Санкт-Петербургский государственный университет
Факультет прикладной математики и процессов управления

Иследование принципов работы алгоритма роя частиц

Лабараторная работа №3

Аннотация

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

Ключевые слова: Алгоритм роя частиц, Python, Tkinter

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

Санкт-Петербург
2023 г.

Содержание

1 Вступление

В последние годы метод роя частиц стал широко применяться в различных областях науки, включая физику, информатику, биологию и многие другие. В основе этого метода лежит симуляция процессов на основе взаимодействия множества небольших объектов, называемых "частицами".

2 Цель работы

Целью данной лабораторной работы является изучение основных принципов и методов роя частиц, а также их применение для решения задачи поиска минимума целевой функции.

3 Задача

Оценить эффективность роевого 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:

Частица - есть наименьшая еденица в колонии, обладающий определенными свойствами, например, массой, положением и скоростью.

Популяция - это множество частиц, каждая из которых представляет собой потенциальное решение задачи. Каждая частица в популяции имеет свою информацию, которая описывает её свойства и характеристики.

Поколение - это новая генерация частиц, получаемая из предыдущей популяции с помощью взаимодейсвия чатиц друг с другом.

Текущий коэффициент скорости - масштабирующий коэффициент, который характеризует влияние собственной скорости особи.

Локальный коэффициент скорости - масштабирующий коэффициент, который характерицует влияние скорости лучшей точки в поколении на остальные точки в этом поколении.

Глобальный коэффициент скорости - масштабирующий коэффициент, который характерицует влияние скорости лучшей точки по всем поколениям на остальные точки в рое.

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 Unit	class Swarm
nextIteration	startSwarm
-	createSwarm

Таблица 1: Классы

В Листиге 1 и Листенге 2 привёдн испольняемый код для классов Swarm и Unit соответсвенно.

```python
class Swarm:
    """ Класс реализующий метод роя """
    def __init__(self,
                 sizeSwarm,  # размер роя (количество особей)
                 currentVelocityRatio,  # общий масштабирующий коэффициент для скорости
                 localVelocityRatio,  # коэффициент, задающий влияние лучшей точки особи
                 globalVelocityRatio,  # коэффициент, задающий влияние лучшей точки, найденной всеми особями
                 numberGeneration,  # количество поколений алгоритма (критерий остановки)
                 function,  # функция для поиска экстремума
                 start, end):  # область поиска

        self.sizeSwarm = sizeSwarm
        self.currentVelocityRatio = currentVelocityRatio
        self.localVelocityRatio = localVelocityRatio
        self.globalVelocityRatio = globalVelocityRatio
        self.numberGeneration = numberGeneration
        self.function = function
        self.start = start
        self.end = end

        # данные о лучшей позиции
        self.globalBestPos = []
        self.globalBestScore = float('inf')
        # рой частиц
        self.swarm = []
        self.createSwarm()

    def createSwarm(self) -> None:
        """ Метод для создания роя, вызывается 1 раз в начале"""
        pack = [self.start, self.end, self.currentVelocityRatio, self.localVelocityRatio, self.globalVelocityRatio,
                self.function]
        self.swarm = [Unit(*pack) for _ in range(self.sizeSwarm)]
        # выбираем лучшее значение для только что созданного роя
        for unit in self.swarm:
            if unit.localBestScore < self.globalBestScore:
                self.globalBestScore = unit.localBestScore
                self.globalBestPos = unit.localBestPos

    def startSwarm(self) -> None:
        """ Метод для запуска алгоритма"""
        for _ in range(self.numberGeneration):
            for unit in self.swarm:
                unit.globalBestPos = self.globalBestPos
                unit.nextIteration()
                if unit.score < self.globalBestScore:
                    self.globalBestScore = unit.score
                    self.globalBestPos = unit.localBestPos
```

Listing 1: class Swarm

```python
class Unit:
    """ Класс ::Пчела """
    def __init__(self, start, end, currentVelocityRatio, localVelocityRatio, globalVelocityRatio, function):
        # область поиска
        self.start = start
        self.end = end
        # коэффициенты для изменения скорости
        self.currentVelocityRatio = currentVelocityRatio
        self.localVelocityRatio = localVelocityRatio
        self.globalVelocityRatio = globalVelocityRatio
        # целевая функция
        self.function = function
        # текущая позиция (первый раз определяется случайно на заданном диапазоне)
        self.currentPos = [random.uniform(self.start, self.end), random.uniform(self.start, self.end)]
        self.score = self.function(*self.currentPos)
        # лучшая локальная позиция
        self.localBestPos = self.currentPos[:]
        self.localBestScore = self.score
        # значение глобальной позиции
        self.globalBestPos = []
        # скорость (первый раз задаётся случайно на диапазоне)
        search_range = abs(self.end - self.start)
        self.velocity = [random.uniform(-search_range, search_range), random.uniform(-search_range, search_range)]

    def nextIteration(self) -> None:
        """ Метод для нахождения новой позиции частицы"""
        # случайные данные для изменения скорости
        rndCurrentBestPosition = [random.random(), random.random()]
        rndGlobalBestPosition = [random.random(), random.random()]
        # делаем перерасчет скорости частицы исходя из всех введенных параметров
        velocityRatio = self.localVelocityRatio + self.globalVelocityRatio
        commonVelocityRatio = 2 * self.currentVelocityRatio / abs(
            2 - velocityRatio - sqrt(abs(velocityRatio ** 2 - 4 * velocityRatio)))

        # изменяем x и y
        multLocal = list(map(lambda x: x * commonVelocityRatio * self.localVelocityRatio, rndCurrentBestPosition))
        multGlobal = list(map(lambda x: x * commonVelocityRatio * self.globalVelocityRatio, rndGlobalBestPosition))

        betweenLocalAndCurPos = [self.localBestPos[0] - self.currentPos[0], self.localBestPos[1] - self.currentPos[1]]
        betweenGlobalAndCurPos = [self.globalBestPos[0] - self.currentPos[0],
                                  self.globalBestPos[1] - self.currentPos[1]]

        newVelocity1 = list(map(lambda x: x * commonVelocityRatio, self.velocity))
        newVelocity2 = [coord1 * coord2 for coord1, coord2 in zip(multLocal, betweenLocalAndCurPos)]
        newVelocity3 = [coord1 * coord2 for coord1, coord2 in zip(multGlobal, betweenGlobalAndCurPos)]
        self.velocity = [coord1 + coord2 + coord3 for coord1, coord2, coord3 in
                         zip(newVelocity1, newVelocity2, newVelocity3)]

        # передвигаем частицу и смотрим, какое значение целевой функции получается
        self.currentPos = [coord1 + coord2 for coord1, coord2 in zip(self.currentPos, self.velocity)]
        self.score = self.function(*self.currentPos)
        if self.score < self.localBestScore:
            self.localBestPos = self.currentPos[:]
            self.localBestScore = self.score
```

Listing 2: class Unit

В Таблице 2 предтсавленно описание работы функций.

Список функций		
createSwarm	startSwarm	nextIteration
Функция отвечает за создания роя.	Функция запускает цикл генерации новых поколений, каждое из которых основывается на предшествующем поколении.	Функция в которой прописывается правило изменение особи в каждом поколении.

Таблица 2: Функции

4.3 Общий ход программы

1. Вызывается класс *class Swarm* в который передаётся, по усмотрению пользователя: **размер популяции роя, количество поколений, текущий, локальный и глабальный коэффециенты для скорости** а также сама **иследуемая функция** с **диапазоном,** на котором будет искаться экстремум.

2. Далее через созданный класс обращаемся к функции *createSwarm*, которая в начаеле создаёт стартовую **популяцию** состоящую из объектов *class Unit* => получим **массив частиц**(пчёл) - это будет нашим первым **поколением**.

3. В дальшейшем генерируются новые **поколения**, путём вызова функции *startSwarm*.

4. Притом, в каждом поколении для каждого элемента списка объектов *class Unit* вызывается метод nextIteration, который изменяет исходные координаты частицы с какой-то псевдо случайной скоростью. Именно введенные выше коэффециенты для скорости влияют на скорость изменения координат частицы.

5. Далее шаги [3-4] вовторяются введенное количество поколений раз, где на каждом шаге запоминается лучшее значение функции в поколении, а так же каждый раз переопределяется глобальный минимум функции, если это необходимо.

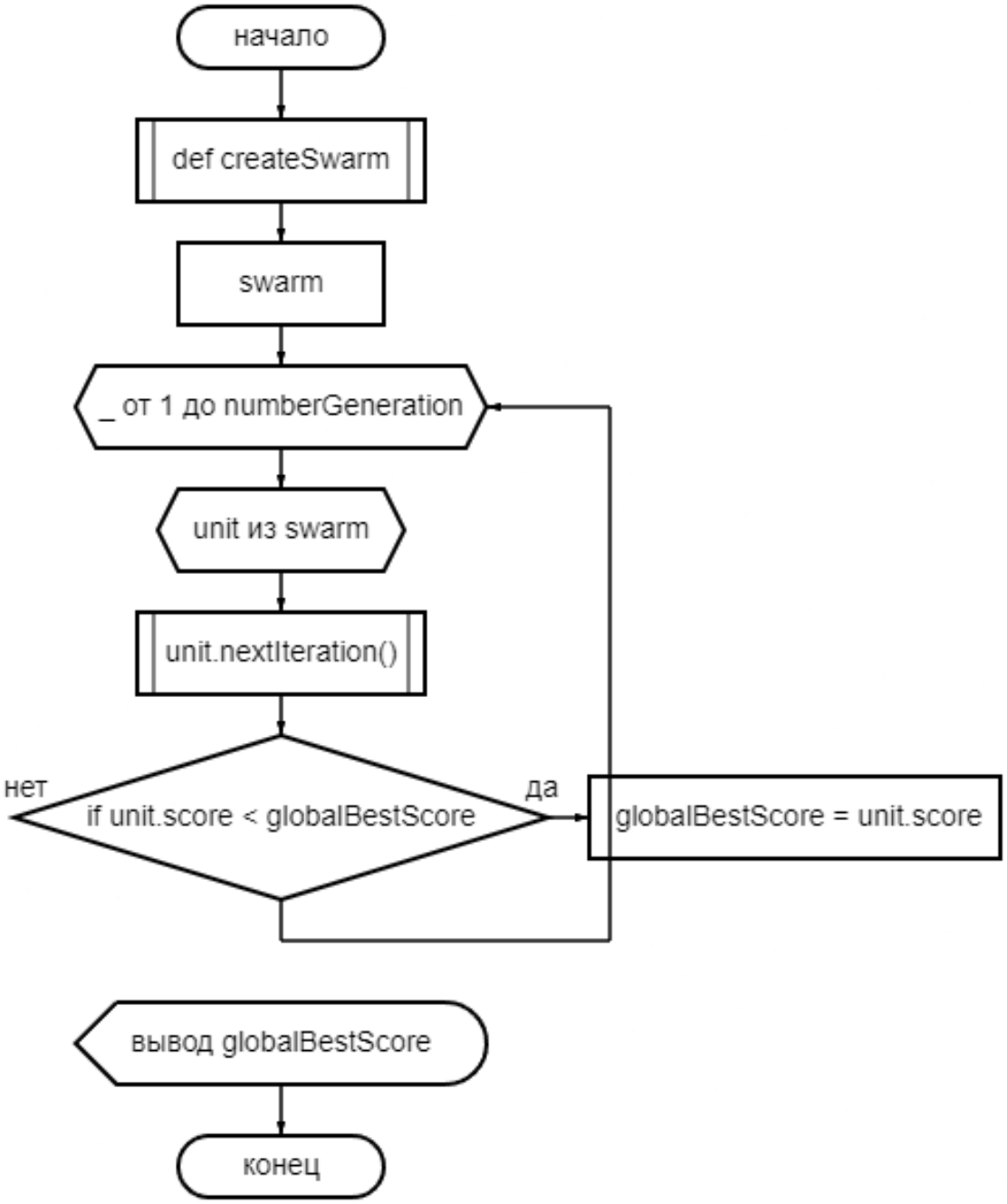

Рис. 1: Блоксхема программы.

5 Визуализация работы программы

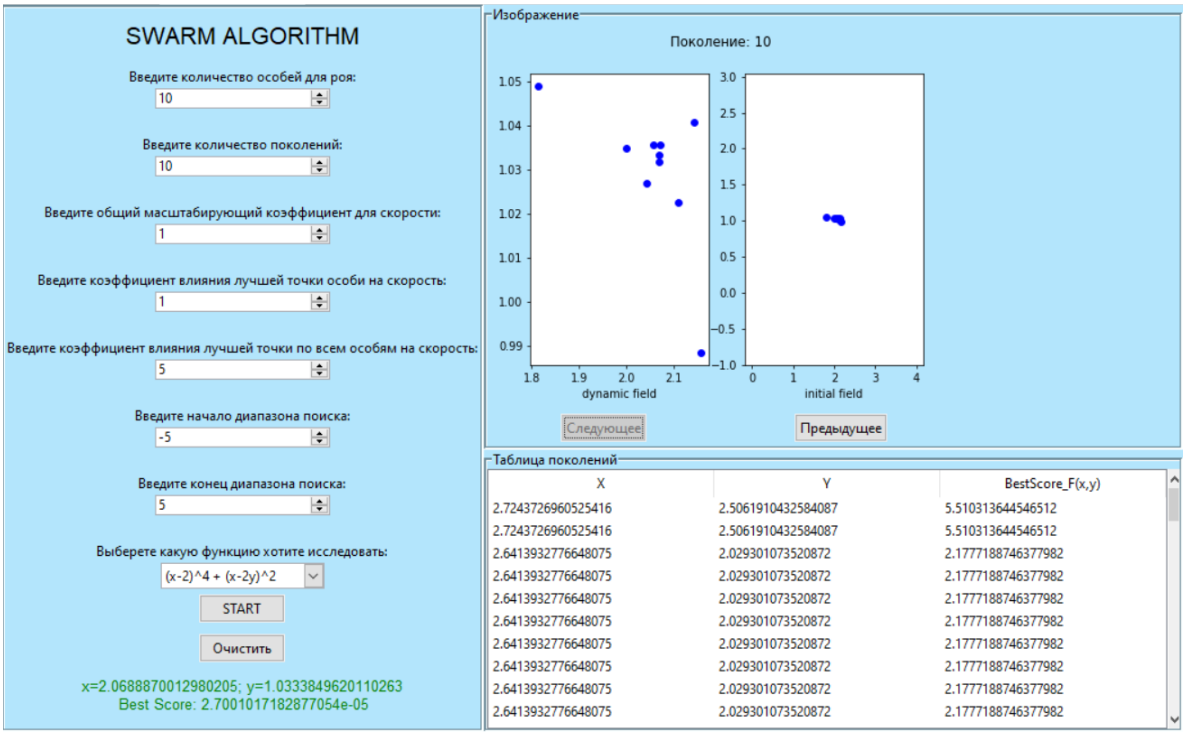

Рис. 2: Окно пользователя.

Устройсво приложения:

В левой части окна можно вводить параметры влияющие на систему, такие как: **размер популяции**, **количество поколений**, **начало и конец деапазона поиска**, **три коэффициента влиящие на скорость изменения координат особи** - 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

Алгоритм роя так же как и генетический аогоритм является эврестическим, из-за чего, мы не можем зарание определить точное число шагов, требуемое для заданой точности экстремума. В целом, роевой алгоритм хорошо проявляет себя на задачах поиска минимума функций, особенно в случаях сложной и многомерной оптимизации. Он способен быстро сходиться к глобальному минимуму при правильном выборе параметров алгоритма. Однако, следует отметить, что производительность роевого алгоритма сильно зависит от выбора начальных параметров и окрестности поиска, а также от специфики самой функции, которую нужно минимизировать. Поэтому важно проводить тщательный анализ перед применением роевого алгоритма к задаче оптимизации.
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.

Таблицы изменения тосности алгоритма от масштабирующих коэффециентов не представленны, так как при их зменении закономерномость повышения/понижения точности полностью отсутсвует, их требуется подобрать самостоятельно, если хотите добиться лучшей точности или оставьте их по умолчанию программы.

6.1 Промежуточный вывод

На [стр.10] представленны Таблица 3 и Таблица 4, которые показывают точность алгоритма при изменении популяции и поколений. Не трудно заметить, что точность, хоть не сильно, но растёт при увеличении одного из параметров, или по крайней мере остаётся не изменной в случае Таблицы 4. В Таблице 5 видно же, что при одновременном изменение двух параметров можно добиться лучшей точности.

Сравнивая результаты работы **генетического алгоритма**, расмотренного в лабалаторной работе №2, и **роевого алгоритма** можно с увереностью сказать, что последний превосходит своего предшесвеника по точности. Более того, так называемые выбросы, когда вычисления при изменении одно из параметров могут сильно скакать, в роевом алгоритме практически отсутсвуют, чего нельзя сказать про генетический.

Зависимость тосности алгоритма от размера популяции					
10	20	50	100	300	500
x=1.97649455 y=0.99057347 Best Score: 2.195001e-05	x=1.99019749 y=0.99499878 Best Score: 4.920362e-08	x=2.00924157 y=1.00450442 Best Score: 6.145689e-08	x=2.00422499 y=1.00202578 Best Score: 3.039830e-08	x=1.99660344 y=0.99829803 Best Score: 1.874195e-10	x=2.00597554 y=1.00297384 Best Score: 2.050993e-09

Таблица 3: Изменение популяции

Зависимость тосности алгоритма от кол-ва поколений					
5	10	20	50	100	500
x=1.79211908 y=0.88683961 Best Score: 0.00220751	x=2.00229725 y=1.00103418 Best Score: 5.241788e-08	x=2.03540366 y=1.01770444 Best Score: 1.571088e-06	x=1.94996673 y=0.97500732 Best Score: 6.268945e-06	x=1.94904121 y=0.97450277 Best Score: 6.744631e-06	x=1.98251142 y=0.99125036 Best Score: 9.365885e-08

Таблица 4: Изменение кол-ва поколений

Зависимость тосности алгоритма от кол-ва поколений и размера популяци					
5	10	20	50	100	500
x=2.15338555 y=0.88019050 Best Score: 0.1550060911	x=2.03002567 y=1.01377983 Best Score: 6.894007e-06	x=1.99856382 y=0.99928027 Best Score: 1.506744e-11	x=1.99590221 y=0.99795121 Best Score: 2.820118e-10	x=2.00293460 y=1.00146713 Best Score: 7.427590e-11	x=2.00077290 y=1.00038644 Best Score: 3.570334e-13

Таблица 5: Изменение кол-ва поколений и размера популяции

Заключение

В данной работе был расмотрен алгоритм роя частиц на нахождение минимума. Были выявленны его недостатки, и сравнение его с генетическим алгоритмом из предыдущей лабараторной работы. Была представленна его прогрманая визуализация и теоретическое растолкование.

Список литературы

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