In [5]:

```python
# Simple genetic Algorithm
import random
import matplotlib.pyplot as plt
import numpy as np

# DeJong Evaluation Functions
def sphere(chromosome, num_bits,min_value, max_value):
    value = int(''.join(map(str, chromosome)), 2) / (2**num_bits - 1) * (max_value - min
    return value**2

def step(chromosome, num_bits, min_value, max_value):
    value = int(''.join(map(str, chromosome)), 2) / (2**num_bits - 1) * (max_value - min
    return abs(value)

def rosenbrock(chromosome, num_bits, min_value, max_value):
    value = int(''.join(map(str, chromosome)), 2) / (2**num_bits - 1) * (max_value - min
    return (1 - value)**2

def quartic(chromosome, num_bits, min_value, max_value):
    max_value = 1.28
    min_value = -1.28
    value = int(''.join(map(str, chromosome)), 2) / (2**num_bits - 1) * (max_value - min
    return value**4

def foxholes(chromosome, num_bits, min_value, max_value):
    max_value = 65.536
    min_value = -65.536
    value = int(''.join(map(str, chromosome)), 2) / (2**num_bits - 1) * (max_value - min
    a = [-32, -16, 0, 16, 32]
    return sum(1 / ((value - ai)**6 + (value - bi)**6 + 1) for ai in a for bi in a)

# Step 3: Crossover and Mutation
def crossover(parent1, parent2, crossover_prob):
    if random.random() < crossover_prob:
        crossover_point = random.randint(1, len(parent1) - 1)
        child1 = parent1[:crossover_point] + parent2[crossover_point:]
        child2 = parent2[:crossover_point] + parent1[crossover_point:]
        return child1, child2
    else:
        return parent1, parent2

def mutate(individual, mutation_prob):
    mutated_individual = list(individual)
    for i in range(len(mutated_individual)):
        if random.random() < mutation_prob:
            mutated_individual[i] = '1' if mutated_individual[i] == '0' else '0'
    return ''.join(mutated_individual)


# Genetic Algorithm
def genetic_algorithm(population_size, num_generations, crossover_prob, mutation_prob, e
    num_bits = 10  # Number of bits for binary encoding
    binary_precision = 2**num_bits - 1
    min_value = -5.12  # Define the lower bound for DeJong functions
    max_value = 5.12   # Define the upper bound for DeJong functions

    # Initialize population
    population = [''.join(random.choice('01') for _ in range(num_bits)) for _ in range(p

    for generation in range(num_generations):
        # Evaluate fitness
        fitness_scores = []
        for individual in population:
            fitness_scores.append(evaluation_function(list(individual), num_bits, min_va
```

```
        # Select parents based on fitness
        parents = random.choices(population, weights=fitness_scores, k=population_size)

        # Create new generation
        new_population = []
        for i in range(0, population_size, 2):
            child1, child2 = crossover(parents[i], parents[i+1], crossover_prob)
            new_population.extend([mutate(child1, mutation_prob), mutate(child2, mutatio

        population = new_population

    # Return best solution
    best_individual = max(population, key=lambda x: evaluation_function(list(x), num_bit
    best_fitness = evaluation_function(list(best_individual), num_bits,min_value, max_va
    return best_individual, best_fitness


def genetic_algorithm_with_plot(population_size, num_generations, crossover_prob, mutati
    random.seed(random_seed)

    num_bits = 10
    binary_precision = 2**num_bits - 1
    min_value = -5.12
    max_value = 5.12

    population = [''.join(random.choice('01') for _ in range(num_bits)) for _ in range(p

    best_fitnesses = []

    for generation in range(num_generations):
        fitness_scores = []

        for individual in population:
            fitness_scores.append(evaluation_function(list(individual), num_bits, min_va

        parents = random.choices(population, weights=fitness_scores, k=population_size)

        new_population = []

        for i in range(0, population_size, 2):
            child1, child2 = crossover(parents[i], parents[i+1], crossover_prob)
            new_population.extend([mutate(child1, mutation_prob), mutate(child2, mutatio

        population = new_population

        best_individual = max(population, key=lambda x: evaluation_function(list(x), num
        best_fitness = evaluation_function(list(best_individual), num_bits, min_value, m
        best_fitnesses.append(best_fitness)

    return best_fitnesses


# Define a list of DeJong functions and their names
dejong_functions = [
    (sphere, "Sphere Function"),
    (step, "Step Function"),
    (rosenbrock, "Rosenbrock Function"),
    (quartic, "Quartic Function"),
    (foxholes, "Foxholes Function")
]

# Define the parameter combinations to experiment with
parameter_combinations = [
    (50, 100, 0.7, 0.001),
```

```python
    (100, 200, 0.8, 0.002),
    (30, 150, 0.6, 0.001),
    (50, 75, 0.95, 0.05),
    # Add more combinations as needed
]

# Loop over each DeJong function
for dejong_function, function_name in dejong_functions:
    #print(f"Running Experiments for {function_name}\033...")
    print(f"Running Experiments for \033[1m{function_name}\033[0m...")

    for population_size, num_generations, crossover_prob, mutation_prob in parameter_com
        print(f"Parameters: Population Size={population_size}, Generations={num_generati

        # Run the genetic algorithm
        best_solution, best_fitness = genetic_algorithm(population_size=population_size,

        print(f"Best Solution for {function_name}: {best_solution}")
        print(f"Best Fitness for {function_name}: {best_fitness}\n")
```

```
Running Experiments for Sphere Function...
Parameters: Population Size=50, Generations=100, Crossover Probability=0.7, Mutation Pro
bability=0.001
Best Solution for Sphere Function: 1111110111
Best Fitness for Sphere Function: 25.400811736320733

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Pr
obability=0.002
Best Solution for Sphere Function: 1111111110
Best Fitness for Sphere Function: 26.112000097847268

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Pro
bability=0.001
Best Solution for Sphere Function: 0000001100
Best Fitness for Sphere Function: 24.998826993231912

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Pro
bability=0.05
Best Solution for Sphere Function: 1111111100
Best Fitness for Sphere Function: 25.907801467135652

Running Experiments for Step Function...
Parameters: Population Size=50, Generations=100, Crossover Probability=0.7, Mutation Pro
bability=0.001
Best Solution for Step Function: 0000010000
Best Fitness for Step Function: 4.959843597262952

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Pr
obability=0.002
Best Solution for Step Function: 0000000001
Best Fitness for Step Function: 5.109990224828935

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Pro
bability=0.001
Best Solution for Step Function: 1111010111
Best Fitness for Step Function: 4.71960899315738

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Pro
bability=0.05
Best Solution for Step Function: 1111111110
Best Fitness for Step Function: 5.109990224828935

Running Experiments for Rosenbrock Function...
Parameters: Population Size=50, Generations=100, Crossover Probability=0.7, Mutation Pro
bability=0.001
Best Solution for Rosenbrock Function: 0000000000
```

```
Best Fitness for Rosenbrock Function: 37.4544

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Pr
obability=0.002
Best Solution for Rosenbrock Function: 0000000000
Best Fitness for Rosenbrock Function: 37.4544

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Pro
bability=0.001
Best Solution for Rosenbrock Function: 0000000001
Best Fitness for Rosenbrock Function: 37.33198054750514

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Pro
bability=0.05
Best Solution for Rosenbrock Function: 0000000000
Best Fitness for Rosenbrock Function: 37.4544

Running Experiments for Quartic Function...
Parameters: Population Size=50, Generations=100, Crossover Probability=0.7, Mutation Pro
bability=0.001
Best Solution for Quartic Function: 0000000001
Best Fitness for Quartic Function: 2.6634240199608428

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Pr
obability=0.002
Best Solution for Quartic Function: 1111111111
Best Fitness for Quartic Function: 2.68435456

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Pro
bability=0.001
Best Solution for Quartic Function: 0000000111
Best Fitness for Quartic Function: 2.540399442096086

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Pro
bability=0.05
Best Solution for Quartic Function: 0000000000
Best Fitness for Quartic Function: 2.68435456

Running Experiments for Foxholes Function...
Parameters: Population Size=50, Generations=100, Crossover Probability=0.7, Mutation Pro
bability=0.001
Best Solution for Foxholes Function: 0100000100
Best Fitness for Foxholes Function: 0.9997511318421858

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Pr
obability=0.002
Best Solution for Foxholes Function: 1001111011
Best Fitness for Foxholes Function: 0.9999398071374573

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Pro
bability=0.001
Best Solution for Foxholes Function: 0110000101
Best Fitness for Foxholes Function: 0.9984032536783233

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Pro
bability=0.05
Best Solution for Foxholes Function: 0100000110
Best Fitness for Foxholes Function: 1.0000001532019536
```

In [6]:

```
import random
import matplotlib.pyplot as plt
import numpy as np

# ... (previous code remains the same)
```

```
# Define the parameter combinations to experiment with
parameter_combinations = [
    (50, 100, 0.7, 0.001),
    (100, 200, 0.8, 0.002),
    (30, 150, 0.6, 0.001),
    (50, 75, 0.95, 0.05),
    # Add more combinations as needed
]

# Loop over each DeJong function
for dejong_function, function_name in dejong_functions:
    print(f"Running Experiments for \033[1m{function_name}\033[0m...")

    avg_min_fitness = np.zeros(100)
    avg_max_fitness = np.zeros(100)
    avg_avg_fitness = np.zeros(100)

    avg_min_obj_value = np.zeros(100)
    avg_max_obj_value = np.zeros(100)
    avg_avg_obj_value = np.zeros(100)

    for seed in range(30):

        # Get min, max and avg fitnesses for this seed
        min_fitnesses = np.array(genetic_algorithm_with_plot(population_size=50, num_gen
        max_fitnesses = np.array(genetic_algorithm_with_plot(population_size=100, num_ge
        avg_fitnesses = np.array(genetic_algorithm_with_plot(population_size=150, num_ge

        avg_min_fitness += min_fitnesses
        avg_max_fitness += max_fitnesses
        avg_avg_fitness += avg_fitnesses

        min_obj_values = min_fitnesses**0.5
        max_obj_values = max_fitnesses**0.5
        avg_obj_values = avg_fitnesses**0.5

        avg_min_obj_value += min_obj_values
        avg_max_obj_value += max_obj_values
        avg_avg_obj_value += avg_obj_values

    avg_min_fitness /= 30
    avg_max_fitness /= 30
    avg_avg_fitness /= 30

    avg_min_obj_value /= 30
    avg_max_obj_value /= 30
    avg_avg_obj_value /= 30

    # Plot fitness progression
    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)
    plt.plot(range(100), avg_min_fitness, label='Min Fitness')
    plt.plot(range(100), avg_max_fitness, label='Max Fitness')
    plt.plot(range(100), avg_avg_fitness, label='Avg Fitness')
    plt.xlabel('Generation')
    plt.ylabel('Fitness')
    plt.title(f'Fitness Progression - {function_name}')
    plt.legend()

    # Plot objective function values
    plt.subplot(1, 2, 2)
    plt.plot(range(100), avg_min_obj_value, label='Min Objective Value')
    plt.plot(range(100), avg_max_obj_value, label='Max Objective Value')
    plt.plot(range(100), avg_avg_obj_value, label='Avg Objective Value')
```

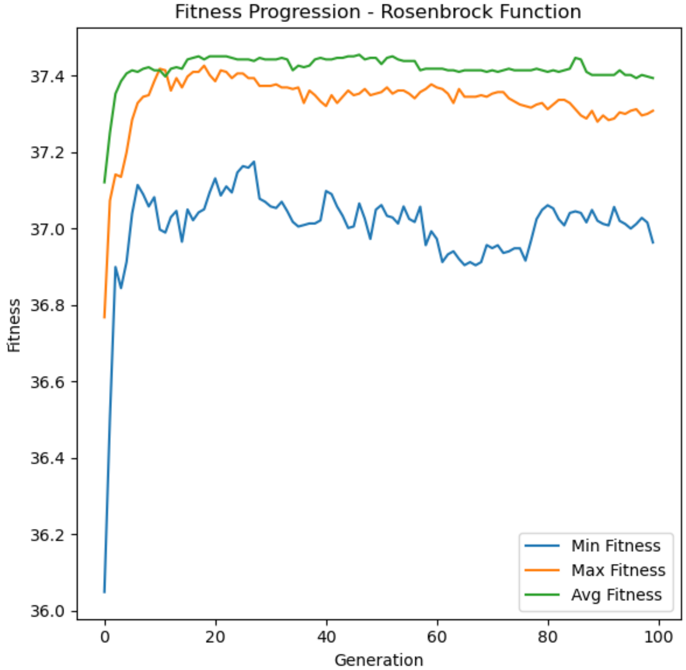
```
    plt.xlabel('Generation')
    plt.ylabel('Objective Value')
    plt.title(f'Objective Value Progression - {function_name}')
    plt.legend()

    plt.tight_layout()
    plt.show()
```

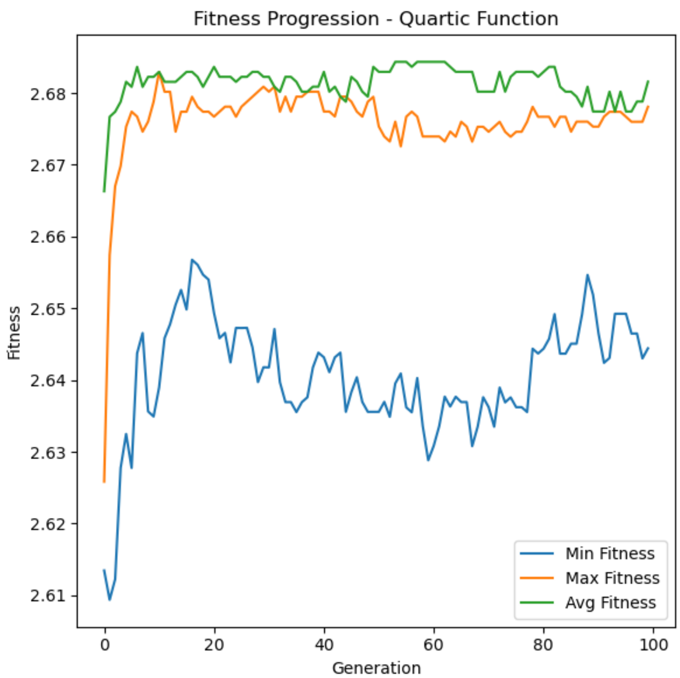
Running Experiments for **Sphere Function...**

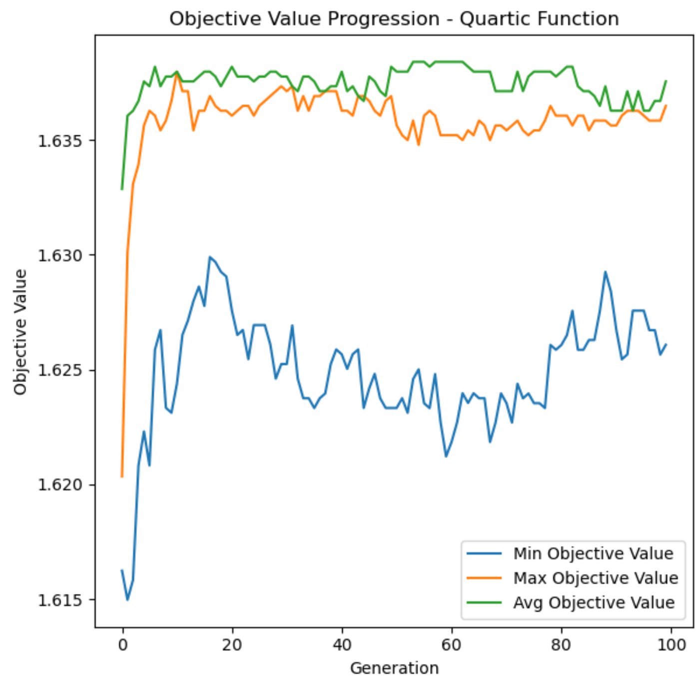
Running Experiments for **Step Function...**

Running Experiments for **Rosenbrock Function...**

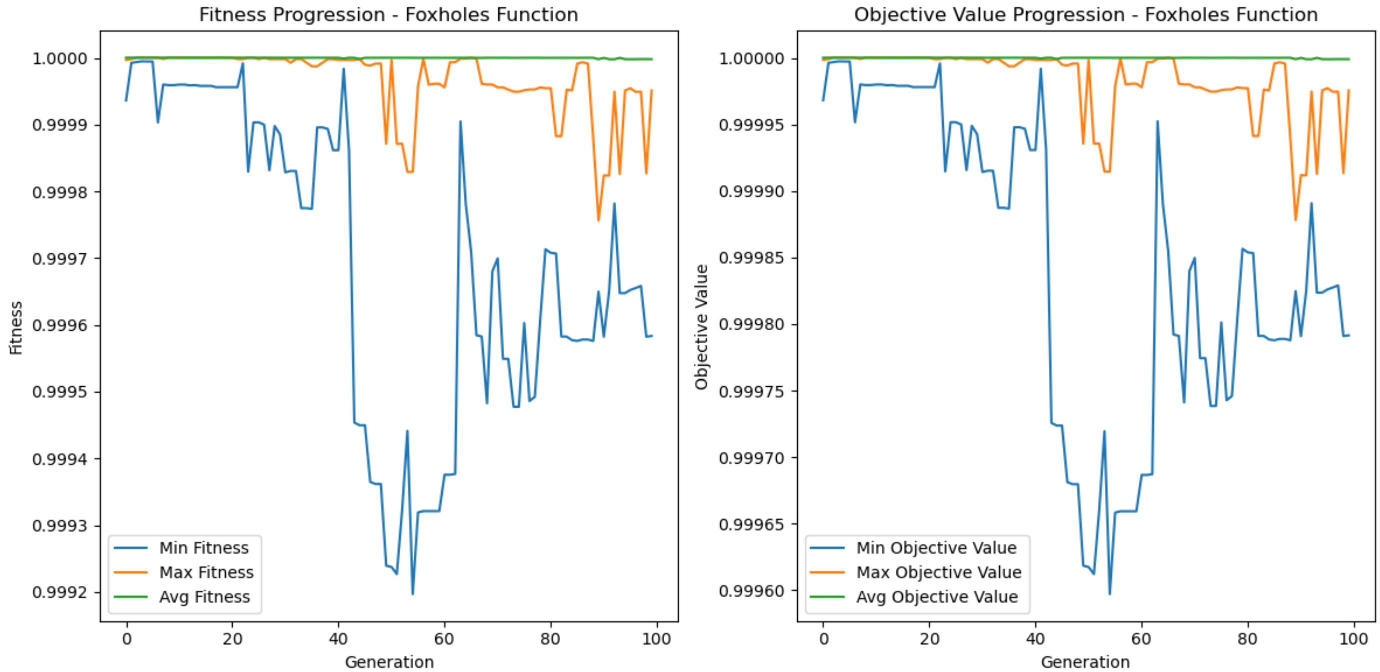


Running Experiments for **Quartic Function...**





Running Experiments for **Foxholes Function...**


Fitness Progression - Foxholes Function
Objective Value Progression - Foxholes Function
Fitness
Objective Value
Generation
Min Fitness
Max Fitness
Avg Fitness
Min Objective Value
Max Objective Value
Avg Objective Value
1.0000
0.9999
0.9998
0.9997
0.9996
0.9995
0.9994
0.9993
0.9992
1.00000
0.99995
0.99990
0.99985
0.99980
0.99975
0.99970
0.99965
0.99960
0
20
40
60
80
100

In [ ]: