Санкт-Петербургский государственный университет
Факультет прикладной математики и процессов управления

Иследование принципов работы генетического алгоритма

Лабараторная работа №2

Аннотация

В данной работе представлено описание принципов работы генетического алгоритма, подкреплённое програмной и графической реализацией.

Ключевые слова: Генетический алгоритм, Python, Tkinter

Автор работы: Шайдуров В.Д.
Группа: 21.Б15-пу
Научный руководитель: Дик А.Г.

Санкт-Петербург
2023 г.

Содержание

1 Вступление

Генетический алгоритм – это эвристический метод оптимизации, который основывается на имитации процессов естественного отбора и генетической рекомбинации в биологической эволюции.

Он широко используется в области оптимизации, где требуется поиск оптимального решения при условии ограничений, а также может быть использован для решения задач различных типов, таких как распределение ресурсов, планирование производства или оптимизации функций в задачах машинного обучения.

Главной идеей генетического алгоритма является создание популяции решений с использованием случайного выбора и последующая их модификация и отбор на основе вычисления функции приспособленности. Алгоритм способен поддерживать разнообразие в популяции, что позволяет ему избегать преждевременной сходимости к локальному оптимуму и обеспечивать достижение глобального оптимума задачи.

За счет своей эффективности, гибкости и возможности работать с большим количеством переменных, генетический алгоритм остается популярным методом оптимизации и по сей день.

2 Цель работы

Иследовать принципы работы генетического алгоритма.

3 Задача

Оценить эффективность генетического алгоритма и реализовать его визуализацию.

4 Описание програмы

В данном разделе приводиться описание кода на уровне идеи. Для более подробного понимания, рекомендуется самостоятельно ознакомиться с кодом по ссылке (github). Каждое дейсвие в программе сопровожденно коментариями, если возникают вопросы по терменалогии или по общему устройству программы обращайтесь к этому разделу.

4.1 Теоретические сведения

Необходимые термины:

Индивид - есть наименьшая еденица в поколении. Может обладать свойствами, такими как значения (x, y, F(x, y)) или другими вводными данными необходимые для мониторига состаяния одного конкретного индивида.

Популяция - это множество индивидов, каждый из которых представляет собой потенциальное решение задачи. Каждый индивид в популяции имеет свою генетическую информацию, которая описывает его свойства и характеристики.

Поколение - это новая генерация индивидов, получаемая из предыдущей популяции с помощью таких операций, как скрещивание, мутация и отбор. Каждый новый индивид в поколении образуется из предыдущих индивидов путем комбинации и изменения их генетической информации.

Скрещивание - функция которая отвечает, за передачу признаков родителей потомкам.

Мутация - функция незначительно изменяющая потомков после скрешевания за определённое количесво итераций.

Отбор - есть отбор лучших потомков для следующего поколения.

4.2 Описание програмной структуры генетического алгоритма

Програмная реализация написана на языке python 3.10, с использованием популярных и общеупотребимых пакетов: os, matplotlib, pandas, imageio, random, и tkinter.

В Таблице 1 предтсавленно описание сожердимого классов.

class Individual	class Genetic
calculateFunction	startGenetic
mutate	crossover

Таблица 1: Классы

В Листиге 1 и Листенге 2 привёдн испольняемый код для классов Individual и Genetic соответсвенно.

```python
class Individual:
    """ Класс одного индивида в популяции """
    def __init__(self, start, end, mutationSteps, function):
        # пределы поиска минимума
        self.start = start
        self.end = end
        self.x = random.triangular(self.start, self.end, mode=(self.start+self.end)/2)  # позиция индивида по X
        self.y = random.triangular(self.start, self.end, mode=(self.start+self.end)/2)  # позиция индивида по Y
        self.score = 0  # значение функции, которую реализует индивид
        self.function = function  # передаем саму функцию
        self.mutationSteps = mutationSteps  # количество шагов мутации
        self.calculateFunction()  # считаем сразу значение функции

    def calculateFunction(self) -> None:
        """ Функция для подсчёта значения нашей функции в индивиде """
        self.score = self.function(self.x, self.y)

    def mutate(self):
        """ Функция для мутации индивида """
        def mutation_rule(p):
            """ Функция описывающая шаги мутации """
            # задаем отклонение по (X, Y)
            delta = 0
            for i in range(1, self.mutationSteps + 1):
                if random.random() < 1 / self.mutationSteps:
                    delta += 1 / (2 ** i)
            if random.randint(0, 1):
                delta = self.end * delta
            else:
                delta = self.start * delta
            p += delta
            # ограничим наших индивидов по (X, Y)
            if p < 0:
                p = max(p, self.start)
            else:
                p = min(p, self.end)
            return p
        # отклонение по x
        self.x = mutation_rule(self.x)
        # отклонение по Y
        self.y = mutation_rule(self.y)
        # пересчитываем значение функции после мутации (x, y)
        self.calculateFunction()
```

Listing 1: class Individual

```python
class Genetic:
    """ Класс, отвечающий за реализацию генетического алгоритма """
    def __init__(self,
                 numberOfIndividuals,  # размер популяции в одном поколении
                 crossoverRate,  # какая часть популяции должна производить потомство
                 mutationSteps,  # количество шагов мутации для одной особи
                 chanceMutations,  # шанс особи на мутацию
                 numberGeneration,  # количество поколений
                 function,  # функция для поиска минимума
                 start, end):  # область поиска

        self.numberOfIndividuals = numberOfIndividuals
        self.crossoverRate = crossoverRate
        self.mutationSteps = mutationSteps
        self.chanceMutations = chanceMutations
        self.numberGeneration = numberGeneration
        self.function = function
        self.start = start
        self.end = end
        # самое минимальное значение, которое было в нашей популяции (в начале присваиваем infinite)
        self.bestScore = float('inf')
        # точка X, Y, где нашли минимальное значение (в начале присваиваем infinite)
        self.xy = [float('inf'), float('inf')]

    def crossover(self, parent1: Individual, parent2: Individual) -> [Individual, Individual]:
        """ Функция для скрещивания двух родителей получаем 2 потомка """
        # создаем 2-х новых детей
        child1 = Individual(self.start, self.end, self.mutationSteps, self.function)
        child2 = Individual(self.start, self.end, self.mutationSteps, self.function)
        # создаем новые координаты для детей
        child1.x, child1.y = parent1.x, parent2.y
        child2.x, child2.y = parent2.x, parent1.y
        return child1, child2

    def startGenetic(self):
        # создаем стартовую популяцию
        pack = [self.start, self.end, self.mutationSteps, self.function]
        population = [Individual(*pack) for _ in range(self.numberOfIndividuals)]
        # запускаем алгоритм
        for _ in range(self.numberGeneration):
            # сортируем популяцию по значению score
            population = sorted(population, key=lambda item: item.score)
            # берем лучший % индивидов, которых будем скрещивать между собой
            bestPopulation = population[:int(self.numberOfIndividuals * self.crossoverRate)]
            # теперь проводим скрещивание столько раз, сколько было задано по коэффициенту
            ↪  кроссовера
            children = []
            for _ in range(len(bestPopulation)):
                # находим случайную пару для каждого индивида и скрещиваем
                individual_mom = random.choice(bestPopulation)
                individual_dad = random.choice(bestPopulation)
                child1, child2 = self.crossover(individual_mom, individual_dad)
                children.append(child1)
                children.append(child2)
            # добавляем всех потомков в нашу популяцию
            population.extend(children)
            # проводим мутации для каждого индивида с вероятностью chanceMutations
            for individual in population:
                if random.choices((0, 1), weights=[1 - self.chanceMutations,
                ↪  self.chanceMutations])[0]:
                    individual.mutate()
            # отбираем лучших индивидов после мутации
            population = sorted(population, key=lambda item: item.score)
            population = population[:self.numberOfIndividuals]
            # запомним лучшего индивида который имеет наилучшее значение экстремума
            if population[0].score < self.bestScore:
                # Мы это делаем потому что все данные мутируют в каждом поколении
                self.bestScore = population[0].score
                self.xy = [population[0].x, population[0].y]
```

Listing 2: class Genetic

В Таблице 2 предтсавленно описание работы функций.

Список функций			
calculateFunction	startGenetic	mutate	crossover
Функция отвечает за вычисление значения нашей целевой функции, которую мы собираемся иследовать на экстремум.	Функция запускает цикл генерации новых поколений, каждое из которых основывается на предшествующем поколении. *Важно подметить что первое поколение создаётся произвольно, т.е. точки беруться случайным образом из заданного диапазона.	Мутирующая функция, которая изменяет нового индивида по определённому правилу.	Скрещивающая функция отвечает за создание новых индивидов, получаемых путём скрещивания двух родителей из предшествующего поколения

Таблица 2: Функции

4.3 Общий ход программы

1. Вызывается класс *class Genetic* в который передаётся, по усмотрению пользователя: **размер популяции для одного поколения, какая часть популяции сможет создать новых индивидов в следующем поколении, количесвто шагов мутирующей функции, само количесво поколений** и **иследуемая функция** с **диапазоном,** на котором будет искаться экстремум.

2. Далее через созданный класс обращаемся к функции *startGenetic*, которая в начаеле создаёт стартовую **популяцию** состоящую из объектов *class Individual* => получим **массив индивидов** - это будет нашим первым **поколением**.

3. В дальшейшем генерируется новая **популяция**, путём скрещевания двух случайных индивидов из предыдущего **поколения** с заданной пользователем вероятностью (вызов функции *crossover*), т.е. не какждая пара индивидов может создать новую пару индивидов.

4. Притом, если индивиды создали новую пару, то для неё вызывается функция *mutate*, которая изменяет только что сгенерированных индивидов за **несколько шагов мутирующей функции**.

5. Далее окончательно ново-образованные индивиды добаляются в **массив индивидов** где происходит сортировка по наименьшему/-наибольшему значению функции *calculateFunction*.

6. Для последующего поколения **массив индивидов** обрезается до изночального **размера популяции**, и шаги [3-6] повторяются заданное пользователем **количесво поколений**.

7. В конечном счёте выводиться индивид с наименьшим/наибольшим заначением исследуемой функции.

4.4 Теоретическое описание функций *mutate* и *crossover*

1. Функция *crossover* - присваивает первомому ребёнку координату по X от первого родителя и по Y от второго, а для второго ребёнка по X от второго родителя и по Y от первого. Таким образом каждый ребёнок несёт в себе информацию от двух родителей одномерменно.

2. Функция *mutate* - формирует координаты индивида по следующему правилу:

   $alfa = \sum_{i=1}^{n} \frac{1}{2^i} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots$, *где n=количесвто шагов мутирующей функции.*

   Приведенная геометриеская прогрессия на n устремлённом к бесконечности сходиться в еденицу, из-за чего мы можем горантировать, что приращение по коорinатам (X**alfa* + *alfa*; Y**alfa* + *alfa*) будет достаточно мало, чтобы не выходть за пределы по X: [X - *alfa*, X + *alfa*] и по Y: [Y - *alfa*, Y + *alfa*].

Примечание: важно понимать, что опредеить функции мутации и скрещивания можно по разному, что будет существенно влиять на точность работы алгоритма.

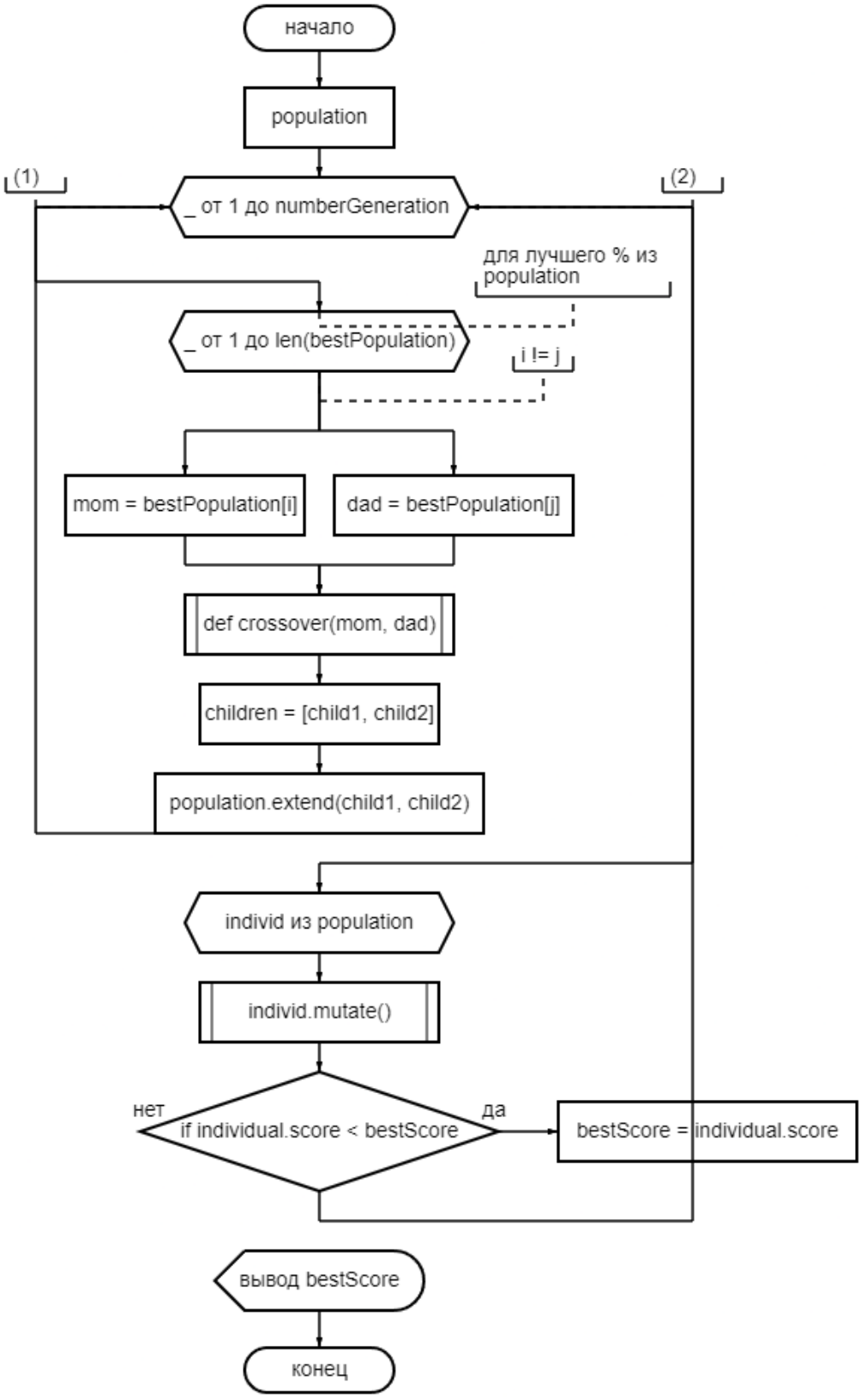

Рис. 1: Блоксхема программы.

5 Визуализация работы программы

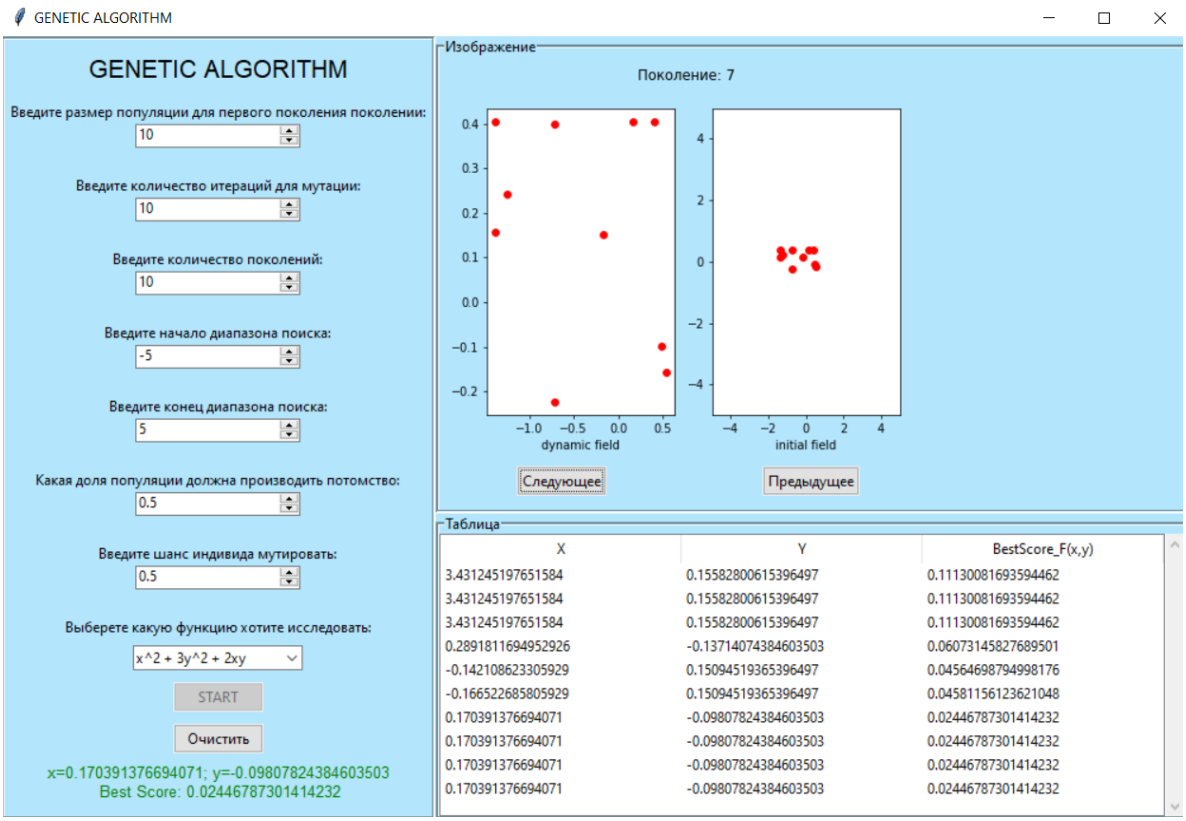

Рис. 2: Окно пользователя.

Устройсво приложения:

В левой части окна можно вводить параметры влияющие на систему, такие как: **размер популяции**, **количество шагов мутирующей функции**, **количество поколений**, **начало и конец деапазона**, **доля индивидов способных создать следующее поколение и шанс индивида мутировать** - а так же выбрать саму **иследуемую функциию** из списка.

В верху с права отображаются популяция в каждом поколении, в двух разных масштабах, один из которых денамических, а второй фиксированный.

Внизу с права таблица в которой записаны лучшие индивиды в каждом поколении.

6 Оценка эффективности алгоритма

Генетический алгоритм использует эврестический подход поиска из-за чего, мы не можем определить точное число шагов, требуемое для заданой точности. Более того исход его работы зависит от многих параметров, очевидных: таких как весь представленный набор переменных вводимых пользрвателем, так и не очевидных: таких как мутирующая и скрещивающая функции, а так же способа выбора случайных точек для первого поколения из задонного диапазона. Так же следует отметить, что скорость схождения к экстремуму будет напрямую зависить и от целевой функции.
Тем не менее я привожу результаты работы программы [Стр.10] при изменении одного из параметров, на ваше суждение, для функции из своего варианта данной лабараторной работы: $(x-2)^4+(x-2y)^2$ с минимумом в точке (2; 1). За базовый набор переменных будем брать значения с Рис.1.

6.1 Промежуточный вывод

Из ниже приведённых выклодок [Стр.10] можно подвести, что не всегда с увеличением одного из параметров мы добиваемся устойчивого роста повышения точности алгоритма. Так на на примере Таблицы 5 можно заметить что наименьшее значение функции достигается при 50 поколениях, тогда как при 100 поколениях точность падает на порядок, это связано с случайным выбором точек для первого поколения и случайным выбором радителя для нового индивида. Или на примере изменения количесва итераций для мутирующей функции Таблица 4, где лучшее значение достигается при 20 шагах. К сожалению такая тэнденция наблюдается и в других параметрах, из-за чего алгоритм становиться крайне не стабильным и требующим наблюдения и курирования человека в достижении лучшей своей конфигурации. Также необходимо понимать что изменение одного из параметров влияет на остальные, из-за чего не возможно точно подобрать их лучшую комбинацию. Но даже если у вас получиться это сделать, формула всё равно будет не универсальной, так как будет зацикленна на нахождение экстремума одной целевой функции.

Зависимость тосности алгоритма от размера популяции					
10	20	50	100	300	500
x=1.83794593 y=0.88509480 Best Score: 0.00528058	x=1.70068896 y=0.82480850 Best Score: 0.01063419	x=2.01097653 y=1.00301300 Best Score: 2.452226e-05	x=1.92503444 y=0.95953800 Best Score: 6.708545e-05	x=1.99050738 y=0.99156457 Best Score: 5.444656e-05	x=2.05291077 y=1.02940125 Best Score: 4.254998e-05

Таблица 3: Изменение популяции

Зависимость тосности алгоритма от кол-ва итераций мутирующей функции					
1	5	10	20	50	100
x=2.14910848 y=1.04814696 Best Score: 0.00328369	x=2.15113094 y=1.08653861 Best Score: 0.00100333	x=2.06216592 y=0.99435123 Best Score: 0.00541181	x=2.03934887 y=1.01587153 Best Score: 6.0245675-05	x=2.21723930 y=1.45280073 Best Score: 0.10454824	x=1.52515811 y=0.58768976 Best Score: 0.17318395

Таблица 4: Изменение кол-ва итераций для мутирующей функции

Зависимость тосности алгоритма от кол-ва поколений					
5	10	20	50	100	500
x=1.67958512 y=0.77591499 Best Score: 0.02686161	x=1.35275931 y=0.69077175 Best Score: 0.17632292	x=1.93283508 y=0.99881331 Best Score: 0.00421829	x=2.00920766 y=1.00388808 Best Score: 2.056368e-06	x=2.04626132 y=1.02028406 Best Score: 3.699251e-05	x=1.99490853 y=0.99520327 Best Score: 2.026859e-05

Таблица 5: Изменение кол-ва поколений

Зависимость тосности алгоритма от доли популяции способной производить потомство					
0.1	0.2	0.3	0.5	0.8	1.0
x=2.20535088 y=0.86315504 Best Score: 0.00165937	x=2.15357466 y=1.10211059 Best Score: 0.00312133	x=1.29163268 y=0.63698831 Best Score: 0.25209916	x=1.98906256 y=1.01059470 Best Score: 0.00103214	x=2.20785420 y=1.20450319 Best Score: 0.04232873	x=2.40796175 y=1.13640539 Best Score: 0.04596565

Таблица 6: Изменение доли популяции способной производить потомство

Зависимость тосности алгоритма от шанса индивида мутировать					
0.1	0.2	0.3	0.5	0.8	1.0
x=3.06274319 y=1.60176176 Best Score: 0.06402551	x=2.61288021 y=1.65774832 Best Score: 0.07280239	x=0.83462109 y=1.03726337 Best Score: 0.32253898	x=2.43586137 y=1.18895045 Best Score: 0.03944997	x=1.80078849 y=0.91655540 Best Score: 0.00261964	x=1.87169503 y=0.92920400 Best Score: 0.00044754

Таблица 7: Изменение шанса на мутацию

Заключение

В данной работе был расмотрен генетический алгоритм на нахождение минимума. Были выявленны проблемы, из-за которых его нельзя назвать лучшим для этой цели. Была представленна его прогрманая визуализация и теоретическое растолкование.

Список литературы

[1] Даниил Горбенко @daniilgorbenko, полезная статья на Habr, 2021.

[2] Т.В. Панченко, Генетические алгоритмы, Издательский дом «Астраханский университет», 2007