```
import numpy as np

# Define the objective function
def objective_function(x):
    # Example: A simple quadratic function to minimize
    return sum([xi**2 for xi in x])

# Gene Expression Algorithms (GEA)
def gene_expression_algorithm(
    objective_function,
    num_genes,
    population_size,
    mutation_rate,
    crossover_rate,
    num_generations,
    bounds
):
    # Initialize the population with random genetic sequences
    population = np.random.uniform(bounds[0], bounds[1], size=(population_size,

    def evaluate_fitness(pop):
        # Evaluate the fitness of each genetic sequence
        return np.array([objective_function(individual) for individual in pop])

    def select_parents(fitness):
        # Selection based on fitness (e.g., roulette wheel selection)
        total_fitness = np.sum(1 / (fitness + 1e-6))  # Invert fitness to favor
        probabilities = (1 / (fitness + 1e-6)) / total_fitness
        indices = np.random.choice(np.arange(population_size), size=population_s
        return population[indices]

    def crossover(parent1, parent2):
        # Perform crossover between two parents
        if np.random.random() < crossover_rate:
            point = np.random.randint(1, num_genes - 1)
            child1 = np.concatenate((parent1[:point], parent2[point:]))
            child2 = np.concatenate((parent2[:point], parent1[point:]))
            return child1, child2
        return parent1, parent2

    def mutate(individual):
        # Apply mutation to an individual
        for gene in range(num_genes):
            if np.random.random() < mutation_rate:
                individual[gene] += np.random.uniform(bounds[0], bounds[1]) * 0
        return np.clip(individual, bounds[0], bounds[1])
```

```
    # Main optimization loop
    best_solution = None
    best_fitness = float("inf")

    for generation in range(num_generations):
        fitness = evaluate_fitness(population)

        # Track the best solution
        current_best_idx = np.argmin(fitness)
        if fitness[current_best_idx] < best_fitness:
            best_fitness = fitness[current_best_idx]
            best_solution = population[current_best_idx]

        # Selection
        selected_population = select_parents(fitness)

        # Crossover
        next_population = []
        for i in range(0, population_size, 2):
            parent1 = selected_population[i]
            parent2 = selected_population[(i + 1) % population_size]
            child1, child2 = crossover(parent1, parent2)
            next_population.append(child1)
            next_population.append(child2)

        # Mutation
        next_population = np.array([mutate(individual) for individual in next_p

        # Update population
        population = next_population

    return best_solution, best_fitness

if __name__ == "__main__":
    # Define problem parameters
    num_genes = 5  # Number of genes in each genetic sequence
    population_size = 20
    mutation_rate = 0.1
    crossover_rate = 0.8
    num_generations = 100
    bounds = (-10, 10)  # Bounds for the search space

    # Run Gene Expression Algorithm
    best_solution, best_value = gene_expression_algorithm(
        objective_function,
        num_genes,
        population_size,
        mutation_rate,
        crossover_rate,
```

```
        num_generations,
        bounds
    )

    print("Best Solution:", best_solution)
    print("Best Value:", best_value)
```

```
Best Solution: [ 0.00928274 -0.02998551  0.01939463 -0.03299994  0.0088211
Best Value: 0.0025282593311297752
```