In [97]:
```
# Simple Genetic Alogorithm

import random
import matplotlib.pyplot as plt

# 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)

# 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_v
```

```python
        # 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_v
    return best_individual, best_fitness


# 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),
    (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: 0000001100
Best Fitness for Sphere Function: 24.998826993231912

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Pr
obability=0.002
Best Solution for Sphere Function: 0000000000
Best Fitness for Sphere Function: 26.2144

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Pro
bability=0.001
Best Solution for Sphere Function: 0000000011
```

```
Best Fitness for Sphere Function: 25.907801467135645

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Probability=0.05
Best Solution for Sphere Function: 0000000000
Best Fitness for Sphere Function: 26.2144
```

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

```
Parameters: Population Size=50, Generations=100, Crossover Probability=0.7, Mutation Probability=0.001
Best Solution for Step Function: 0000000000
Best Fitness for Step Function: 5.12

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Probability=0.002
Best Solution for Step Function: 0000000000
Best Fitness for Step Function: 5.12

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Probability=0.001
Best Solution for Step Function: 0000000011
Best Fitness for Step Function: 5.089970674486803

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Probability=0.05
Best Solution for Step Function: 1111111101
Best Fitness for Step Function: 5.09998044965787
```

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

```
Parameters: Population Size=50, Generations=100, Crossover Probability=0.7, Mutation Probability=0.001
Best Solution for Rosenbrock Function: 0000010000
Best Fitness for Rosenbrock Function: 35.519735703836204

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Probability=0.002
Best Solution for Rosenbrock Function: 0000000001
Best Fitness for Rosenbrock Function: 37.33198054750514

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Probability=0.001
Best Solution for Rosenbrock Function: 0000000000
Best Fitness for Rosenbrock Function: 37.4544

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Probability=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 Probability=0.001
Best Solution for Quartic Function: 1111110110
Best Fitness for Quartic Function: 2.500354520179903

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Probability=0.002
Best Solution for Quartic Function: 0000000000
Best Fitness for Quartic Function: 2.68435456

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Probability=0.001
Best Solution for Quartic Function: 0000000000
Best Fitness for Quartic Function: 2.68435456

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Pro
```

```
bability=0.05
Best Solution for Quartic Function: 1111111100
Best Fitness for Quartic Function: 2.621930378361392

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: 0100000111
Best Fitness for Foxholes Function: 0.9999654522987196

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Pr
obability=0.002
Best Solution for Foxholes Function: 0110000011
Best Fitness for Foxholes Function: 1.0000003386318617

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Pro
bability=0.001
Best Solution for Foxholes Function: 1001111100
Best Fitness for Foxholes Function: 1.0000003386318617

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Pro
bability=0.05
Best Solution for Foxholes Function: 0110000011
Best Fitness for Foxholes Function: 1.0000003386318617
```

In [98]:

```python
# CHC Genetic Alogorithm
import random

# DeJong Evaluation Functions
def sphere(chromosome):
    value = int(''.join(map(str, chromosome)), 2) / (2**len(chromosome) - 1) * (5.12 - (
    return value**2

def step(chromosome):
    value = int(''.join(map(str, chromosome)), 2) / (2**len(chromosome) - 1) * (5.12 - (
    return abs(value)

def rosenbrock(chromosome):
    value = int(''.join(map(str, chromosome)), 2) / (2**len(chromosome) - 1) * (5.12 - (
    return (1 - value)**2

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

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

# Crossover function (uniform crossover)
def crossover(parent1, parent2, prob):
    child1 = []
    child2 = []
    for bit1, bit2 in zip(parent1, parent2):
        if random.random() < prob:
            child1.append(bit2)
            child2.append(bit1)
        else:
            child1.append(bit1)
            child2.append(bit2)
    return ''.join(child1), ''.join(child2)

# Mutation function (bit-flip mutation)
```

```python
def mutate(chromosome, mutation_rate):
    mutated_chromosome = list(chromosome)
    for i in range(len(mutated_chromosome)):
        if random.random() < mutation_rate:
            mutated_chromosome[i] = '1' if chromosome[i] == '0' else '0'
    return ''.join(mutated_chromosome)

# CHC Adaptive Search Algorithm
def chc_adaptive_search(chromosome_length, population_size, divergence_rate, dejong_func
    population = [''.join(random.choice('01') for _ in range(chromosome_length)) for _ i
    best_individual = max(population, key=dejong_func)
    d = chromosome_length // 4
    t = 0
    best_solution = None  # Initialize best_solution to None
    best_fitness = float('inf')

    while t < divergence_rate:
        # Crossover
        children = [crossover(parent1, parent2, crossover_prob) for parent1, parent2 in
        children = [mutate(child, mutation_prob) for child in children]  # Apply mutatio

        # Evaluate fitness of children
        children_fitness = [dejong_func(list(map(int, child))) for child in children]

        # Select best individuals from parents and children
        combined_population = population + children
        new_population = sorted(combined_population, key=dejong_func, reverse=True)[:pop

        # If new population is the same as the previous population, decrement d
        if new_population == population:
            d -= 1
            if d < 0:
                population = [''.join(random.choice('01') for _ in range(chromosome_leng
                d = chromosome_length // 4
        else:
            d = chromosome_length // 4

        # Set the best individual from the previous generation
        new_population[0] = best_individual

        # Update population
        population = new_population

        t += 1

        # Track the best solution of the current generation
        current_best_individual = max(population, key=dejong_func)
        current_best_fitness = dejong_func(list(map(int, current_best_individual)))

        # Update the best_solution if it has the same chromosome length
        if len(current_best_individual) == chromosome_length:
            if best_solution is None or current_best_fitness > dejong_func(list(map(int,
                best_solution = current_best_individual

    # Evaluate the fitness of the best_solution
    best_fitness = dejong_func(list(map(int, best_solution))) if best_solution else None

    return best_solution, best_fitness

# Define a stopping condition function (you should define this based on your specific pr
def stopping_condition_met():
    # Define your own stopping condition logic here
    return False

# Define parameters
chromosome_length = 10
```

```python
divergence_rate = 100

# 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),
    (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 CHC genetic algorithm

        best_solution, best_fitness = chc_adaptive_search(chromosome_length, population_

        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: 0000000000
Best Fitness for Sphere Function: 26.2144

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Pr
obability=0.002
Best Solution for Sphere Function: 0000001001
Best Fitness for Sphere Function: 25.300014963751604

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Pro
bability=0.001
Best Solution for Sphere Function: 0000000000
Best Fitness for Sphere Function: 26.2144

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Pro
bability=0.05
Best Solution for Sphere Function: 0000000100
Best Fitness for Sphere Function: 25.806002738576762

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: 1111110011
Best Fitness for Step Function: 4.999882697947215

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: 1111111101
Best Fitness for Step Function: 5.09998044965787

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Pro
bability=0.05
Best Solution for Step Function: 1111110000
Best Fitness for Step Function: 4.9698533724340175
```

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: 0000001010
Best Fitness for Rosenbrock Function: 36.23922307895911

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Pro
bability=0.001
Best Solution for Rosenbrock Function: 0000111010
Best Fitness for Rosenbrock Function: 30.68531840551002

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Pro
bability=0.05
Best Solution for Rosenbrock Function: None
Best Fitness for Rosenbrock Function: None
```

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: 1111110111
Best Fitness for Quartic Function: 2.520317331500036

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Pr
obability=0.002
Best Solution for Quartic Function: 1111111011
Best Fitness for Quartic Function: 2.6013663177477806

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Pro
bability=0.001
Best Solution for Quartic Function: 0000010001
Best Fitness for Quartic Function: 2.3448901921121506

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: 0100000101
Best Fitness for Foxholes Function: 0.9999986461815477

Parameters: Population Size=100, Generations=200, Crossover Probability=0.8, Mutation Pr
obability=0.002
Best Solution for Foxholes Function: 1001111100
Best Fitness for Foxholes Function: 1.0000003386318617

Parameters: Population Size=30, Generations=150, Crossover Probability=0.6, Mutation Pro
```

```
bability=0.001
Best Solution for Foxholes Function: 0100000101
Best Fitness for Foxholes Function: 0.9999986461815477

Parameters: Population Size=50, Generations=75, Crossover Probability=0.95, Mutation Pro
bability=0.05
Best Solution for Foxholes Function: 1001111011
Best Fitness for Foxholes Function: 0.9999398071374573
```

In [99]:

```python
import random
import matplotlib.pyplot as plt

# 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)

# 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, 10, 100),
    (100, 200, 0.8, 0.002, 10, 100),
    (30, 150, 0.6, 0.001, 10, 100),
    (50, 75, 0.95, 0.05, 10, 100),
    # Add more combinations as needed
]

# Create a dictionary to store results
results = {}

# Loop over each DeJong function
for dejong_function, function_name in dejong_functions:
    results[function_name] = {}

    for population_size, num_generations, crossover_prob, mutation_prob, chromosome_leng
```

```
        #print(f"Parameters: Population Size={population_size}, Generations={num_generat
        avg_evaluations_sga = 0
        avg_evaluations_chc_ga = 0

        # Run SGA and CHC-GA multiple times
        num_runs = 10  # Adjust as needed
        for _ in range(num_runs):

            # Run SGA
            best_solution_sga, best_fitness_sga = genetic_algorithm(population_size=popu
            avg_evaluations_sga += best_fitness_sga

            # Run CHC-GA
            #best_solution_chc_ga, best_fitness_chc_ga = chc_adaptive_search(chromosome_
            #avg_evaluations_chc_ga += best_fitness_chc_ga

        # Calculate average for SGA and CHC-GA
        avg_evaluations_sga /= num_runs
        avg_evaluations_chc_ga /= num_runs

        # Store results
        #results[function_name][str(parameters)] = (avg_evaluations_sga)
        results[function_name][(population_size, num_generations, crossover_prob, mutati

# Print the results in a table format
table = PrettyTable()
table.field_names = ["DeJong Function", "Population Size", "Generations", "Crossover Pro

for function_name, parameter_results in results.items():
    for parameters, avg_evaluations_sga in parameter_results.items():
        table.add_row([function_name] + list(parameters) + [f"{avg_evaluations_sga:.2f}"

print(table)
```

DeJong Function	Population Size	Generations	Crossover Prob	Mutation Prob	Avg Func Evals (SGA)
Sphere Function	50	100	0.7	0.001	25.87
Sphere Function	100	200	0.8	0.002	26.19
Sphere Function	30	150	0.6	0.001	25.00
Sphere Function	50	75	0.95	0.05	26.19
Step Function	50	100	0.7	0.001	5.06
Step Function	100	200	0.8	0.002	5.12
Step Function	30	150	0.6	0.001	5.07
Step Function	50	75	0.95	0.05	5.08
Rosenbrock Function	50	100	0.7	0.001	36.87
Rosenbrock Function	100	200	0.8	0.002	37.41
Rosenbrock Function	30	150	0.6	0.001	37.06
Rosenbrock Function	50	75	0.95	0.05	37.37
Quartic Function	50	100	0.7	0.001	2.65

```
|   Quartic Function  |        100        |      200      |       0.8       |      0.002      |
        2.67         |
|   Quartic Function  |         30        |      150      |       0.6       |      0.001      |
        2.63         |
|   Quartic Function  |         50        |       75      |       0.95      |       0.05      |
        2.67         |
|  Foxholes Function  |         50        |      100      |       0.7       |      0.001      |
        1.00         |
|  Foxholes Function  |        100        |      200      |       0.8       |      0.002      |
        1.00         |
|  Foxholes Function  |         30        |      150      |       0.6       |      0.001      |
        1.00         |
|  Foxholes Function  |         50        |       75      |       0.95      |       0.05      |
        1.00         |
+---------------------+-------------------+---------------+-----------------+-----------------+
----------------------+
```

In [100…

```
import random
from prettytable import PrettyTable

# DeJong Evaluation Functions
def sphere(chromosome):
    value = int(''.join(map(str, chromosome)), 2) / (2**len(chromosome) - 1) * (5.12 - (
    return value**2

def step(chromosome):
    value = int(''.join(map(str, chromosome)), 2) / (2**len(chromosome) - 1) * (5.12 - (
    return abs(value)

def rosenbrock(chromosome):
    value = int(''.join(map(str, chromosome)), 2) / (2**len(chromosome) - 1) * (5.12 - (
    return (1 - value)**2

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

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


# 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, 10, 100),
    (100, 200, 0.8, 0.002, 10, 100),
    (30, 150, 0.6, 0.001, 10, 100),
    (50, 75, 0.95, 0.05, 10, 100),
    # Add more combinations as needed
]

# Create a dictionary to store results
results = {}

# Loop over each DeJong function
```

```python
for dejong_function, function_name in dejong_functions:
    results[function_name] = {}

    for population_size, num_generations, crossover_prob, mutation_prob, chromosome_leng
        #print(f"Parameters: Population Size={population_size}, Generations={num_generat
        avg_evaluations_sga = 0.0
        avg_evaluations_chc_ga = 0.0

        # Run SGA and CHC-GA multiple times
        num_runs = 10  # Adjust as needed
        for _ in range(num_runs):

            # Run SGA
            #best_solution_sga, best_fitness_sga = genetic_algorithm(population_size=pop
            #avg_evaluations_sga += best_fitness_sga

            # Run CHC-GA
            best_solution_chc_ga, best_fitness_chc_ga = chc_adaptive_search(chromosome_l
            if best_fitness_chc_ga is not None:
                avg_evaluations_chc_ga += best_fitness_chc_ga

        # Calculate average for SGA and CHC-GA
        avg_evaluations_sga /= num_runs
        avg_evaluations_chc_ga /= num_runs

        # Store results
        results[function_name][(population_size, num_generations, crossover_prob, mutati

# Print the results in a table format
table = PrettyTable()
table.field_names = ["DeJong Function", "Population Size", "Generations", "Crossover Pro

for function_name, parameter_results in results.items():
    for parameters, avg_evaluations_chc_ga in parameter_results.items():
        table.add_row([function_name] + list(parameters) + [f"{avg_evaluations_chc_ga:.2

print(table)
```

```
+---------------------+-----------------+-------------+----------------+----------------+
-------------------------+
|   DeJong Function   | Population Size | Generations | Crossover Prob | Mutation Prob |
Avg Func Evals (CHC-GA) |
+---------------------+-----------------+-------------+----------------+----------------+
-------------------------+
|   Sphere Function   |        50       |     100     |      0.7       |     0.001      |
        25.29           |
|   Sphere Function   |       100       |     200     |      0.8       |     0.002      |
        25.87           |
|   Sphere Function   |        30       |     150     |      0.6       |     0.001      |
        24.43           |
|   Sphere Function   |        50       |      75     |      0.95      |      0.05      |
        25.77           |
|    Step Function    |        50       |     100     |      0.7       |     0.001      |
         4.97           |
|    Step Function    |       100       |     200     |      0.8       |     0.002      |
         5.10           |
|    Step Function    |        30       |     150     |      0.6       |     0.001      |
         4.99           |
|    Step Function    |        50       |      75     |      0.95      |      0.05      |
         5.02           |
| Rosenbrock Function |        50       |     100     |      0.7       |     0.001      |
        35.83           |
| Rosenbrock Function |       100       |     200     |      0.8       |     0.002      |
        35.45           |
| Rosenbrock Function |        30       |     150     |      0.6       |     0.001      |
        32.52           |
```

```
| Rosenbrock Function |        50        |      75      |      0.95      |      0.05      |
        34.98          |
|   Quartic Function  |        50        |     100      |      0.7       |     0.001      |
         2.51          |
|   Quartic Function  |       100        |     200      |      0.8       |     0.002      |
         2.31          |
|   Quartic Function  |        30        |     150      |      0.6       |     0.001      |
         2.46          |
|   Quartic Function  |        50        |      75      |      0.95      |      0.05      |
         2.52          |
|  Foxholes Function  |        50        |     100      |      0.7       |     0.001      |
         0.90          |
|  Foxholes Function  |       100        |     200      |      0.8       |     0.002      |
         1.00          |
|  Foxholes Function  |        30        |     150      |      0.6       |     0.001      |
         0.71          |
|  Foxholes Function  |        50        |      75      |      0.95      |      0.05      |
         1.00          |
+---------------------+------------------+--------------+----------------+----------------+
-----------------------+
```

In [ ]: