Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 3 по дисциплине «Основы искусственного интеллекта»

Тема Генетические и биоподобные

Студент Сапожков А.М,

Группа ИУ7-13М

Преподаватель Строганов Ю.В.

Москва, 2024

Содержание

ВВЕДЕНИЕ

Генетические и эволюционные алгоритмы являются разновидностью численных методов решения оптимизационных задач. Основное внимание в них уделяется использованию принципов естественного отбора, мутации и рекомбинации, имитирующих естественный процесс эволюции, для поиска оптимального или близкого к оптимальному решения сложной проблемы.

Целью данной лабораторной работы является составление программы, которая, с использованием алгоритмов оптимизации (генетического и эволюционного на основе муравьиной кучи), аппроксимирует функции $f(x) = sin(x)*(sin(x)+cos(x)), x \in [-2\pi; +2\pi], g(x) = x^3-x+3$.

Задачи данной лабораторной работы:

1) описать общие этапы функционирования системы;
2) описать предлагаемый генетический алгоритм;
3) описать алгоритм муравьиной кучи;
4) реализовать программу для аппроксимации заданных функций;
5) измерить среднеквадратичную ошибку и время выполнения алгоритмов;
6) сравнить реализованные алгоритмы.

1 Аналитическая часть

1.1 Аппроксимация функций

Под **аппроксимацией** понимается некоторое **приближение**, приближенное представление. Соответственно, если речь идёт об аппроксимации функции, то ставится вопрос о её замене другой, например, более простой функцией, которая в каком-то смысле близка к заданной. В результате строится система алгебраических уравнений, аппроксимирующих исходную дифференциальную задачу.

По сути, речь идёт о восстановлении непрерывной функции по заданному набору её значений для непрерывного множества значений аргумента. Так, исходную функциональную зависимость $y(x)$, заменяют приближенной функцией $\phi(x, \overline{a})$, которую и используют в дальнейших вычислениях. При этом близость $y(x)$ и $\phi(x, \overline{a})$ обеспечивается подбором свободных параметров $\overline{a} = a_0, a_1, \ldots, a_m$ в соответствии с некоторым критерием, определяющим степень совпадения аппроксимируемой и аппроксимирующей функций. В данной работе в качестве такого критерия предлагается использовать среднеквадратичное отклонение

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(y(x_i) - \phi(x_i, \overline{a}))^2. \tag{1.1}$$

При этом функция $\phi(x_i, \overline{a})$ определяется следующим образом.

$$\phi(x_i, \overline{a}) = \sum_{k=0}^{m} a_k x_i^k. \tag{1.2}$$

Для аппроксимации функции $f(x) = sin(x) * (sin(x) + cos(x))$ предлагается взять полином 5 степени ($m = 5$), а для функции $g(x) = x^3 - x + 3$ — полином 3 степени ($m = 3$).

1.2 Общие этапы функционирования системы

Генетические и эволюционные алгоритмы будут использоваться для подбора и оптимизации коэффициентов $\overline{a}$ полинома $\phi(x_i, \overline{a})$.

На рисунке 1.1 представлены основные этапы работы генетического алгоритма. [1]

Вывод

В данном разделе были описаны общие этапы работы генетических алгоритмов.

Рисунок 1.1 — Основные этапы работы генетического алгоритма

2 Конструкторская часть

2.1 Генетический алгоритм

Генетический алгоритм начинается с популяции случайно выбранных потенциальных решений (индивидуумов), для которых вычисляется функция приспособленности. Алгоритм выполняет цикл, в котором последовательно применяются операторы отбора, скрещивания и мутации, после чего приспособленность индивидуумов пересчитывается. Цикл продолжается, пока не выполнено условие остановки, после чего лучший индивидуум в текущей популяции считается решением. [1]

Начальная популяция состоит из случайным образом выбранных потенциальных решений (индивидуумов). Поскольку в генетических алгоритмах индивидуумы представлены хромосомами, начальная популяция – это, по сути дела, набор хромосом. Формат хромосом должен соответствовать принятым для решаемой задачи правилам, например это могут быть двоичные строки определённой длины. [1] В случае аппроксимации функций хромосомой будет являться значение схожести аппроксимированного и истинного значения функции в точке. Данное значение предлагается рассчитывать по формуле

$$fitness = \frac{1}{\sigma^2}, \tag{2.1}$$

где σ - среднеквадратичное отклонение между аппроксимируемой и аппроксимирующей функциями.

Для каждого индивидуума вычисляется функция приспособленности. Это делается один раз для начальной популяции, а затем для каждого нового поколения после применения операторов отбора, скрещивания и мутации. Поскольку приспособленность любого индивидуума не зависит от всех остальных, эти вычисления можно производить параллельно. [1]

Так как на этапе отбора, следующем за вычислением приспособленности, более приспособленные индивидуумы обычно считаются лучшими решениями, генетические алгоритмы естественно «заточены» под нахождение максимумов функции приспособленности. Если в какой-то задаче нужен минимум, то при вычислении приспособленности следует инвертировать найденное значение, например умножив его на $^{\smile}1$. [1]

Применение генетических операторов к популяции приводит к созданию новой популяции, основанной на лучших индивидуумах из текущей. [1]

Оператор **отбора** отвечает за отбор индивидуумов из текущей популяции таким образом, что предпочтение отдаётся лучшим. [1]

Оператор **скрещивания** (или рекомбинации) создаёт потомка выбранных индивидуумов. Обычно для этого берутся два индивидуума, и части их хромосом меняются местами, в результате чего создаются две новые хромосомы, представляющие двух потомков. [1]

Оператор **мутации** вносит случайные изменения в один или несколько генов хромосомы вновь созданного индивидуума. Обычно вероятность мутации очень мала. [1]

2.2 Эволюционный алгоритм муравьиной кучи

Идея **муравьиных алгоритмов оптимизации** (Ant Colony Optimization – ACO) подсказана способом поиска пищи некоторыми видами насекомых. Муравьи начинают рыскать случайным образом, и когда один находит пищу, он возвращается в муравейник, помечая свой путь феромонами. Другие муравьи, нашедшие пищу в том же месте, усиливают след, оставляя собственные феромоны. Со временем феромоновые метки выветриваются, поэтому более короткие пути и пути, по которым прошло больше муравьёв, имеют преимущество. [1]

В муравьиных алгоритмах используются искусственные муравьи, которые обследуют пространство поиска, стремясь найти лучшие решения. Критерием отбора решений в случае аппроксимации функций является среднеквадратичное отклонение между аппроксимируемой и аппроксимирующей функциями. Муравьи запоминают, где были и какие решения нашли по пути. Эта информация используется муравьями на следующих итерациях для поиска ещё лучших решений. Часто такие алгоритмы сочетаются с методами локального поиска, которые подключаются, когда найдена перспективная область. [1]

Алгоритм оптимизации на основе муравьиной кучи (Dorigo, 1992) изначально был предложены для решения задач комбинаторной оптимизации (COP). Примеры COP включают планирование, маршрутизацию транспортных средств, составление расписания и т. д. С момента появления ACO как инструмента комбинаторной оптимизации предпринимались попытки использовать его для решения задач в непрерывных областях (continous domains). Однако применить метаэвристику ACO к непрерывным областям было непросто, и предлагаемые методы часто черпали вдохновение из ACO, но не следовали ему в точности. [2] [3]

Центральным компонентом алгоритмов ACO является модель феромонов, которая используется для вероятностной выборки пространства поиска. Его можно вывести из рассматриваемой модели COP. Модель COP можно определить следующим образом: [3]

— пространство поиска S, определённое над конечным набором дискретных переменных и набором Ω ограничений среди переменных;

— целевая функция $f : S \longrightarrow \mathbb{R}_0^+$, значение которой должно быть минимизировано.

Модель COP используется для получения модели феромонов, используемой ACO. Подобно задаче комбинаторной оптимизации (COP), также может быть формально определена модель задачи непрерывной оптимизации (CnOP): [3]

— пространство поиска S, определённое над конечным набором непрерывных переменных и набором Ω ограничений среди переменных;

— целевая функция $f : S \longrightarrow \mathbb{R}_0^+$, значение которой должно быть минимизировано.

В ACO для комбинаторной оптимизации информация о феромонах хранится в виде таблицы. На каждой итерации при выборе компонента, добавляемого к текущему частному решению, муравей использует некоторые значения из этой таблицы в качестве дискретного распределения вероятностей. В случае непрерывной оптимизации выбор, который делает муравей, не ограничивается конечным множеством. Следовательно, невозможно представить феромон в

виде таблицы. Поэтому для непрерывной оптимизации используется следующая идея. Таблица феромонов обновляется на основе найденных компонентов хороших решений — так же, как и в обычном ACO. Однако в обычном ACO фактические решения, найденные муравьями, отбрасываются после обновления таблицы феромонов. Напротив, в данном случае необходимо отслеживать определённое количество решений, используемых для обновления таблицы феромонов. Вместо использования испарения феромонов феромон, связанный с самыми старыми решениями, в конечном итоге удаляется путём обновления таблицы феромонов, тем самым сводя на нет своё влияние.

Количество решений, сохранённых в архиве, установлено равным k, и поэтому этот параметр определяет сложность функции PDF (Probability Density Function — функция плотности вероятности, чаще всего для распределения Гаусса). Далее на основе PDF формируется вектор весов решений, используемых для их отбора.

Вывод

В данном разделе были описаны алгоритмы генетической и эволюционной оптимизации в контексте задачи аппроксимации функций.

3 Технологическая часть

3.1 Средства реализации

В качестве языка программирования для реализации выбранных алгоритмов был выбран язык программирования Python [4] ввиду наличия библиотек, реализующих генетические и эволюционные алгоритмы. Для реализации генетического алгоритма использовалась библиотека PyGAD [5], для эволюционного — ACOR [6].

3.2 Реализация алгоритмов

На листинге 3.1 представлена реализация генетического алгоритма аппроксимации функций.

Листинг 3.1 — Генетический алгоритм аппроксимации функций

```
from math import sin, cos, pow, pi
import numpy as np
import pygad
from time import process_time

def f(x: float) -> float:
  return sin(x)*(sin(x)+cos(x))

def g(x: float) -> float:
  return x**3 - x + 3

polynomial_power = 3
func = g
x_start, x_end = -5, 5
x_step = 0.001

x = [i for i in np.arange(x_start, x_end + x_step/2, x_step)]
function_inputs = [[pow(arg, polynomial_power - i) for i in range(
   polynomial_power + 1)] for arg in x]

def fitness(_, solution, solution_idx):
  deviation = 0
  for i in range(len(x)):
    desired_output = func(x[i])
    actual_output = np.sum(solution*function_inputs[i])
    deviation += (desired_output - actual_output)**2
  return 1.0 / ((deviation/len(x))**0.5)
```

```
  start_time = process_time()
  ga_instance = pygad.GA(num_generations=1000,
    num_parents_mating=4,
    fitness_func=fitness,
    sol_per_pop=8,
    num_genes=polynomial_power+1,
    init_range_low=-2,
    init_range_high=5,
    parent_selection_type="sss",
    keep_parents=1,
    crossover_type="single_point",
    mutation_type="random",
    mutation_percent_genes=10)
  ga_instance.run()
  solution, solution_fitness, _ = ga_instance.best_solution()
  print("Time: {diff} seconds".format(diff = process_time() -
     start_time))
  print("Parameters of the best solution : {solution}".format(solution=
     solution))
  print("Fitness value of the best solution = {solution_fitness}".
     format(solution_fitness=solution_fitness))

  ga_instance.plot_fitness()
```

На листинге 3.2 представлена реализация эволюционного алгоритма аппроксимации функций на основе муравьиной кучи.

Листинг 3.2 — Эволюционный алгоритм аппроксимации функций

```
from math import sin, cos, pow, pi
import numpy as np

import Population, Acor, Constants
from time import process_time

def f(x: float) -> float:
  return sin(x)*(sin(x)+cos(x))

def g(x: float) -> float:
  return x**3 - x + 3

polynomial_power = 5
```

```
func = f
x_start, x_end = -2*pi, 2*pi
x_step = 0.001

x = [i for i in np.arange(x_start, x_end + x_step/2, x_step)]
function_inputs = [[pow(arg, polynomial_power - i) for i in range(
   polynomial_power + 1)] for arg in x]

def error(solution):
  deviation = 0
  for i in range(len(x)):
    desired_output = func(x[i])
    actual_output = np.sum(solution*function_inputs[i])
    deviation += (desired_output - actual_output)**2
  return (deviation/len(x))**0.5

def cost(function_input):
  return -func(function_input[0])

start_time = process_time()
acor = Acor.AcorContinuousDomain(n_pop=Constants.AcoConstants.N_POP,
  n_vars=polynomial_power+1,
  cost_func=error,
  domain_bounds=[-20, 20])
acor.runMainLoop()
print("Time: {diff} seconds".format(diff = process_time() - start_time)
   )
print("Parameters of the best solution : {solution}".format(solution=
   acor.final_best_solution.position))
print("Cost value of the best solution = {solution_cost}".format(
   solution_cost=acor.final_best_solution.cost_function))
```

Вывод

В данном разделе были приведены детали реализации алгоритмов аппроксимации функций.

4 Исследовательская часть

4.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором выполнялось тестирование.

— Операционная система: macOS 14.6.1.

— Объём оперативной памяти: 18 Гб.

— Процессор: Apple M3 Pro.

Тестирование проводилось на ноутбуке, включённом в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.2 Время выполнения реализаций алгоритмов

Алгоритмы тестировались при помощи функции process_time() из библиотеки time языка Python. Данная функция всегда возвращает значения времени в секундах типа float, которые являются суммой системного и пользовательского процессорного времени процессора, на котором выполнялась программа. [7]

Среднее время выполнения генетического алгоритма составило 149.64 с. для аппроксимации функции $f(x)$ и 117.12 с. для $g(x)$. Результат аппроксимации функции $f(x)$:

$$\phi_f(x, \overline{a}) = -0.045774x^5 - 0.032708x^4 + 1.996497x^3 + 1.078365x^2 - 16.730258x - 3.44288, \quad (4.1)$$

$$\sigma_f = 17.544. \quad (4.2)$$

Результат аппроксимации функции $g(x)$:

$$\phi_g(x, \overline{a}) = 0.99128784x^3 + 0.00712201x^2 - 0.88137867x + 2.93948756, \quad (4.3)$$

$$\sigma_g = 0.177. \quad (4.4)$$

На рисунках 4.1 и 4.2 приведены графики зависимостей между числом пройденных поколений оптимизации и похожестью между аппроксимирующей и аппроксимируемой функциями.

Среднее время выполнения эволюционного алгоритма составило 173.17 с. для аппроксимации функции $f(x)$ и 128.098286 с. для $g(x)$. Результат аппроксимации функции $f(x)$:

$$\phi_f(x, \overline{a}) = -17.00417829x^5 - 3.73938996x^4 - 17.54370723x^3 + 20x^2 - 1.720966x + 20, \quad (4.5)$$

$$\sigma_f = 25.447. \quad (4.6)$$

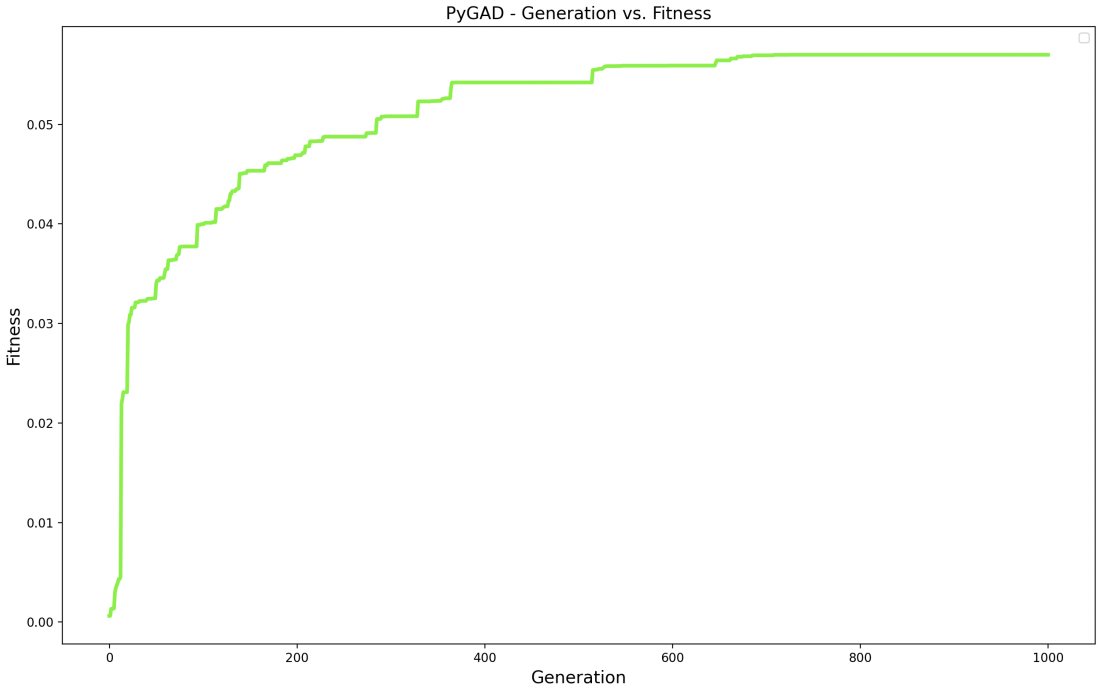

Рисунок 4.1 — Зависимость похожести $f(x)$ и $\phi_f(x, \overline{a})$ от числа пройденных поколений оптимизации

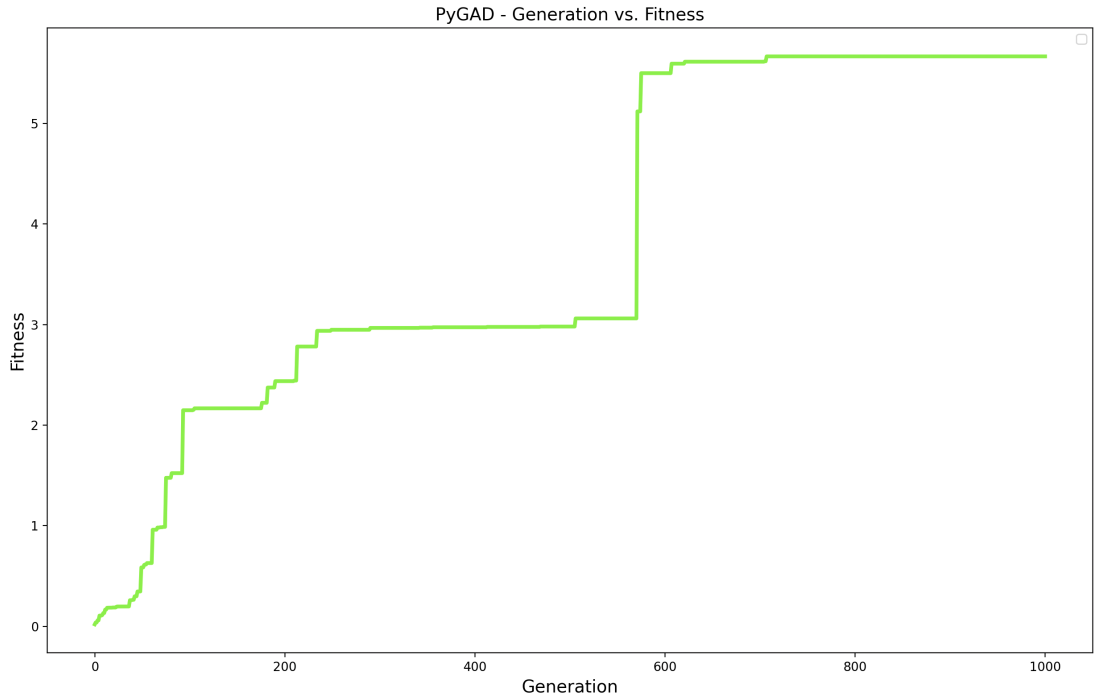

Рисунок 4.2 — Зависимость похожести $g(x)$ и $\phi_g(x, \overline{a})$ от числа пройденных поколений оптимизации

Результат аппроксимации функции $g(x)$:

$$\phi_g(x, \overline{a}) = 0.48981863x^3 + 0.21815401x^2 + 0.14574346x + 0.00134332, \tag{4.7}$$

$$\sigma_g = 8.591. \tag{4.8}$$

Вывод

В данном разделе были приведены результаты замеров времени выполнения алгоритмов аппроксимации функций. При сравнимом времени выполнения эволюционный алгоритм на основе муравьиной кучи показал меньшую точность, чем генетический алгоритм: для $f(x)$ среднеквадратичная ошибка была в 1.5 раз больше, для $g(x)$ — в 48 раз больше.

ЗАКЛЮЧЕНИЕ

В рамках лабораторной работы была составлена программа, которая, с использованием алгоритмов оптимизации (генетического и эволюционного на основе муравьиной кучи), аппроксимирует функции $f(x) = sin(x) * (sin(x) + cos(x)), x \in [-2\pi; +2\pi], g(x) = x^3 - x + 3$.. Все поставленные задачи были выполнены.

1) описаны общие этапы функционирования системы;
2) описан предлагаемый генетический алгоритм;
3) описан алгоритм муравьиной кучи;
4) реализован программу для аппроксимации заданных функций;
5) измерены среднеквадратичную ошибку и время выполнения алгоритмов;
6) проведено сравнение реализованных алгоритмов.

При сравнимом времени выполнения эволюционный алгоритм на основе муравьиной кучи показал меньшую точность, чем генетический алгоритм: для $f(x)$ среднеквадратичная ошибка была в 1.5 раз больше, для $g(x)$ — в 48 раз больше. Для оптимизации качества работы эволюционного алгоритма можно провести исследование с целью подбора параметров алгоритма, а также изменить критерий для отбора решений поставленной задачи.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Вирсански Э. Генетические алгоритмы на Python / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2020. – 286 с.

2. Дэн Саймон Алгоритмы эволюционной оптимизации / пер. с англ. А. В. Логунова. – М.: ДМК Пресс, 2020. – 1002 с.

3. Socha K., Dorigo M., Ant colony optimization for continuous domains, European Journal of Operational Research, Volume 185, Issue 3, 2008, Pages 1155-1173, ISSN 0377-2217.

4. Python [Электронный ресурс]. — Режим доступа, URL: https://www.python.org/ (дата обращения: 19.11.2024).

5. PyGAD: Genetic Algorithm in Python [Электронный ресурс]. — Режим доступа, URL: https://pypi.org/project/pygad/ (дата обращения: 19.11.2024).

6. Ant Colony Optimization [Электронный ресурс]. — Режим доступа, URL: https://github.com/aidowu1/Py-Optimization-Algorithms/tree/master/AntColonyOptimizations (дата обращения: 19.11.2024).

7. time — Time access and conversions [Электронный ресурс]. — Режим доступа, URL: https://docs.python.org/3/library/time.html#time.process_time (дата обращения: 19.11.2024).