ASSIGNMENT 6

PROBLEM STATEMENT:

Implementation of Clonal selection algorithm using Python.

OBJECTIVE:

To efficiently solve optimization problems by emulating the adaptive immune system's principles.

THEORY:

The Clonal Selection Algorithm (CSA) is a bio-inspired optimization algorithm based on the principles of clonal selection and affinity maturation observed in the immune system. The theory behind implementing CSA using Python involves understanding its key components and mechanisms:

- **Clonal Selection:**

  In the immune system, B cells undergo clonal selection, where those with receptors (antibodies) that bind strongly to antigens are replicated (cloned) in response to an antigenic stimulus. This principle is mimicked in CSA, where candidate solutions (antibodies) with higher affinity to the problem's objective function (antigens) are replicated.

- **Affinity Maturation:**

  After clonal selection, B cells undergo affinity maturation, a process where their receptors undergo mutations to improve binding affinity to antigens. Similarly, in CSA, cloned candidate solutions undergo mutation or hypermutation to explore the search space and potentially improve their fitness.

- **Population Initialization:**

  The CSA starts with an initial population of candidate solutions (antibodies) representing potential solutions to the optimization problem.

- **Cloning Operation:**

  Candidate solutions are selected based on their fitness (affinity) to the problem's objective function, and a proportion of the fittest solutions are replicated or cloned to form an expanded population.

- **Mutation Operation:**

  Cloned candidate solutions undergo mutation or hyper mutation to introduce diversity into the population and explore new regions of the search space.

- **Selection Operation:**

  The mutated candidate solutions are evaluated, and the fittest individuals are selected to form the next generation population.

- **Termination Criteria:**

  The algorithm iterates through these operations until a termination criterion is met, such as reaching a maximum number of iterations or achieving a satisfactory solution quality.

Clonal Selection Algorithm (CSA):

1. Start by generating an initial population of candidate solutions (antibodies) randomly or using a heuristic method.
2. Evaluate the fitness of each candidate solution using the objective function, determining how well it performs in solving the optimization problem.
3. Select a proportion of the fittest candidate solutions based on their fitness scores to undergo cloning. The number of clones generated for each selected solution is proportional to its fitness.
4. Introduce diversity into the population by applying mutation or hypermutation to the cloned candidate solutions. This involves randomly modifying the cloned solutions to explore new regions of the search space.
5. Evaluate the fitness of the mutated candidate solutions.
6. Select candidate solutions for the next generation population based on their fitness, considering both the original candidate solutions and their mutated clones.
7. Repeat steps 3-6 for a predefined number of generations or until a termination criterion is met (e.g., a satisfactory solution is found, or a maximum number of iterations is reached).
8. Return the best candidate solution found during the optimization process as the solution to the optimization problem.

By following these steps, the Clonal Selection Algorithm iteratively evolves a population of candidate solutions, favoring those with higher fitness, to efficiently search for an optimal solution to the given optimization problem.

CONCLUSION

We've implemented a Clonal Selection Algorithm using Python. This implementation offers a practical and versatile solution for solving optimization problems.

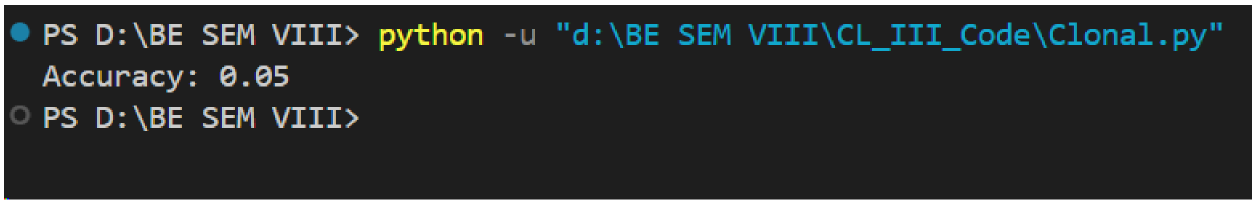
ORAL QUESTION

1. How does the Clonal Selection Algorithm differ from traditional optimization algorithms?
2. What are the key components of your implementation of the Clonal Selection Algorithm?

Code:
```
import numpy as np

# Generate dummy data for demonstration purposes (replace this with
your actual data)
def generate_dummy_data(samples=100, features=10):
    data = np.random.rand(samples, features)
    labels = np.random.randint(0, 2, size=samples)
    return data, labels

# Define the AIRS algorithm
class AIRS:
    def __init__(self, num_detectors=10, hypermutation_rate=0.1):
        self.num_detectors = num_detectors
        self.hypermutation_rate = hypermutation_rate

    def train(self, X, y):
        self.detectors = X[np.random.choice(len(X),
self.num_detectors, replace=False)]

    def predict(self, X):
        predictions = []
        for sample in X:
            distances = np.linalg.norm(self.detectors - sample,
axis=1)
            prediction = int(np.argmin(distances))
            predictions.append(prediction)
        return predictions

# Generate dummy data
data, labels = generate_dummy_data()

# Split data into training and testing sets
split_ratio = 0.8
split_index = int(split_ratio * len(data))
train_data, test_data = data[:split_index], data[split_index:]
train_labels, test_labels = labels[:split_index],
labels[split_index:]

# Initialize and train AIRS
airs = AIRS(num_detectors=10, hypermutation_rate=0.1)
airs.train(train_data, train_labels)

# Test AIRS on the test set
predictions = airs.predict(test_data)
```

```
# Evaluate accuracy
accuracy = np.mean(predictions == test_labels)
print(f"Accuracy: {accuracy}")
```

Output:



```
PS D:\BE SEM VIII> python -u "d:\BE SEM VIII\CL_III_Code\Clonal.py"
Accuracy: 0.05
PS D:\BE SEM VIII>
```