Experiment No. 5

Problem Statement: Implement DEAP (Distributed Evolutionary Algorithms) using Python

Theory:

Distributed Evolutionary Algorithms (DEAP): Distributed Evolutionary Algorithms are a class of evolutionary algorithms designed to solve complex optimization problems by distributing computation across multiple processors or computational nodes. DEAP offers scalability, efficiency, and parallelism, making it suitable for solving large-scale optimization problems.

DEAP Components:

- Population: A collection of candidate solutions (individuals) representing potential solutions to the optimization problem.
- Selection: The process of selecting individuals from the population for reproduction based on their fitness.
- Crossover: The process of combining genetic material from selected individuals to create new offspring.
- Mutation: The process of introducing random changes to the genetic material of individuals to maintain diversity.
- Evaluation: The process of evaluating the fitness of individuals in the population based on their performance on the optimization problem.
- Termination Criteria: The condition under which the optimization process terminates, such as reaching a maximum number of generations or achieving a desired fitness level.

Features of DEAP:

- Parallelism: DEAP utilizes parallel processing to evaluate fitness functions and explore the search space efficiently.
- Flexibility: DEAP provides a flexible framework for implementing various evolutionary algorithms, including Genetic Algorithms, Evolutionary Strategies, and Genetic Programming.
- Customization: DEAP allows customization of operators, selection methods, and termination criteria to tailor the algorithm to specific problem domains.

Conclusion: Thus, we have successfully implemented a program for DEAP.

Code and Output:

Experiment 5

In [8]:
```
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)
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))
```

In [9]:
```
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)
```

```
Best individual: [-0.006022710726212742, -0.003661986292915549, -0.014722959206355638]
Best fitness: 0.00026644871589315164
```