CSE 848 Home Assignment 4

Submitted by: Ritam Guha (MSU ID: guharita)

Date: March 15, 2021

1 Question

Write a binary-coded genetic algorithm (BGA) with binary tournament selection operator, one-point crossover operator and bit-wise mutation operators. No elite preservation is to be used. Apply the BGA code to solve two, 30-variable maximization problems constructed from 10, three-bit substrings ($s_i, i = 1, 2, ..., 10$). The structure of the overall function is given below:

$$F(s) = 10 \sum_{i=1}^{1} f(s_i) \tag{1}$$

where 30-bit string s is constructed from 10, three-bit substrings as $s = (s_1 \cup s_2 \cup \cup s_1 0)$. Two subfunctions as a function of three bits are defined below:

Problem 1: The subfunction f is a function of unitation u, defined as the number of 1s in the three-bit substring: $f(u) = u/3$. For example, $f(011) = f(101) = f(110) = 2/3 = 0.67$, as for these three substrings $u = 2$.

Problem2: $f(u) = 0.9 - u/3$ for $u < 3$, and $f(3) = 1$.

Notice that for both problems, the optimal string is the all-1 string $s* = (111...1)$ having $F(s*) = 100$.

Two construction procedures are used:

Construction 1: The first three bits of s are used to construct the first subproblem, the next three bits of s are used to construct the second subproblem, and so on. Thus, the final three bits (28^{th}, 29^{th} and 30^{th} bits) are used to construct 10^{th} subfunction.

Construction2: 1^{st},11^{th} and 21^{st} bits of s are used to construct the first subproblem,then 2^{nd}, 12^{th} and 22^{nd} bits of s are used to construct the second subproblem, and so on. The 10^{th} subfunction uses 10^{th}, 20^{th} and 30^{th} bits of s.

Response:

The given problem can be treated as an engineering function optimization problem where the function is given by $F(s)$. It is a maximization problem where the goal is to maximize the given function.

1.1 Solution Formulation

A Binary Genetic Algorithm (BGA) is used to solve this engineering function optimization problem. The fitness function for the BGA is the engineering function mentioned as $F(s)$. For 2 constructions and 2 problems, we have $(2 \times 2) = 4$ different objective function formulations.

The BGA consists of evolutionary operators like mutation, crossover and binary tournament selection. No elitist preservation strategy is to be used for solving these problems.

The hyperparameter combination used for constructing the BGA is provided in the following Table:

Hyper-parameter	**Value**
Number of Chromosomes	60
Number of Generations	200
Mutation Rate	$\frac{1}{30}$
Crossover Rate	0.9
Number of Runs	30

Table 1: Hyperparameter combination used for the BGA

1.2 Code

The BGA code used for this procedure is shown below:

```python
import numpy as np

def fitness(solution, construction=1, problem=1):
    # function for computing fitness based on the problem and construction
    dim = np.shape(solution)[0]
    val = 0.0
    if(construction == 1 and problem == 1):
        for i in range(0,dim,3):
            val += sum(solution[i:(i+3)])/3

    elif(construction == 1 and problem == 2):
        for i in range(0,dim,3):
            temp = sum(solution[i:(i+3)])
            if(temp == 3):
                val += 1
            else:
                val += 0.9 - (temp/3)

    elif(construction == 2 and problem == 1):
        for i in range(0, int(dim/3)):
            val += (solution[i]+solution[i+10]+solution[i+20])/3

    else:
        for i in range(0, int(dim/3)):
            temp = solution[i]+solution[i+10]+solution[i+20]
            if(temp == 3):
                val += 1
            else:
                val += 0.9 - (temp/3)

    return (10*val)


def mutation(solution, mut_prob=0.1):
    # function for mutation
```

```python
    dim = np.shape(solution)[0]
    for i in range(dim):
        if(np.random.rand()<mut_prob):
            solution[i] = 1-solution[i]

    return solution


def crossover(parent1, parent2):
    # performs single-point crossover
    dim = np.shape(parent1)[0]
    child1 = np.zeros(dim)
    child2 = np.zeros(dim)
    cross_point = np.random.randint(dim-2)+1

    for i in range(dim):
        if(i<cross_point):
            child1[i] = parent1[i]
            child2[i] = parent2[i]
        else:
            child1[i] = parent2[i]
            child2[i] = parent1[i]

    return child1, child2


def binary_tournament_selection(population, objective):
    # used for binary tournament selection
    num_pop, dim = np.shape(population)
    num_sub_pop = int(num_pop/2)
    shuffle_order = np.random.permutation(num_pop)
    population = population[shuffle_order, :]
    objective = objective[shuffle_order]
    sub_population = np.zeros((num_sub_pop, dim))
    sub_objective = np.zeros(num_sub_pop)

    for i in range(0, num_pop, 2):
        pos_idx = int(i/2)

        if(objective[i] > objective[i+1]):
            sub_population[pos_idx,:] = population[i, :]
            sub_objective[pos_idx] = objective[i]

        else:
            sub_population[pos_idx,:] = population[i+1, :]
            sub_objective[pos_idx] = objective[i+1]

    return sub_population, sub_objective


def initialization(pop_size, dim):
    # initialize the population
    population = np.random.randint(low=0, high=2, size=(pop_size,dim))

```

```python
    return population


def count_competitors(population, construction=1):
    # helper function to count the number of competitors in each generation
    pop_size, dim = np.shape(population)
    comp_scores = np.zeros((2, 10))

    if(construction == 1):
        for i in range(0, 10):
            for pop_no in range(pop_size):
                if(sum(population[pop_no][(i*3):(i*3+3)])==3):
                    comp_scores[0][i] += 1

                if(sum(population[pop_no][(i*3):(i*3+3)])==0):
                    comp_scores[1][i] += 1

    else:
        for i in range(0, 10):
            for pop_no in range(pop_size):
                if((population[pop_no][i] + population[pop_no][i+10] + population[
                    pop_no][i+20])==3):
                    comp_scores[0][i] += 1

                if((population[pop_no][i] + population[pop_no][i+10] + population[
                    pop_no][i+20])==0):
                    comp_scores[1][i] += 1

    comp_scores /= pop_size

    return comp_scores



def BGA(pop_size=60, dim=30, num_gen=200, mut_rate=1/30, cross_rate=0.9,
    construction=1, problem=1):
    # driver function for the binary genetic algorithm
    population = initialization(pop_size, dim)
    obj_values = np.zeros(pop_size)
    best_values = np.zeros(num_gen)
    avg_values = np.zeros(num_gen)
    comp_scores = np.zeros((2, 10, num_gen))    # competitor scores

    for pop_no in range(pop_size):
        obj_values[pop_no] = fitness(population[pop_no], construction, problem)

    for iter_no in range(num_gen):
        population[0:int(pop_size/2), :], obj_values[0:int(pop_size/2)] =
            binary_tournament_selection(population, obj_values)
        child_idx = int(pop_size/2)

        while(child_idx+1 < pop_size):
            if(np.random.rand() < cross_rate):  # crossover occurring based on the
                probability
```

```
                pid1, pid2 = np.random.randint(low=0, high=int(pop_size/2), size=2)
                child1, child2 = crossover(population[pid1], population[pid2])

                child1 = mutation(child1)
                child2 = mutation(child1)
                obj_child1 = fitness(child1, construction, problem)
                obj_child2 = fitness(child2, construction, problem)

                population[child_idx] = child1
                obj_values[child_idx] = obj_child1

                population[child_idx+1] = child2
                obj_values[child_idx+1] = obj_child2

            child_idx += 2

        comp_scores[:, :, iter_no] = count_competitors(population, construction)

        best_values[iter_no] = np.max(obj_values)
        avg_values[iter_no] = np.mean(obj_values)

    return best_values, avg_values, comp_scores
```

The main function running the BGA code is presented below:

```
from BGA import BGA
import numpy as np
from matplotlib import pyplot as plt

# define hyper-parameters
pop_size = 60
dim = 30
num_gen = 200
mut_rate = 1/dim
cross_rate = 0.9
construction = 2
problem = 2
num_runs = 30

# initialize the variables
gen_best_values = np.zeros((num_runs, num_gen))
gen_avg_values = np.zeros((num_runs, num_gen))
best_values = np.zeros(num_runs)
comp_scores = np.zeros((2, 10, num_gen, num_runs))

# main run
for i in range(num_runs):
    gen_best_values[i,:], gen_avg_values[i,:], comp_scores[:, :, :, i] = BGA(pop_size, dim, num_gen, mut_rate, cross_rate, construction, problem)
    best_values[i] = gen_best_values[i, num_gen-1]

median_idx = np.argsort(best_values)[num_runs//2]

# Generating statistics for all the runs of BGA
```

```python
print('==========================================================')
print('Construction:{}, Problem:{}'.format(construction, problem))
print('==========================================================')
print('Best:{}'.format(np.max(best_values)))
print('Median:{}'.format(np.median(best_values)))
print('Mean:{}'.format(np.mean(best_values)))
print('Worst:{}'.format(np.min(best_values)))

fig = plt.figure()
X = [i for i in range(num_gen)]
Y1 = gen_best_values[median_idx,:]
Y2 = gen_avg_values[median_idx,:]
plt.plot(X, Y1)
plt.plot(X, Y2)

plt.legend(['Best Values', 'Avg Values'])
plt.xlabel('Generation Number')
plt.ylabel('Objective Value')
# plt.title('Generation-wise Objective Values for the Median Run of Problem:{}, Construction:{}'.format(problem, construction))
# plt.show()
fig.savefig('Problem_' + str(problem) + ' Construction_' + str(construction) + '/Plot.jpg')
print('==========================================================')


# Competitor plotting for median run
X = [i for i in range(num_gen)]

for i in range(10):
    Y1 = comp_scores[0, i, :, median_idx]
    Y0 = comp_scores[1, i, :, median_idx]

    fig = plt.figure()
    plt.plot(X, Y1)
    plt.plot(X, Y0)

    plt.legend(['1(' + str(i+1) + ')', '0(' + str(i+1) + ')'])
    plt.xlabel('Generation Number')
    plt.ylabel('#Competitors')
    # plt.title('Generation-wise trend for occurrences of competitors for different subproblems')
    # plt.show()
    fig.savefig('Problem_' + str(problem) + ' Construction_' + str(construction) + '/Comp_' + str(i+1) + '.jpg')
```

By running the code over 4 combinations of (problem, construction), I obtained the following results:

Table 2: Best, Median, Mean and Worst objective value obtained for 30 runs of BGA over different problem-construction settings.

Problem Statement	**Best**	**Median**	**Mean**	**Worst**
Problem-1 Construction-1	100	93.33	94.67	90
Problem-1 Construction-2	100	93.33	94.44	90
Problem-2 Construction-1	97	91.67	91.7	88.33
Problem-2 Construction-2	96	90	90.01	84.33

From the results, it is clear than Problem-1 is relatively easier to solve, while Problem-2 is harder. The code could not get to the global optimum solution for problem 2. Construction-wise, the result shows that construction 1 is easier to follow than construction 2.

Now let us discuss the problem-construction setting-wise discussion of BGA.

1.3 Problem-1

Problem 1 is easier to solve. This is because problem 1 only focuses on increasing the number of 1's in the candidate solutions and the global maximum is present at the position where all the variables are 1s. So, here, we can use some guidance from the better solutions (Here better solutions are the ones having more number of 1s) in forms of recombination or selection to move towards the global maximum solution. This problem is excellent for checking the exploitation capabilities of the algorithm.

1.3.1 Construction-1

The goal of the 1st construction is to correlate variables which are placed closer to each other in the candidate solutions (testing the *linkage* property). For Problem-1, Construction-1, the BGA I formulated worked really well and was able to achieve good performace in terms of the objective function. For the median run, out of all the 30 runs, the plot of the objective scores against the generations is presented below:

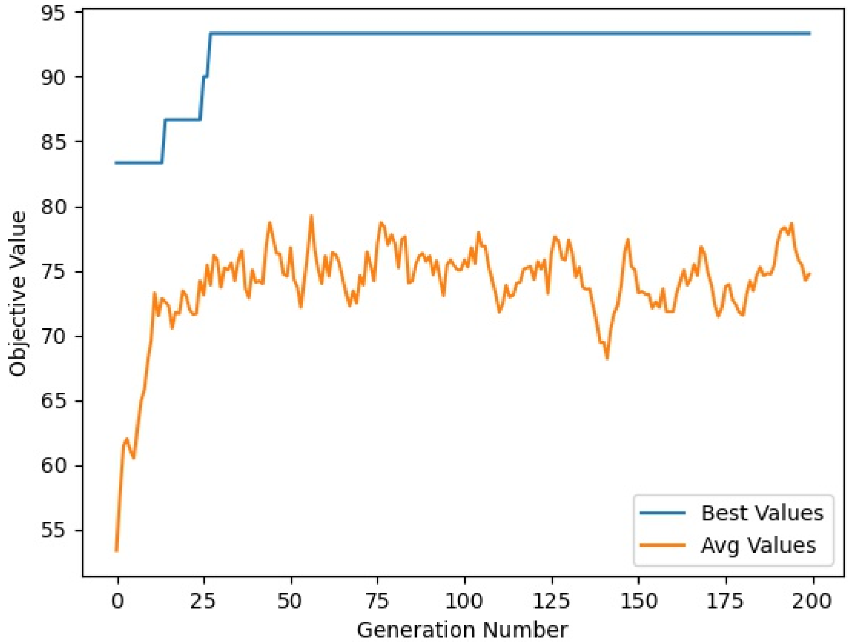

Figure 1: The best and average objective scores for the generations of the median run of BGA

From the plot, it is clear that, the algorithm always prefers better solutions over the course of generations. The best solution is always as good as the solution of the previous generation and sometimes better than that. Although the average objective score is fluctuating, in the long run, it is moving towards better objective scores.

Another interesting thing to see over here is the count of different competiting schemas present in the population across all the generations. For this reason I have plotted the fraction of competiting schemas for every generation in the next Figure. In Figure 2, 0(x) represents all 0s for subproblem x and 1(x) represents all 1s for subproblem x. So, these two are competiting schemas for subproblem x. As there are 10 subproblems in the given problem, we have 10 plots for competitors in the Figure. From the Figure, it is clear that the algorithm is prefering increasing number of 1s in the solutions over the 0s across different generations.

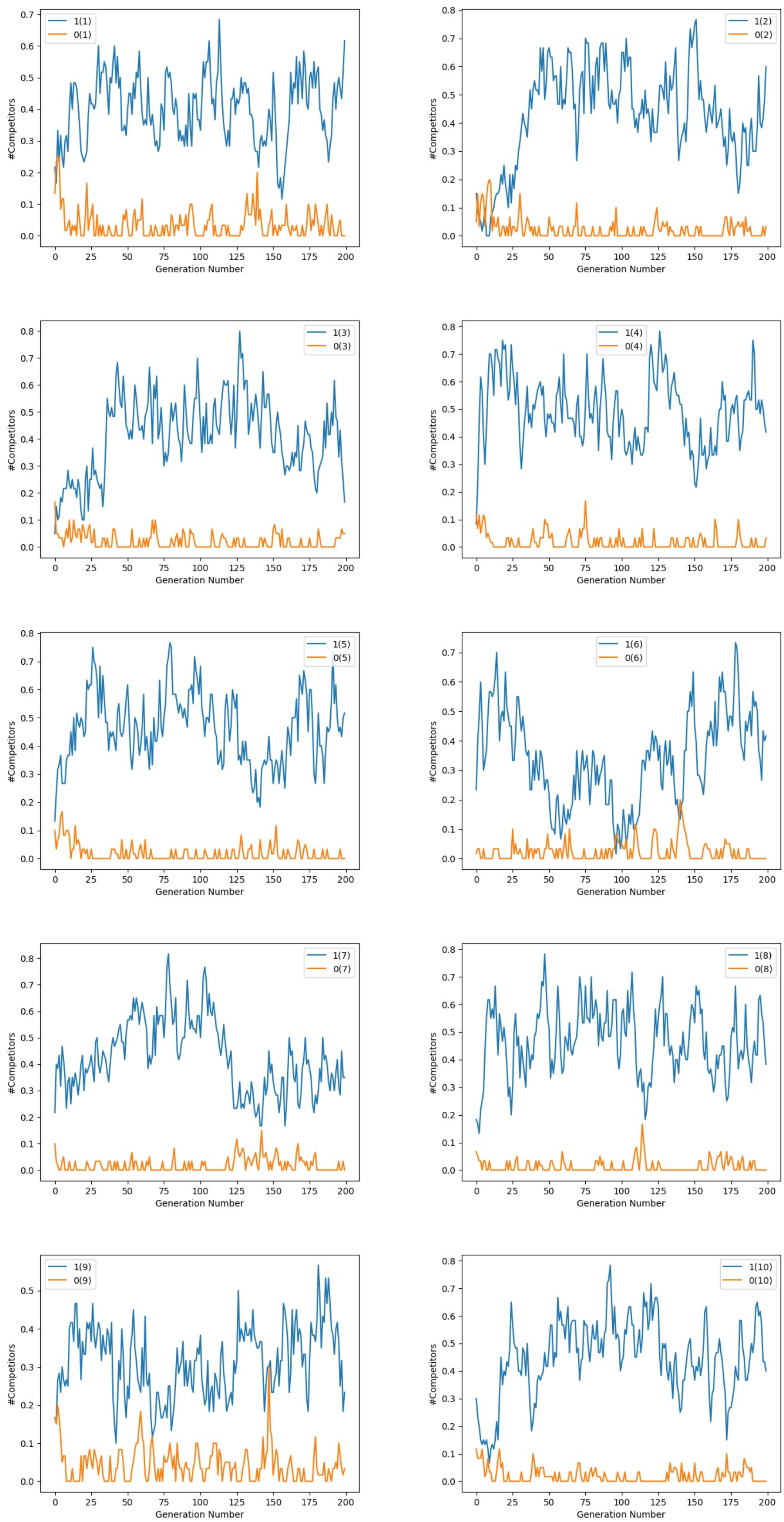

Figure 2: Plot of fraction of competiting schemas across the generations of the median run. In the legend, 1(x) and 0(x) mean all 1s and all 0s for subproblem x respectively.

1.3.2 Construction-2

Construction 2 tries to find distant relationships among variables. It takes variables at a distance of 10 from each other and combines them to provide the objective score. The goal of the construction is to model the concept of *epistasis* through the algorithm. Construction 2 for Problem 1 provides similar results as Construction 1. The objective scores over the generations for this setting is presented in Figure 3. The same trend of increasing objective score is observed in this plot too.

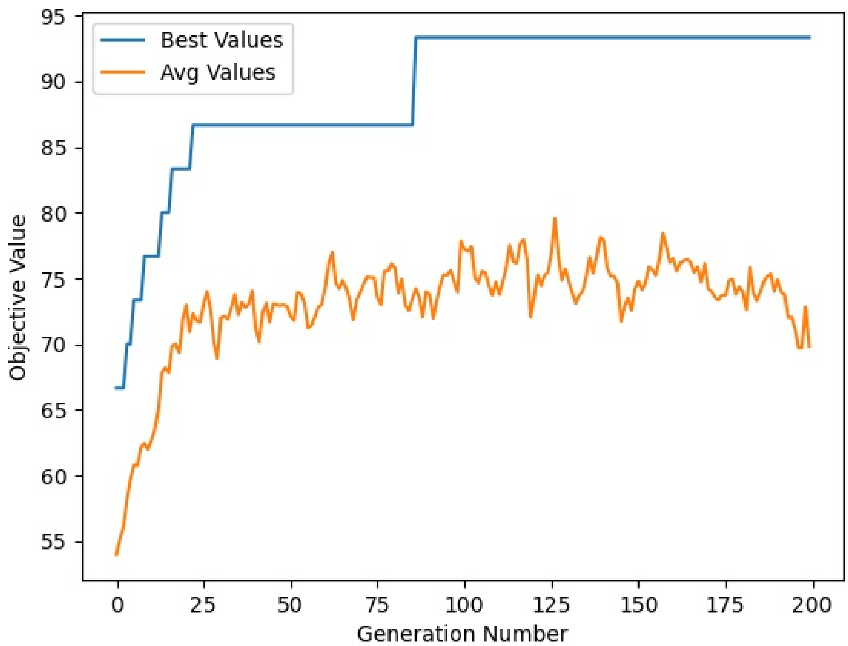

Figure 3: The best and average objective scores for the generations of the median run of BGA

The same trend is also continued in the schema competitors' plot which is shown in Figure 4. 1s are getting same kind of importance over 0s in the candidate solutions over the course of iterations.

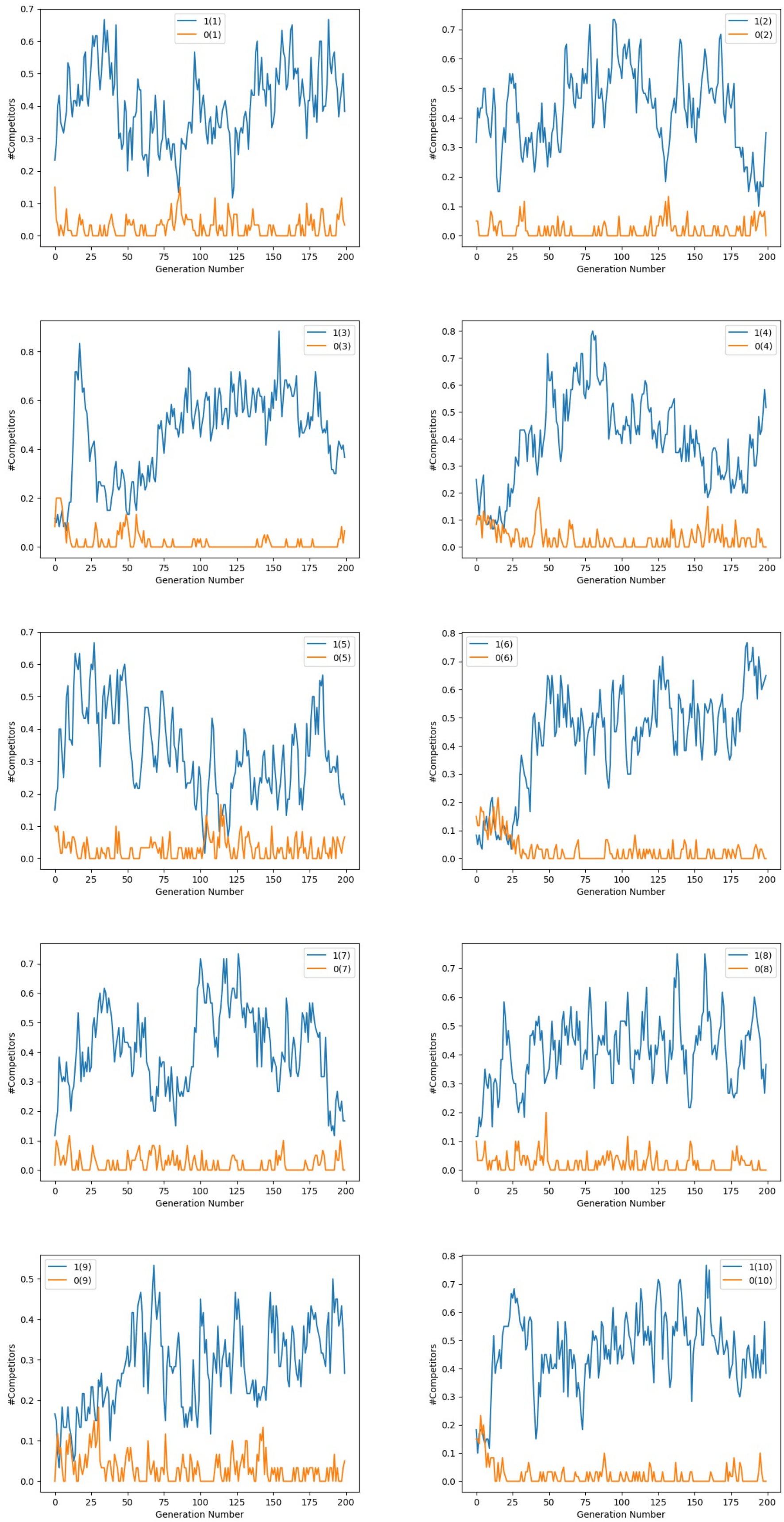

Figure 4: Plot of fraction of competiting schemas across the generations of the median run. In the legend, 1(x) and 0(x) mean all 1s and all 0s for subproblem x respectively.

1.4 Problem-2

Problem 2 is more complicated than Problem 1. In problem 2, 0s are preferred over 1s unless a subproblem has all 1s in its variables. This makes the situation worse because the problem formulation guides solution towards the direction opposite to the direction of the global maximum unless a solution is able to randomly find the global maximum. This problem focuses more on the explorational capabilities of the algorithm under consideration.

1.4.1 Construction-1

In Figure 5, we can see that the algorithm is still able to find decent solutions over the course of generations. But it was not able to achieve the best objective score. So, it could not get to the global maximum solution. This shows the deceptive nature of the problem 2.

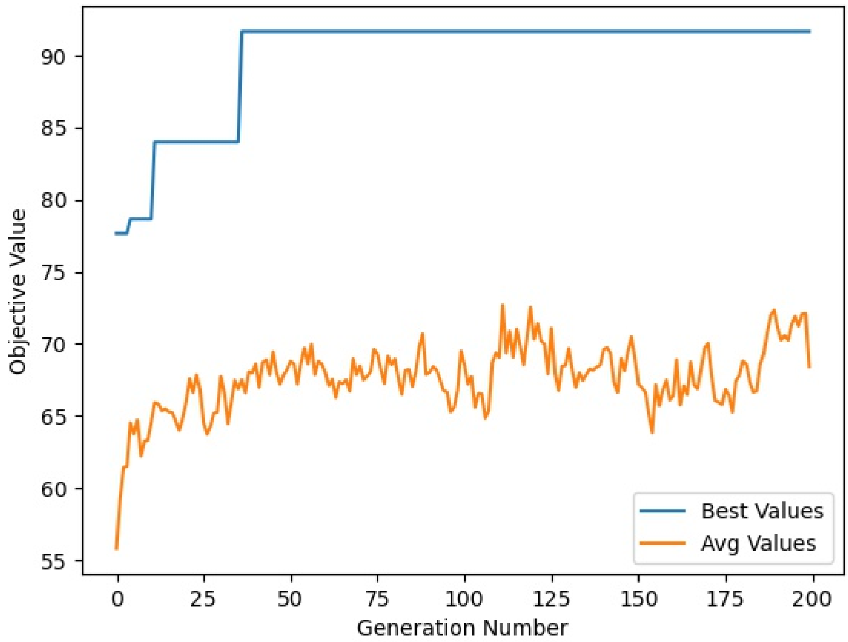

Figure 5: The best and average objective scores for the generations of the median run of BGA

The interesting thing about problem 2 starts getting noticed when we move our discussion to the fraction of schemas getting preferred over the generations. From Figure 6, it is visible how the algorithm got confused between selecting 1s and selecting 0s. Sometimes 0s got ahead of the 1s and sometimes it went the other way round for different subproblems. This trend is completely differnt from problem 1 which always favoured 1s over 0s.

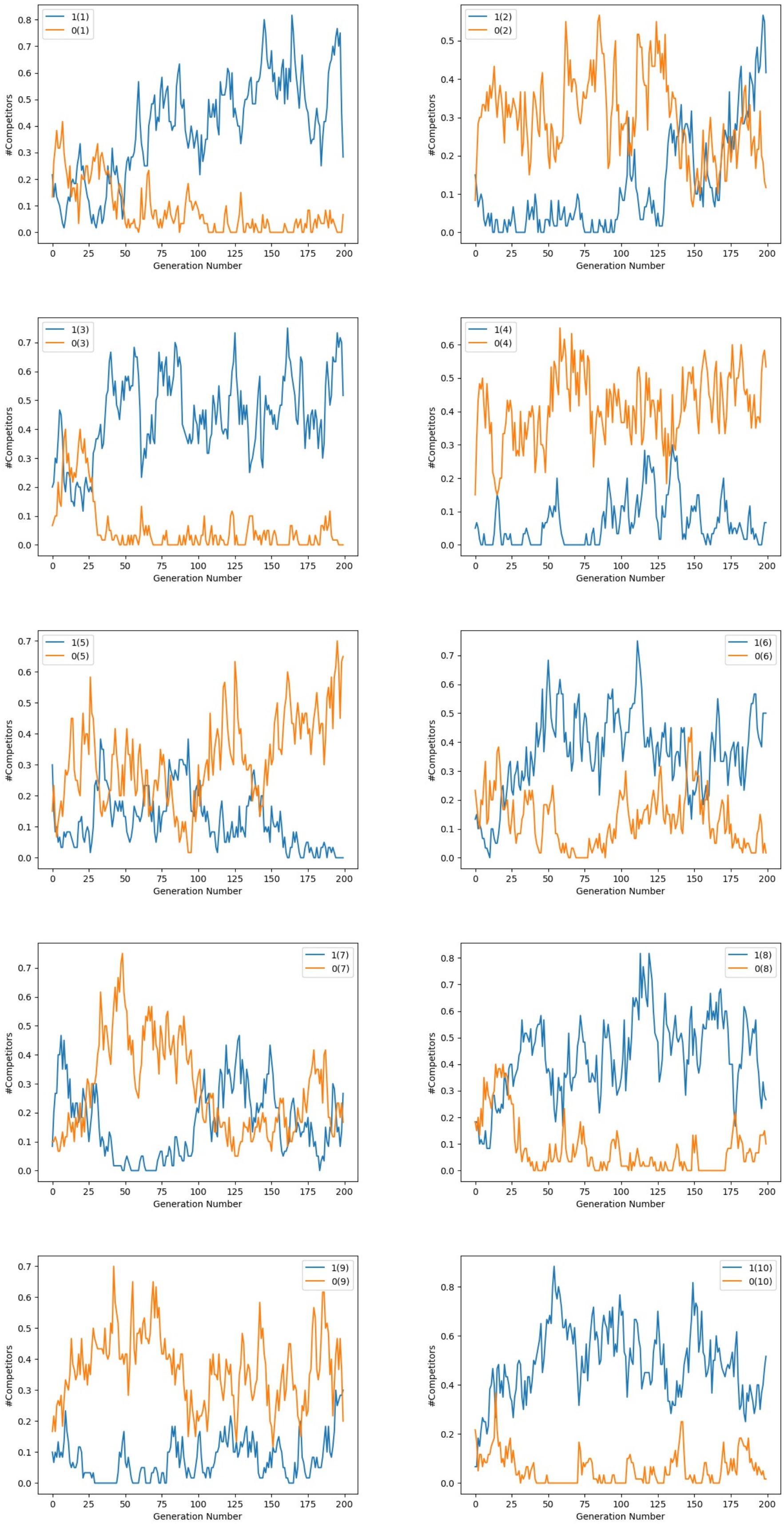

Figure 6: Plot of fraction of competiting schemas across the generations of the median run. In the legend, 1(x) and 0(x) mean all 1s and all 0s for subproblem x respectively.

1.4.2 Construction-2

Similar to construction 1 of for problem 2, contruction 2 also shows the same trend in terms of objective values and schemas. The objetive scores across the generations are plotted in Figure 7 and the competitor schemas are shown in Figure 8.

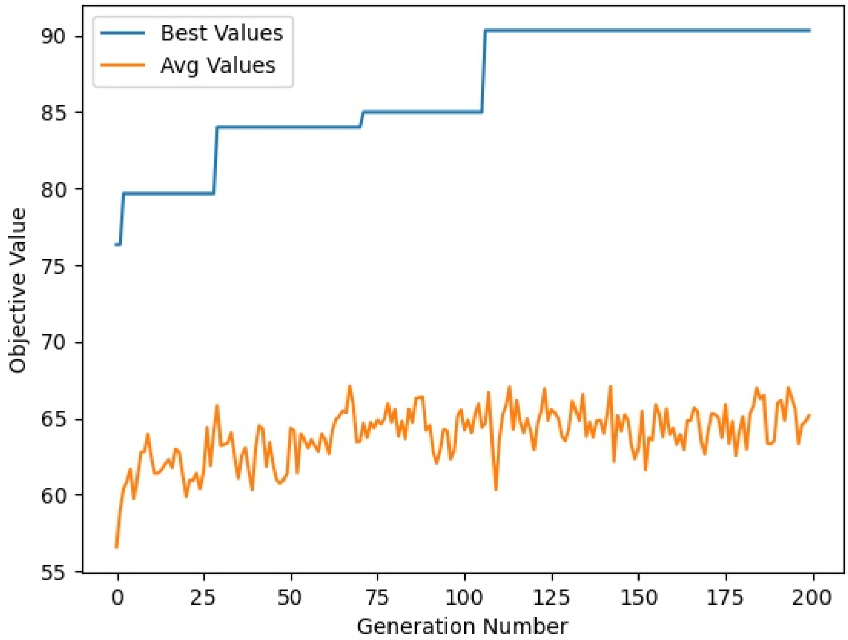

Figure 7: The best and average objective scores for the generations of the median run of BGA

Here also the algorithm got really confused between selecting 1s and 0s for different subproblems.

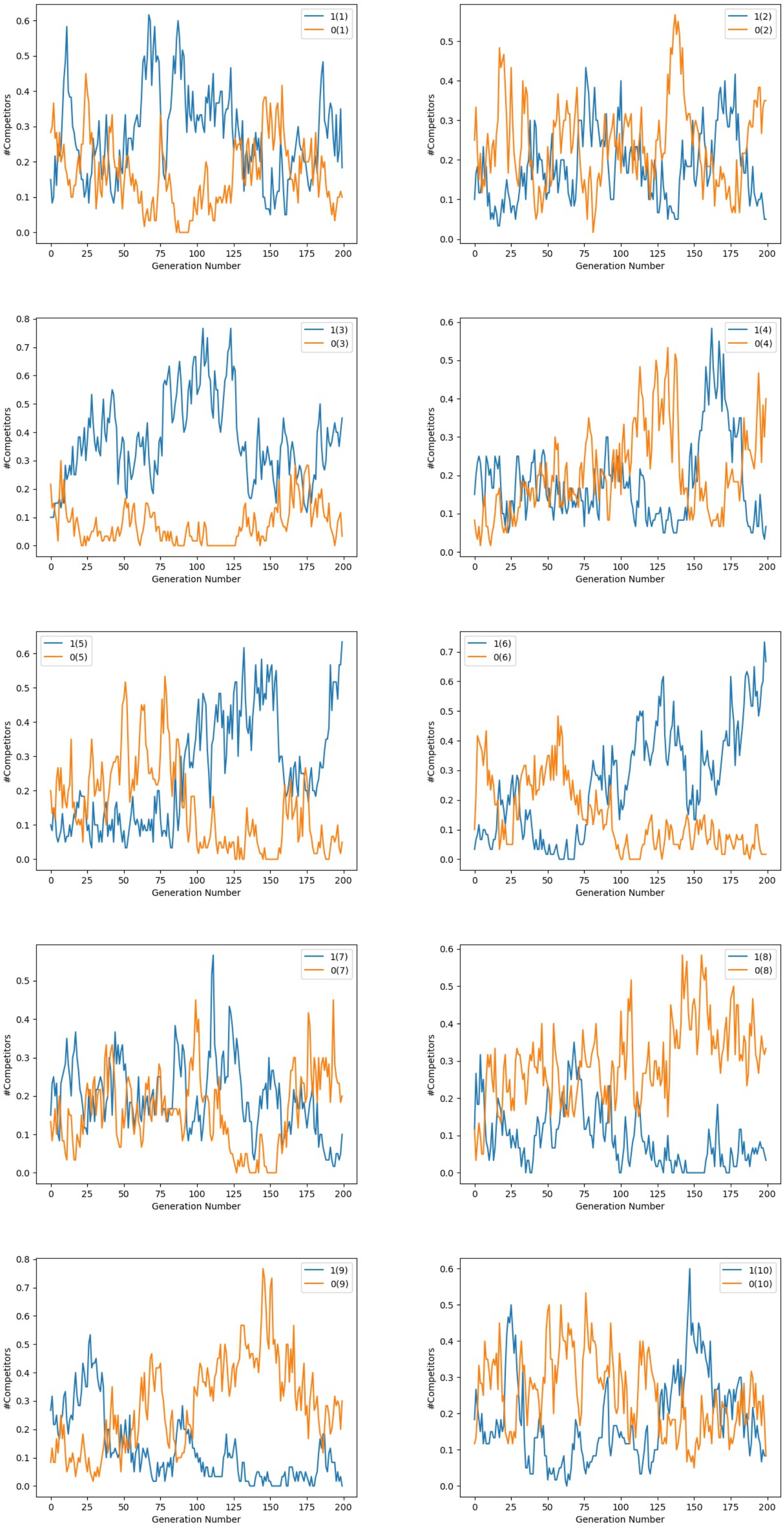

Figure 8: Plot of fraction of competiting schemas across the generations of the median run. In the legend, 1(x) and 0(x) mean all 1s and all 0s for subproblem x respectively.

1.5 Conclusion

In conclusion, I can say that the BGA implementation is able to find the global maximum solution for problem 1 easily but it is unable to find the same for problem 2. The exploration-exploitation trade-off of BGA can be checked using these 2 problems as problem 1 focuses mostly on exploitation, but problem 2 tests the explorational abilities of the algorithm too. The large crossover rate tries to enhance exploration, but still it was not able to find the global maximum.

According to my interpretation, the reason why the BGA could not reach the global maximum solution for problem 2 is because the global maximum solution is not in a stable equilibrium position. So, if even one bit for the global optimum solution gets changed, it deviates a lot in terms of the objective score. Also, one major point in terms of the algorithm is that it does not use elitist preservation. So, even if the global optimum was reached in the mid of the generations, it might get changed easily by the algorithm and would not be preserved due to the absence of the elitist preservation strategy.

The competitive schema preservation plots show how the algorithm got confused for problem 2. It is really interesting how the growth in schema can be changed in such a way just for different formulations of the objective function. The solution to this problem is to use separate schemes for the two problems. I think introducing operators like elitist preservations and parent-child comparison can help to solve problem 2.