m, )"
    assert g_dec.shape == (n, m), "g_dec should be of shape (n, m)"

    return np.matrix([
        np.hstack([
            __cod_binary(g_dec[i, j], x_min[j], nn[j], dd[j]) for j in range(m)
```

```
        ]) for i in range(n)
    ])


def a_cod_binary(g_dec: np.matrix, x_min: np.array, nn: np.array,
                 dd: np.array) -> np.matrix:
    """
    :param g_dec: dec nums matrix to encode
    :param x_min: min nums possible to encode
    :param nn: num bin digits to encode nums
    :param dd: discretionaries of encodings
    :return: encoded nums as np.matrix of np.arrays of digits
    """
    assert g_dec.ndim == 2, f"g_dec should be a matrix (have dimension 2). " + \
                            f"got g_dec.ndim = {g_dec.ndim}, g_dec = {g_dec}"
    assert x_min.ndim == 1, "x_min should be an array (have dimension 1)"
    assert nn.ndim == 1, "nn should be an array (have dimension 1)"
    assert dd.ndim == 1, "dd should be an array (have dimension 1)"
    assert g_dec.shape[1] == x_min.shape[0], "g_dec and x_min shapes mismatch"
    assert g_dec.shape[1] == nn.shape[0], "g_dec and nn shapes mismatch"
    assert g_dec.shape[1] == dd.shape[0], "g_dec and dd shapes mismatch"
    n, m = g_dec.shape

    return __a_cod_binary(n, m, g_dec, x_min, nn, dd)
```

3.9 ACodDecimal

Призначення

Опрацьовує вміст матриці G_{bin} з елементами 0 або 1 використовуючи структуру матриці і перетворює послідовності з 0 і 1 заданих довжин у дійсні числа.

Вхідні параметри

- N, M — розмірності матриці $G_{dec}(1..N, 1..M+1)$;
- Матриця G_{bin};
- $X_{min}(1..M)$ — масив, де $X_{min}[j]$ — мінімальное значення елемениів стовпчика j;
- Глобальні параметри nn, dd, NN процедури BinDecParam.

Вихідні параметри

- Матриця $G_{dec}(1..N, 1..M+1)$.

Обчислення

- У циклі послідовно опрацьовуються всі рядки матриці G_{bin}.

- З кожного рядка вибираються M послідовностей з 0 і 1, які опрацьовує процедура CodDecimal.
- Отримані дійсні числа записуються у матрицю $G_{dec}(1..N, 1..M+1)$.

Власне реалізація

```
def __a_cod_decimal(n: int, m: int, g_bin: np.matrix, x_min: np.array, NN: np.array,
                    dd: np.array) -> np.matrix:
    """
    :param n: num rows in g_bin
    :param m: num cols in g_bin
    :param g_bin: bin nums as np.matrix of np.arrays of digigts to decode
    :param x_min: min nums possible to encode
    :param NN: prefix sums of num bin digits in encoded nums
    :param dd: discretionaries of encodings
    :return: decoded nums as np.matrix of floats
    """
    _m: int = NN.shape[0] - 1
    assert x_min.shape == (_m, ), "x_min should be of shape (_m, )"
    assert NN.shape == (_m + 1, ), "dd should be of shape (_m + 1, )"
    assert dd.shape == (_m, ), "dd should be of shape (_m, )"

    return np.matrix([[
        __cod_decimal(np.asarray(g_bin[i, NN[j]:NN[j + 1]]).reshape(-1), x_min[j], dd[j])
    for j in range(_m)] for i in range(n)])


def a_cod_decimal(g_bin: np.matrix, x_min: np.array, NN: np.array,
                  dd: np.array) -> np.matrix:
    """
    :param g_bin: bin nums as np.matrix of np.arrays of digigts to decode
    :param x_min: min nums possible to encode
    :param NN: prefix sums of num bin digits in encoded nums
    :param dd: discretionaries of encodings
    :return: decoded nums as np.matrix of floats
    """
    assert g_bin.ndim == 2, "g_bin should be a matrix (have dimension 2)"
    assert x_min.ndim == 1, "x_min should be an array (have dimension 1)"
    assert NN.ndim == 1, "NN should be an array (have dimension 1)"
    assert dd.ndim == 1, "dd should be an array (have dimension 1)"
    assert g_bin.shape[1] == NN[-1], "g_bin and NN shapes mismatch"
    assert NN.shape[0] == dd.shape[0] + 1, "NN and dd shapes mismatch"
    assert x_min.shape[0] == dd.shape[0], "x_min and dd shapes mismatch"
    n, m = g_bin.shape
```

```
return __a_cod_decimal(n, m, g_bin, x_min, NN, dd)
```

3.10 Adapt

Призначення

Генерирує масив N цілих невід'ємних чисел, сума яких дорівнює N.

Кожне значення з номером j масиву обчислюється як кількість потраплянь випадкового числа рівномірно розподіленого на відрізку $[0..1]$ у підінтервал з номером $1 \leq j \leq N$, що належить відрізну $[0..1]$.

Сума довжин всіх підінтервалів дорівнює одиниці.

Вхідні параметри

- p — послідовність N дійсних чисел;
- N — ціле додатнє число.

Вихідні параметри

- num — масив N цілих невід'ємних чисел, сума яких дорівнює N.

Обчислення

- Перетворюємо послідовнітсь p:
  - Знаходимо мінимальний елемент p.
  - Віднімаємо від кожного елемента p мінімальний елемент, утворюємо p_1.
- Знаходимо суму всіх елементів p_1;
- Ділимо кожен елемент p_1 на суму значень його елементів, утворюємо p_2.
- Утворюємо список префіксних сум p_2:

$$\text{adaptability} = [0, p_2[1], p_2[1] + p_2[2], \ldots, p_2[1] + p_2[2] + \ldots + p_2[N]]. \tag{3.5}$$

- N разів генеруємо випадкове число $0 \leq \text{roll} < 1$ і перевіряємо умову належності випадкового числа roll одному з діапазонів списку adaptability:
  - Якщо $\text{adaptability}[j] \leq \text{roll} < \text{adaptability}[j + 1]$, то $\text{num}[j] = \text{num}[j] + 1$.

Вказівки

Для генерування випадкових чисел з заданого діапазону можна використовувати функцію Generate з пакету Random Tools.

Власне реалізація

```
def __adapt(p: np.array, n: int) -> np.array:
        """
        :param p: n nums
        :param n: len p
        :return: adaptability
        """
        p -= np.min(p)

        p /= np.sum(p)

        num = np.zeros(n)

        for roll in np.random.choice(np.arange(n), n, p=p):
                num[roll] += 1

        return num.astype(int)


def adapt(p: np.array) -> np.array:
        """
        :param p: n nums
        :return: adaptability
        """
        n: int = len(p)

        return __adapt(p, n)
```

3.11 Best і Worst

Призначення

Процедури опрацьовують (M+1)-ий стовпчик матриці G_{dec}, який містить дійсні значення деякої функції від M аргументів, самі аргументи містять у стовпчиках з 1 по M.

Обчислення

Процедура Best обчислює номер рядка з мінімальним значенням функції, а процедура Worst — номер рядка з максимальним значенням функції.

Вхідні параметри

- G_{dec} — вищезгадана матриця.

Вихідні параметри

- Кожна з процедур повертає знайдене ціле число.

Власне реалізація

```
def __best(m: int, g_dec: np.matrix) -> int:
        """
        :param m: num args
        :param g_dec: args and func vals mat
        :return: argmax func val index
        """
        return np.argmax(g_dec[:, m])


def __worst(m: int, g_dec: np.matrix) -> int:
        """
        :param m: num args
        :param g_dec: args and func vals mat
        :return: argmin func val index
        """
        return np.argmin(g_dec[:, m])


def best(g_dec: np.matrix) -> int:
        """
        :param g_dec: args and func vals mat
        :return: argmax func val index
        """
        return __best(-1, g_dec)


def worst(g_dec: np.matrix) -> int:
        """
        :param g_dec: args and func vals mat
        :return: argmin func val index
        """
        return __worst(-1, g_dec)
```

3.12 NewGeneration

Призначення

Опрацьовує рядки вхідної матриці виходячи зі значень вхідного цілочисельного масиву з невід'ємними елементами (сума елементов цілочислного масиву дорівнює кількості рядків матриці).

Розмірність масива і кількість рядків матриці рівні.

Рядок з номером j вхідної матриці записується у вихідну матрицю таку кількість разів, яка дорівнює цілому значенню з номером j масива.

Якщо на позиції j знаходиться число 0, то рядок у вихідну матрицю не записується.

Вхідні параметри

- $G_{dec}(1..N, 1..M+1)$ — матриця;
- $num(1..N)$ — масив цілочисльних значень.

Вихідні параметри

- $G_{dec_{new}}(1..N, 1..M+1)$ — вихідна матриця.

Власне реалізація

```
def new_generation(g_dec: np.matrix, num: np.array) -> np.matrix:
        """
        :param g_dec: old generation mat
        :param num: repeats
        :return: new generation mat
        """
        return np.repeat(g_dec, repeats=num, axis=0)
```

3.13 Програма-драйвер

```
#!/usr/bin/env python
import numpy as np
from generation_dec import generation_dec
from bin_dec_param import bin_dec_param
from cod_binary import a_cod_binary
from cod_decimal import a_cod_decimal
from mutation import mutation
from parents import parents
from crossover import crossover
from adapt import adapt
from new_generation import new_generation
from best_worst import best, worst


np.set_printoptions(linewidth=100)
np.random.seed(65537)

n, _m = 1 << 10, 20
x_min, x_max = np.repeat(-2, _m), np.repeat(2, _m)
g_dec = generation_dec(n, x_min, x_max)
eps = 1e-5
nn, dd, NN = bin_dec_param(x_min, x_max, eps)
p = 2e-3

def rastrigin(x: np.array) -> float:
        return -10 * _m - np.sum(np.power(x, 2) - 10 * np.cos(2 * np.pi * x))
```

```
for _ in range(1, 1 << 10):
        g_bin = a_cod_binary(g_dec, x_min, nn, dd)
        g_bin, mutation_count = mutation(g_bin, p)
        m, f = parents(n >> 1)
        g_bin = crossover(g_bin, m, f)
        g_dec = a_cod_decimal(g_bin, x_min, NN, dd)
        f_vals = np.array([rastrigin(g_dec[i]) for i in range(n)]).reshape((n, 1))
        # if not (_ & 63):
        #         print(f'it {_:0>4}: {np.max(f_vals):>12.7f}')
        g_dec = np.hstack([g_dec, f_vals])
        b, w = best(g_dec), worst(g_dec)
        g_best = np.asarray(g_dec[b, :-1]).flatten()
        g_dec = np.delete(g_dec, w, axis=0)
        num = adapt(np.asarray(g_dec[:, -1]).flatten())
        g_dec = np.vstack([new_generation(g_dec[:, :-1], num), g_best])
```

4 Тестування програмного продукту

Окремо зауважимо, що для проводилося тестування окремих модулів за допомогою бібліотеки unittest.

Окрім цього, проводилося тестування цілісного алгоритму на багатовимірній (в $\mathbb{R}^{20}$) функції Растрігіна.

Сильною стороною програмного продукту виявилася **швидкодія**: 2^{10} ітерацій з розміром популяції 2^{10} особин зайняли близько 13 хвилин, що більш ніж **у 2.5 рази швидше** за наступний по швидкодії код здатний працювати на задачах такого масштабу (Олексія Кравця).

Окрім цього алгоритм демонструє непогану динаміку точності:

```
C:\Users\NikitaSkybytskyi\Desktop\gen\gen\code>main.py
it 0001: -134.2913119
it 0002: -125.0144292
it 0003: -117.6224736
it 0004:  -99.8343776
it 0005:  -95.0405274
it 0006:  -95.0399947
it 0007:  -81.9499555
it 0008:  -68.6859237
.....................
it 0050:  -13.8048245
.....................
it 0100:   -7.4214788
.....................
it 0200:   -1.9735074
```

```
.....................
it 0500:   -0.5820756
.....................
it 1000:   -0.3860838
```

Література

[1] B. Tomoiaga, M. Chindris, A. Sumper, A. Sudria-Andreu, and R. Villafafila-Robles, "Pareto optimal reconfiguration of power distribution systems using a genetic algorithm based on nsga-ii," *Energies*, vol. 6, no. 3, pp. 1439–1455, 2013. http://www.mdpi.com/1996-1073/6/3/1439/pdf.

[2] B. Gross, "A solar energy system that tracks the sun." https://www.ted.com/talks/bill_gross_on_new_energy, 2013.

[3] G. S. Hornby, D. S. Linden, and J. D. Lohn, "Automated antenna design with evolutionary algorithms." https://ti.arc.nasa.gov/m/pub-archive/1244h/1244%20(Hornby).pdf.

[4] "Flexible muscle-based locomotion for bipedal creatures." https://www.goatstream.com/research/papers/SA2013/index.html.

[5] B. Evans and S. P. Walton, "Aerodynamic optimisation of a hypersonic reentry vehicle based on solution of the boltzmann–bgk equation and evolutionary optimisation," *Applied Mathematical Modelling*, vol. 52, pp. 215–240, 2017. https://cronfa.swan.ac.uk/Record/cronfa34688.