# Name: Nirmal Avhad

# Roll no: 71

# Subject: CL3

**Practical No 8**

**Title :-** Implement DEAP (Distributed Evolutionary Algorithms) using Python.

**Problem Statement:-** Develop a distributed evolutionary algorithm using DEAP (Distributed  Evolutionary Algorithms in Python) to optimize a complex problem that requires intensive  computational resources.

**Objective :-**

1. Implement a scalable and efficient distributed evolutionary algorithm using DEAP to solve  the optimization problem.

2. Improve the optimization process by leveraging parallel processing capabilities and  distributing the workload across multiple nodes.

**Outcome :-**

After completion of this assignment students are able to understand how to developed distributed  evolutionary algorithm will significantly reduce the optimization time and enable the solution of  larger and more complex optimization problems.

**Software Requirements:-**

• Python (3.x recommended)

• Jupyter Notebook or any Python IDE

**Hardware Requirement :-**

A machine with sufficient RAM and processing power for model training (8GB RAM  recommended)

**Prerequisities:-**

• Basic understanding of Java programming

**Theory :-**

DEAP (Distributed Evolutionary Algorithms in Python) is a framework for building and analyzing  distributed evolutionary algorithms. It provides tools for implementing genetic algorithms and other  evolutionary computation techniques in a flexible and easy-to-use manner. DEAP allows users to  parallelize their evolutionary algorithms across multiple processors or computers, making it suitable  for tackling complex optimization problems that require significant computational resources.

DEAP provides a wide range of tools and functionalities to facilitate the implementation and  experimentation of evolutionary algorithms. Some key features of DEAP include:

1. **Genetic Operators:** DEAP provides a set of standard genetic operators such as crossover,  mutation, and selection, which can be easily customized and combined to suit the problem  being solved.

2. **Fitness Evaluation:** DEAP allows users to define custom fitness functions to evaluate the  quality of candidate solutions.

3. **Population Management:** DEAP provides tools for managing populations of candidate  solutions, including initialization methods and selection strategies.

4. **Statistics and Logging:** DEAP includes utilities for tracking and logging the evolution of  candidate solutions over time, including statistics such as average fitness and best fitness.

5. **Parallelization:** DEAP supports parallelization of evolutionary algorithms, allowing users to  take advantage of multi-core processors and distributed computing environments to speed up  optimization tasks.

6. **Integration:** DEAP can be easily integrated with other Python libraries and frameworks,  making it a versatile tool for a wide range of optimization problems.

Overall, DEAP provides a convenient and efficient way to implement evolutionary algorithms in  Python, making it a popular choice for researchers and practitioners working in the field of  evolutionary computation.

**Use and Applications:**

• DEAP is used to solve optimization problems where traditional methods may be impractical  or inefficient.

• It is commonly used in various fields such as engineering, biology, finance, and data science  for optimization tasks.

• Applications include parameter optimization, feature selection, function optimization, and  more.

**Conclusion:**- This way DEAP (Distributed Evolutionary Algorithms) using Python is done.

# Code:

import random

from deap import base, creator, tools, algorithms

# Define the evaluation function (minimize a simple mathematical function)

def eval\_func(individual):

    # Example evaluation function (minimize a quadratic function)

    return sum(x \*\* 2 for x in individual),

# DEAP setup

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))

creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

# Define attributes and individuals

toolbox.register("attr\_float", random.uniform, -5.0, 5.0)  # Example: Float values between -5 and 5

toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr\_float, n=3)  # Example: 3-dimensional individual

toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Evaluation function and genetic operators

toolbox.register("evaluate", eval\_func)

toolbox.register("mate", tools.cxBlend, alpha=0.5)

toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb=0.2)

toolbox.register("select", tools.selTournament, tournsize=3)

# Create population

population = toolbox.population(n=50)

# Genetic Algorithm parameters

generations = 20

# Run the algorithm

for gen in range(generations):

    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)

    fits = toolbox.map(toolbox.evaluate, offspring)

    for fit, ind in zip(fits, offspring):

        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))

# Get the best individual after generations

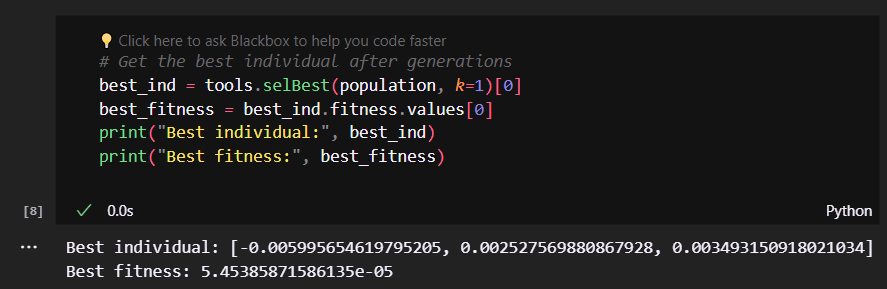
best\_ind = tools.selBest(population, k=1)[0]

best\_fitness = best\_ind.fitness.values[0]

print("Best individual:", best\_ind)

print("Best fitness:", best\_fitness)

**Output:**

****