МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КУБГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**Отчет**

**по лабораторной работе №3 по курсу**

**«МЕТОДЫ ПОИСКОВОЙ ОПТИМИЗАЦИИ»**

Работу выполнили

Студены 36 группы

Воробьев А.Д.

Андреев А.П.

Преподаватель:

Полупанова Е.Е

Краснодар 2024

**СОДЕРЖАНИЕ**

[Описание задачи 3](file:///C:\Users\miste\Downloads\Eremin_Lab3%20(1).docx#_Toc148042159)

[Алгоритм 5](file:///C:\Users\miste\Downloads\Eremin_Lab3%20(1).docx#_Toc148042160)

[Листинг программы 6](file:///C:\Users\miste\Downloads\Eremin_Lab3%20(1).docx#_Toc148042161)

[Результаты работы программы 8](file:///C:\Users\miste\Downloads\Eremin_Lab3%20(1).docx#_Toc148042162)

[Вывод 9](file:///C:\Users\miste\Downloads\Eremin_Lab3%20(1).docx#_Toc148042163)

**Цель работы:** разработать генетический алгоритм оптимизации функции Розенброкка.

**Ход работы:**

Генетический алгоритм (ГА) — это метаэвристический метод оптимизации, инспирированный процессами естественного отбора и генетики. Этот алгоритм используется для решения задачи оптимизации и поиска, особенно в ситуациях, где простое и полный переборное решение невозможно или неэффективно. Генетические алгоритмы часто применяются для нахождения приближенных решений в множестве возможных вариантов.

Рассмотрим ключевые элементы генетического алгоритма и его работа:

1. **Популяция**: Генетический алгоритм начинается с создания начальной популяции. Каждый элемент популяции представляет собой потенциальное решение задачи и называется "особью". Эти особи могут быть представлены в виде строк, векторов или других структур данных, в зависимости от конкретной задачи.
2. **Фитнес-функция**: Для оценки качества каждой особи в популяции определяется фитнес-функция. Фитнес-функция принимает особь в качестве входных данных и возвращает численное значение, которое характеризует "подходит ли" эта особь для решения задачи. Цель состоит в том, чтобы максимизировать или минимизировать значение этой функции, в зависимости от задачи оптимизации.
3. **Селекция**: Особи с более высокими значениями фитнес-функции имеют больший шанс выжить и передать свои гены следующему поколению. Это имитирует естественный отбор, где успешные особи имеют больше потомков. Селекция может быть случайной, но вероятность выбора особей зависит от их фитнес-значения.
4. **Скрещивание (кроссовер)**:Выбранные особи скрещиваются, и их гены комбинируются, чтобы создать потомство. Как именно это происходит, зависит от метода скрещивания, который может варьироваться от простой одноточечной кроссовер до более сложных методов. Цель скрещивания - внести разнообразие в популяцию и, возможно, сочетать положительные черты разных особей.
5. **Мутация**: Некоторые гены в потомстве могут случайным образом мутировать. Мутация вносит случайные изменения в гены особей и помогает сохранить разнообразие в популяции.
6. **Замена поколения**: Новое поколение особей формируется путем комбинирования селекции, скрещивания и мутации. Часть старой популяции заменяется новой на основе их фитнес-значений.
7. **Критерий остановки**: Генетический алгоритм продолжает создавать новые поколения и улучшать популяцию до тех пор, пока не выполнится определенный критерий остановки. Критерии могут включать ограничение по количеству поколений, достижение желаемого значения фитнес-функции или истечение времени.

Теперь рассмотрим подробнее реализацию генетического алгоритма на функции Розенброка:

1. **Инициализация популяции**: В начале работы алгоритма создается начальная популяция из случайных особей. Каждая особь представляется двумя генами (x, y), и ей назначается значение фитнес-функции (z), которое рассчитывается на основе значений x и y, используя вашу оптимизируемую функцию. Наша цель - найти особь (x, y), для которой значение z минимальное.
2. **Эволюция популяции:**

* *Селекция*: Особи в текущей популяции ранжируются в порядке их значений фитнес-функции. Лучшие особи (те, у которых z ближе к оптимуму) имеют большие шансы быть выбранными как родители. Количество особей, которые будут выбраны в качестве родителей, определяется коэффициентом выживаемости (survive\_cof).
* *Репродукция* (скрещивание): Выбранные родители принимают участие в создании потомства. Для каждой новой особи, x и y наследуются от родителей, и значение z пересчитывается на основе новых x и y. Вероятность того, что x или y будут заменены значениями другого родителя, зависит от случайного числа. Это процесс повторяется для нескольких новых особей.
* *Мутация*: Каждая особь в популяции имеет шанс быть подверженной мутации. Это означает, что x и/или y могут быть незначительно изменены случайным образом. Вероятность мутации определяется параметром mut\_chance.

1. **Итерации**: Эти шаги (селекция, скрещивание и мутация) повторяются в течение заданного количества поколений (generations). В каждом поколении лучшие особи сохраняются, а менее успешные могут быть заменены новыми особями.
2. **Завершение и выбор лучшей особи**: После завершения всех итераций, алгоритм возвращает особь с наилучшим значением z, то есть особь с минимальным значением z.

**Результат работы программы**

Процесс эволюции для функции Розенброка с параметрами:

* Размер популяции - 5000
* Количество поколений - 20
* Вероятность мутации - 20%

Изображение выглядит как текст, снимок экрана, диаграмма, дизайн

Автоматически созданное описание

Рисунок 1 – Результат выполнения алгоритма

# 

# Вывод

В ходе работы был изучен и реализован генетический алгоритм на определенной функции.

# Листинг программы

import random

def getFunc1():

def mathFunc1(x, y):

return (1 - x) \*\* 2 + 100 \* (y - x \*\* 2) \*\* 2

return [mathFunc1, (-5,5 ), (-5, 5)]

def generateFirstPopulation(targetFunc, populationSize):

limitsX = targetFunc[1]

limitsY = targetFunc[2]

# шанс мутации

result = Population(0.02)

randXs = [random.randint(\*limitsX) for \_ in range(populationSize)]

randYs = [random.randint(\*limitsY) for \_ in range(populationSize)]

for randX, randY in zip(randXs, randYs):

result.add\_species(

Species.from\_floats((randX, randY), min(limitsX[0], limitsY[0]), max(limitsX[1], limitsY[1])))

print(f'[DEBUG] First population:')

for species in result.species:

print(f'[DEBUG] \* {species.to\_floats()}')

return result

def trigger():

scoringFunction = getFunc1()[0]

# Размер популяции

population = generateFirstPopulation(getFunc1(), 51)

# Количество поколений

generationsCount = 150

result = []

for i in range(generationsCount - 1):

bestSpecies = population.bestSpecies(scoringFunction).to\_floats()

result.append({

"generation": i,

"data": bestSpecies,

"fitness": getFunc1()[0](\*bestSpecies)

})

print(f'Best in generation {i + 1}: {bestSpecies}; fitness: {getFunc1()[0](\*bestSpecies)}')

population.reproduce(scoringFunction)

bestSpecies = population.bestSpecies(scoringFunction).to\_floats()

print(f'Last generation best: {bestSpecies}; fitness: {getFunc1()[0](\*bestSpecies)}')

result.append({

"generation": generationsCount,

"data": bestSpecies,

"fitness": getFunc1()[0](\*bestSpecies)

})

return result

class Population:

def \_\_init\_\_(self, mutationChance):

self.species = []

self.mutationChance = mutationChance

def add\_species(self, species):

self.species.append(species)

# Произвести размножение. В результате - замена особей этой популяции на новые

def reproduce(self, scoringFunction, resultSpeciesCount=None):

if (resultSpeciesCount is None):

resultSpeciesCount = len(self.species)

children = self.crossingover()

self.mutate(children)

# for child in children:

# print(f'[DEBUG] {child.bits}')

contestants = self.species + children

scores = {}

for contestant in contestants:

scores[contestant] = scoringFunction(\*contestant.to\_floats())

self.species = sorted(scores, key=scores.get)[:resultSpeciesCount]

def bestSpecies(self, scoringFunction):

scores = {}

for speciesSubject in self.species:

scores[speciesSubject] = scoringFunction(\*speciesSubject.to\_floats())

return min(scores, key=scores.get)

# Произвести скрещивание. Формируем брачные пары и обменяемся участками особей.

# Возвращает список получившихся детей

def crossingover(self):

children = []

pairs = {}

for speciesSubject in self.species:

pairs[speciesSubject] = random.choice(self.species)

for parent1, parent2 in pairs.items():

chromos1 = parent1.bits

chromos2 = parent2.bits

breakPoint = random.randrange(1, len(chromos1[0]))

resultChromos = []

for coordIdx in range(len(chromos1)):

bitsList1 = chromos1[coordIdx]

bitsList2 = chromos2[coordIdx]

resultBitsList1 = bitsList1[0:breakPoint] + bitsList2[breakPoint:]

resultBitsList2 = bitsList2[0:breakPoint] + bitsList1[breakPoint:]

resultChromos.append((resultBitsList1, resultBitsList2))

for coordIdx in range(len(chromos1)):

speciesCromos = []

for resultChromo in resultChromos:

speciesCromos.append(resultChromo[coordIdx])

children.append(Species(speciesCromos, min(parent1.encodingFrom, parent2.encodingFrom),

max(parent1.encodingTo, parent2.encodingTo)))

return children

def mutate(self, species):

for speciesSubject in species:

if random.uniform(0, 1) < self.mutationChance:

changingBitNumber = random.randrange(0, len(speciesSubject.bits[0]))

# print(f'MuTaTiOn! changing bit: {changingBitNumber}')

# print(f'BEFORE: {speciesSubject.bits}')

coordinateIdx = random.randrange(0, len(speciesSubject.bits))

speciesSubject.bits[coordinateIdx][changingBitNumber] = 1 - speciesSubject.bits[coordinateIdx][

changingBitNumber]

# print(f'AFTER: {speciesSubject.bits}')

class Species:

# Точность знака

ENCODING\_PRECISION = 40

# Конструктор из массива вещественных чисел

def from\_floats(floatsValues, encodingFrom, encodingTo):

# Если передан не список, оборачиваем в список

if (type(floatsValues) is not list) and (type(floatsValues) is not tuple):

floatsValues = [floatsValues]

bitsStrings = []

for floatValue in floatsValues:

encoded\_int = round(

(floatValue - encodingFrom) \* (2 \*\* Species.ENCODING\_PRECISION - 1) / (encodingTo - encodingFrom))

binString = "{0:b}".format(encoded\_int)

bitsStrings.append(('0' \* (Species.ENCODING\_PRECISION - len(binString))) + binString)

return Species(bitsStrings, encodingFrom, encodingTo)

def \_\_init\_\_(self, bitsStrings, encodingFrom, encodingTo):

self.encodingFrom = encodingFrom

self.encodingTo = encodingTo

# Если передан не список, оборачиваем в список

if (type(bitsStrings) is not list) and (type(bitsStrings) is not tuple):

bitsStrings = [bitsStrings]

self.bits = [[int(x) for x in bitsString] for bitsString in bitsStrings]

def to\_floats(self):

result = []

for bitsList in self.bits:

bitsString = ''.join(map(str, bitsList))

encoded\_int = int(bitsString, base=2)

floatValue = (encoded\_int \* (self.encodingTo - self.encodingFrom)) / (

2 \*\* Species.ENCODING\_PRECISION - 1) + self.encodingFrom

result.append(floatValue)

return result

import tkinter

from tkinter import \*

from tkinter import scrolledtext, messagebox

from tkinter.ttk import Combobox, Notebook, Style

import numpy as np

from matplotlib import pyplot as plt

from matplotlib.backends.backend\_tkagg import (FigureCanvasTkAgg, NavigationToolbar2Tk)

from scipy.optimize import differential\_evolution

from lab3.core import trigger, getFunc1

def rosenbrock(x):

return np.sum(100.0 \* (x[1:] - x[:-1] \*\* 2.0) \*\* 2.0 + (1 - x[:-1]) \*\* 2.0, axis=0)

def make\_data\_lab\_3():

x = np.linspace(-5, 5, 100)

y = np.linspace(-5, 5, 100)

x\_grid, y\_grid = np.meshgrid(x, y)

z = rosenbrock(np.array([x\_grid, y\_grid]))

return x\_grid, y\_grid, z

def main():

window = Tk()

width = window.winfo\_screenwidth()

height = window.winfo\_screenheight()

window.geometry("%dx%d" % (width, height))

window.title("SEARCH OPTIMISATION METHODS")

fig = plt.figure(figsize=(11, 11))

fig.add\_subplot(projection='3d')

canvas = FigureCanvasTkAgg(fig, master=window)

canvas.draw()

canvas.get\_tk\_widget().pack(side=tkinter.RIGHT, fill=tkinter.BOTH)

toolbar = NavigationToolbar2Tk(canvas, window)

toolbar.update()

canvas.get\_tk\_widget().pack(side=tkinter.RIGHT, fill=tkinter.BOTH)

style = Style()

style.theme\_create("dummy", parent="alt", settings={

"TNotebook": {"configure": {"tabmargins": [2, 5, 2, 0]}},

"TNotebook.Tab": {

"configure": {"padding": [5, 1], "background": "#808080"},

"map": {"background": [("selected", "FFFF00")],

"expand": [("selected", [1, 1, 1, 0])]}}})

style.theme\_use("dummy")

tab\_control = Notebook(window)

def draw\_lab\_3():

fig.clf()

x, y, z = make\_data\_lab\_3()

if combo\_tab\_3.get() == "Min":

min\_max = True

else:

min\_max = False

ax = fig.add\_subplot(projection='3d')

ax.plot\_surface(x, y, z, rstride=5, cstride=5, alpha=0.3, cmap="inferno")

canvas.draw()

result = trigger()

canvas.draw()

window.update()

# Эти 4 строки ниже это считай удалить точку/точки

fig.clf()

ax = fig.add\_subplot(projection='3d')

ax.plot\_surface(x, y, z, rstride=5, cstride=5, alpha=0.5, cmap="inferno")

canvas.draw()

for item in result:

x = item.get("data")[0]

y = item.get("data")[1]

ax.scatter(x, y, getFunc1()[0](x, y), c="red")

txt\_tab\_1.insert(INSERT,

f"{item.get('generation')}) ({round(x, 3)})({round(y, 3)}) = {getFunc1()[0](x, y)}\n")

resultForRef = differential\_evolution(rosenbrock, [(-5, 5), (-5, 5)])

ax.scatter(resultForRef.x[0], resultForRef.x[1], resultForRef.fun, c="blue")

txt\_tab\_1.insert(INSERT,

f"{1}) ({round(resultForRef.x[0], 2)})({round(resultForRef.x[1], 2)}) = {round(resultForRef.fun, 4)}\n")

canvas.draw()

ax.set\_xlabel('X')

ax.set\_ylabel('Y')

ax.set\_zlabel('Z')

window.update()

messagebox.showinfo('Уведомление', 'Готово')

def delete\_lab\_3():

txt\_tab\_1.delete(1.0, END)

tab\_1 = Frame(tab\_control)

tab\_control.add(tab\_1, text="LAB\_1")

main\_f\_tab\_1 = LabelFrame(tab\_1, text="Parameters")

left\_f\_tab\_1 = Frame(main\_f\_tab\_1)

right\_f\_tab\_1 = Frame(main\_f\_tab\_1)

txt\_f\_tab\_1 = LabelFrame(tab\_1, text="Execution and results")

lbl\_5\_tab\_1 = Label(tab\_1, text="Лаба 3")

txt\_tab\_1 = scrolledtext.ScrolledText(txt\_f\_tab\_1)

btn\_del\_tab\_1 = Button(tab\_1, text="Clear", command=delete\_lab\_3, foreground="black", background="grey")

btn\_tab\_1 = Button(tab\_1, text="Execute", foreground="black", background="yellow", command=draw\_lab\_3)

lbl\_5\_tab\_1.pack(side=TOP, padx=5, pady=5)

main\_f\_tab\_1.pack(side=TOP, padx=5, pady=5, fill=BOTH, expand=True)

left\_f\_tab\_1.pack(side=LEFT, fill=BOTH, expand=True)

right\_f\_tab\_1.pack(side=RIGHT, fill=BOTH, expand=True)

txt\_tab\_1.pack(padx=5, pady=5, fill=BOTH, expand=True)

combo\_tab\_3 = Combobox(right\_f\_tab\_1)

combo\_tab\_3['values'] = ("Min", "Max")

combo\_tab\_3.set("Min")

btn\_tab\_1.pack(side=BOTTOM, padx=5, pady=5, fill=BOTH, expand=True)

btn\_del\_tab\_1.pack(side=BOTTOM, padx=5, pady=5, fill=BOTH, expand=True)

txt\_f\_tab\_1.pack(side=BOTTOM, padx=5, pady=5, fill=BOTH, expand=True)

tab\_control.pack(side=RIGHT, fill=BOTH, expand=True)

window.mainloop()

window.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

main()