Practical No: 06

Problem Statement: Implementation of Clonal selection algorithm using Python

Objective:
1. To implement the clonal selection algorithm in Python for solving optimization problems.
2. To apply the algorithm to a specific optimization problem and compare its performance with other optimization algorithms.
3. To analyze the effectiveness and efficiency of the clonal selection algorithm in finding optimal solutions.

Title: The Clonal Selection Algorithm (CSA) is a computational optimization technique inspired by the clonal selection process in the immune system.

Outcomes:
1. Implementation of the clonal selection algorithm in Python, including the main algorithmic steps such as cloning, mutation, and selection.
2. Evaluation of the algorithm's performance in terms of convergence speed and solution quality compared to other optimization algorithms.

Theory

The Clonal Selection Algorithm (CSA) is a bio-inspired optimization algorithm that is based on the clonal selection process of B cells in the immune system. The algorithm was first proposed by de Castro and Timmis in 2002 as a way to solve complex optimization problems. The CSA mimics the behavior of the immune system by using an iterative process of selection, cloning, and mutation to evolve a population of candidate solutions towards an optimal solution.

Key Concepts:
1. **Antibody Representation:** In the CSA, candidate solutions, called antibodies, are represented as vectors in a multidimensional search space. Each dimension of the vector corresponds to a possible solution to the optimization problem.
2. **Affinity Maturation:** Affinity maturation is the process by which antibodies with higher affinities to the target antigen are selected and cloned. In the context of the CSA, affinity represents the fitness of an antibody, i.e., how well it solves the optimization problem.
3. **Cloning:** Cloning is the process of creating multiple copies of high-affinity antibodies. In the CSA, antibodies are cloned based on their affinity, with higher-affinity antibodies being cloned more frequently.
4. **Mutation:** Mutation introduces diversity into the population by randomly perturbing the cloned antibodies. This helps to explore new regions of the search space and avoid local optima.
5. **Selection:** Selection is the process of choosing antibodies to form the next generation based on their affinity. Higher-affinity antibodies are more likely to be selected, while lower-affinity antibodies may be discarded.

Algorithm Steps:
1. **Initialization:** Initialize a population of antibodies randomly or using a heuristic method.
2. **Affinity Calculation:** Calculate the affinity of each antibody in the population based on its fitness in solving the optimization problem.
3. **Clonal Selection:** Select antibodies for cloning based on their affinity, with higher-affinity antibodies being cloned more.
4. **Cloning:** Create multiple copies of selected antibodies based on their affinity.
5. **Mutation:** Perturb the cloned antibodies to introduce diversity into the population.

6. **Selection:** Select antibodies for the next generation based on their affinity.
7. **Repeat:** Repeat steps 3-6 for a specified number of iterations or until a convergence criteria is met.

Termination: The algorithm terminates when a stopping criterion is met, such as a maximum number of iterations or the convergence of the population towards an optimal solution.

Python Implementation:
Below is a basic implementation of the Clonal Selection Algorithm in Python:

```
# Pseudocode for Clonal Selection Algorithm
def clonal_selection_algorithm(population_size, mutation_rate, num_iterations):
  # Initialize population of antibodies
  antibodies = initialize_population(population_size)

  for _ in range(num_iterations):
      # Calculate affinity of antibodies
      calculate_affinity(antibodies)
     # Select antibodies for cloning
    selected_antibodies = select_antibodies_for_cloning(antibodies)
    # Clone selected antibodies
    cloned_antibodies = clone_antibodies(selected_antibodies)
   # Mutate cloned antibodies
   mutated_antibodies = mutate_antibodies(cloned_antibodies, mutation_rate)
  # Select antibodies for next generation
  antibodies = select_next_generation(antibodies, mutated_antibodies)
# Return best antibody
return best_antibody(antibodies)
```

This implementation is a basic framework for the Clonal Selection Algorithm and can be customized for specific optimization problems by defining appropriate functions for initialization, affinity calculation, selection, cloning, mutation, and selection of the next generation.

Conclusion: Thus designed and implemented of Clonal selection algorithm using Python

Implementation of Clonal selection algorithm using Python.

In [5]:
```python
import random
import numpy as np
```

In [6]:
```python
# Define objective function (fitness function)
def objective_function(x):
    # Example: Sphere function
    return sum([i**2 for i in x])
```

In [7]:
```python
# Define Clonal Selection Algorithm
def clonal_selection_algorithm(dimensions, num_candidates, num_clones, mutation_rate, max_iterations):
    # Initialize population with random solutions
    population = [np.random.uniform(-5, 5, dimensions) for _ in range(num_candidates)]

    for iteration in range(max_iterations):
        # Evaluate fitness of each candidate solution
        fitness = [objective_function(candidate) for candidate in population]

        # Sort candidates by fitness
        sorted_indices = np.argsort(fitness)
        population = [population[i] for i in sorted_indices]

        # Select top candidates for cloning
        clones = population[:num_clones]

        # Clone candidates
        clones = [clone for candidate in clones for clone in [candidate] * num_clones]

        # Mutate clones
        for i in range(len(clones)):
            for j in range(dimensions):
                if random.random() < mutation_rate:
                    clones[i][j] += random.uniform(-0.5, 0.5)  # Add small random value

        # Evaluate fitness of clones
        clone_fitness = [objective_function(clone) for clone in clones]

        # Select top clones
        population = [clones[i] for i in np.argsort(clone_fitness)[:num_candidates]]

        # Output best candidate solution in this iteration
        print(f"Iteration {iteration+1}: Best solution - {population[0]}, Fitness - {clone_fitness[0]}

    # Return the best solution found
    best_solution = population[0]
    best_fitness = objective_function(best_solution)
    return best_solution, best_fitness
```

In [8]:
```python
# Example usage
dimensions = 3
num_candidates = 20
num_clones = 10
mutation_rate = 0.1
max_iterations = 50
```

In [8]:
```
best_solution, best_fitness = clonal_selection_algorithm(dimensions, num_candidates, num_clones, mutat
print(f"Best solution found: {best_solution}, Fitness: {best_fitness}")
```

```
Iteration 1: Best solution - [ 0.91494858  1.01400593 -2.39791813], Fitness - 7.615350268320803
Iteration 2: Best solution - [ 1.60209206  2.59021467 -1.19909973], Fitness - 10.713751158144106
Iteration 3: Best solution - [ 1.42312629  2.07922172 -1.69479632], Fitness - 9.220785966408673
Iteration 4: Best solution - [ 0.96573011  2.27420406 -0.86828463], Fitness - 6.858556949621429
Iteration 5: Best solution - [0.683616   2.83709384 0.3032284 ], Fitness - 8.608379741443851
Iteration 6: Best solution - [3.43164546 2.68369373 1.659446  ], Fitness - 21.732163664663943
Iteration 7: Best solution - [2.96446195 3.1890361  1.26491354], Fitness - 20.557992151840644
Iteration 8: Best solution - [3.26072364 4.15791539 0.80838415], Fitness - 28.57406402067401
Iteration 9: Best solution - [ 2.61482958  5.58271401 -1.78015004], Fitness - 41.17296368105042
Iteration 10: Best solution - [ 2.47328579  6.31538949 -2.9963477 ], Fitness - 54.97938661818103
Iteration 11: Best solution - [ 2.05972626  5.37735527 -4.37433476], Fitness - 52.29322650164934
Iteration 12: Best solution - [ 2.12608908  5.63193699 -4.00519776], Fitness - 52.28057810825926
Iteration 13: Best solution - [ 2.66368938  7.11195186 -3.15753106], Fitness - 67.64510270531488
Iteration 14: Best solution - [ 2.2122908   8.36295808 -3.37704285], Fitness - 86.23771680291934
Iteration 15: Best solution - [ 3.03750638  7.90192279 -3.72393677], Fitness - 85.5345338365582
Iteration 16: Best solution - [ 2.21574144  8.60582485 -2.00216338], Fitness - 82.9783897154675
Iteration 17: Best solution - [ 1.76279686  8.4072032  -1.89198129], Fitness - 77.36811170848237
Iteration 18: Best solution - [ 1.27890636  8.73954307 -1.72413996], Fitness - 80.98787313718957
Iteration 19: Best solution - [2.78114442 9.24174347 0.02398885], Fitness - 93.1451620818952
Iteration 20: Best solution - [ 3.57039716  9.72833607 -0.71518503], Fitness - 107.8997481516875
Iteration 21: Best solution - [ 6.83767656  8.68786652 -1.89595739], Fitness - 125.8274998606515
Iteration 22: Best solution - [ 7.89746375  9.30444443 -2.91508646], Fitness - 157.44034889104051
Iteration 23: Best solution - [ 8.93805805  8.93820034 -1.84808755], Fitness - 163.19573463434202
Iteration 24: Best solution - [ 7.36642173  9.39837464 -1.79974764], Fitness - 145.83270660955677
Iteration 25: Best solution - [ 7.02312633 10.16228756 -2.54298353], Fitness - 159.06315721955175
Iteration 26: Best solution - [ 6.11983918 11.60160467 -4.277392  ], Fitness - 190.34574472944317
Iteration 27: Best solution - [ 7.65631496 10.68429968 -5.2990351 ], Fitness - 200.85319152729213
Iteration 28: Best solution - [ 6.7805429   9.39152375 -5.44355512], Fitness - 163.80877278578743
Iteration 29: Best solution - [ 8.5459746   9.03896064 -5.47538902], Fitness - 184.7163761873154
Iteration 30: Best solution - [ 7.95772484  9.87177115 -4.4042275 ], Fitness - 180.17447017028223
Iteration 31: Best solution - [ 7.29444799 10.00182054 -4.45781148], Fitness - 173.1174689054609
Iteration 32: Best solution - [ 8.45743249  9.98213384 -3.27837054], Fitness - 181.91887378249584
Iteration 33: Best solution - [ 7.93419371  9.92799838 -4.24205523], Fitness - 179.51161417201558
Iteration 34: Best solution - [ 9.60413951 10.72541441 -5.48086365], Fitness - 237.3138764141969
Iteration 35: Best solution - [10.72283926 10.08601155 -4.6033547 ], Fitness - 237.8977853479663
Iteration 36: Best solution - [ 9.6469934   7.77701448 -4.71346566], Fitness - 175.76319439629063
Iteration 37: Best solution - [ 7.76267087  7.90223352 -5.98207565], Fitness - 158.48958280697758
Iteration 38: Best solution - [ 7.29767659  6.76555379 -4.17621227], Fitness - 116.469550676761
Iteration 39: Best solution - [ 6.64250521  6.50322157 -4.95803877], Fitness - 110.99691480969192
Iteration 40: Best solution - [ 5.71004245  6.10204292 -5.68109324], Fitness - 102.11433303868967
Iteration 41: Best solution - [ 5.11179322  6.58309134 -5.0636237 ], Fitness - 95.1078065280305
Iteration 42: Best solution - [ 3.56350787  7.58868184 -5.22866295], Fitness - 97.62559662407713
Iteration 43: Best solution - [ 3.23675993  5.62380744 -5.62390721], Fitness - 73.73215733614
Iteration 44: Best solution - [ 3.72155283  5.73205796 -5.66965201], Fitness - 78.85139783845587
Iteration 45: Best solution - [ 4.47868412  5.55526512 -4.81253004], Fitness - 74.08002741036972
Iteration 46: Best solution - [ 5.36072266  4.88191934 -5.32270129], Fitness - 80.9016329090797
Iteration 47: Best solution - [ 5.98664831  3.73983302 -4.53821888], Fitness - 70.42173964928165
Iteration 48: Best solution - [ 6.72388055  3.44943585 -4.43404505], Fitness - 76.76993288291636
Iteration 49: Best solution - [ 7.2419514   3.98077697 -4.78012762], Fitness - 91.14206540514779
Iteration 50: Best solution - [ 7.42590928  4.35072837 -4.50884633], Fitness - 94.40266133134968
Best solution found: [ 7.42590928  4.35072837 -4.50884633], Fitness: 94.40266133134968
```