CSE848: Evolutionary Computation
Michigan State University

Instructors: Professor Banzhaf, Deb, Goodman
Home Assignment 4

Jaturong Kongmanee

March 15, 2021

1 GA Operators

The GA parameters used in this experimental study are:

- Population size = 60
- Binary tournament selection
- Single-point crossover operator with a probability $p_c = 0.9$
- Bit-wise mutation operator with a probability $p_m = 1/30$
- The maximum generation = 200

2 Test Functions

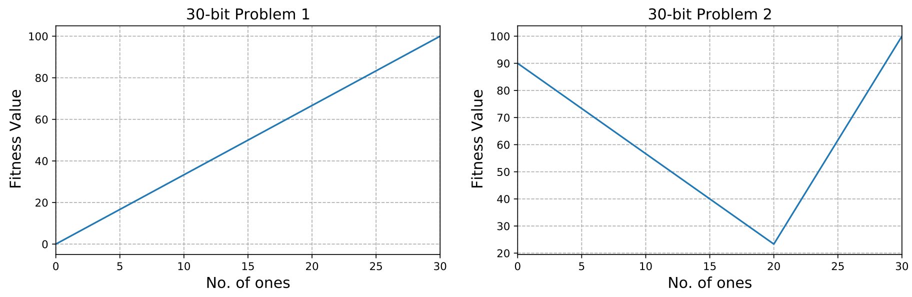

Figure 1: Two 30-bit subfunctions constructed from two construction procedures.

The problem 1 has a total of 2^{10} - 1 local maximum and only one global maximum (the string with all $1s$), which has a function value equal to 100. However, problem 2 is more complex and challenging to solve than linear problem 1 because the low-order building blocks tend to be guided towards local maximum (a string with all $0s$) rather than the global maximum solution (the string with all $1s$). In other words, problem 2 is used to make it challenging to find the best point (all-ones, 111 *...* 111), and easy to find the second-best (all-zeros, 000 *...* 000), leading to local maximum. Note that the optimal solution (a string with all $1s$) for both problems has a function value equal to 100. The plots showing the fitness value as a function of the #$1s$ in strings are shown in Figure 1.

3 Experimental Results

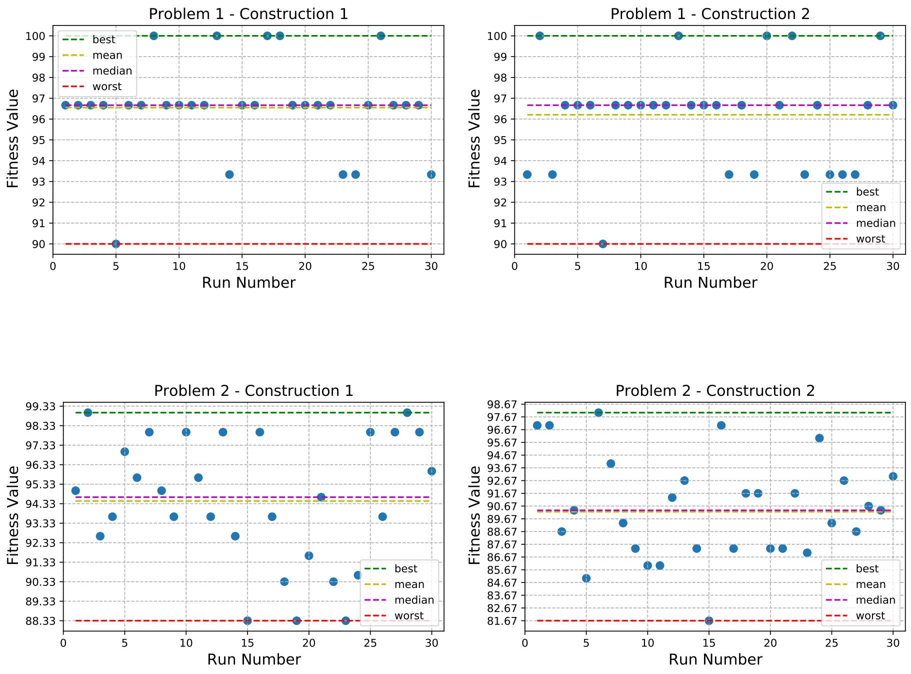

Figure 2: The experimental results showing the best, median, mean, and worst fitness values of 30 independent runs for each case.

As shown in Figure 2, the binary-coded GA can find the global maximum solutions and near-optimal solutions of problem 1 for both constructions. Moreover, the performance of 30 independent runs in terms of fitness value is similar

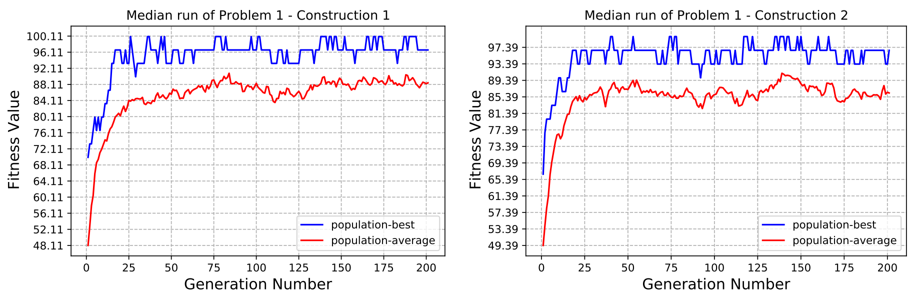

Figure 3: The population-best and population-average of two median runs of each construction for Problem 1.

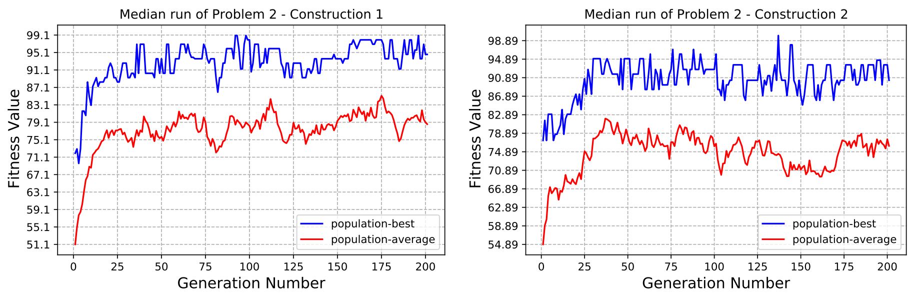

Figure 4: The population-best and population-average of two median runs of each construction for Problem 2.

and relatively high. In contrast, the binary-coded GA's performance in solving problem 2 is highly diverse, leading to unreliable results, and rarely find the optimal solution for both constructions.

The population-best and population-average of two median runs of each construction for problem 1 are similar, as shown in Figure 3. The population-best line has fluctuated, possibly due to the lack of the elite-preserving operator and high probability of mutation operator. Without an elite-preserving operator, the previous-found best solutions are not preserved in subsequent generations. Thus, the population-best solutions can be degraded with generations, and overall performance can be decreased or slightly improved, depending on the given problem. Similarly, the plot of two median runs of each construction for problem 2 shows the fluctuation of fitness values from generation to next generation but with lower fitness values than problem 1 for both population-best and population-average, as shown in Figure 4.

4 Schema Growth Analysis

The schema theorem tells us that the good schemata's proportion increases when they are above the average fitness and have relatively low crossover disruption. Thus, having a sufficient supply of the good building blocks (BBs) required to solve a given problem and the proportion of these good BBs in the population grow is one of the factors in making sure that GAs work well. Figure 5 - 8 show the proportion of 10 three-bit building blocks and their competitors for each case over 30 independent runs. $s(i)$ anD $s'(i)$, where $i = 1, 2, ..., 10$, represent the ith building block and its competitors respectively. For example, the first building block $s(1)$ and its competitor $s'(1)$ for construction 1 are (111 *...*) and (000 *...*). For construction 2 of a problem 1, the first block $s(1)$ and its competitor $s'(1)$ are (1********* 1********* 1*********) and (0********* 0********* 0*********) respectively.

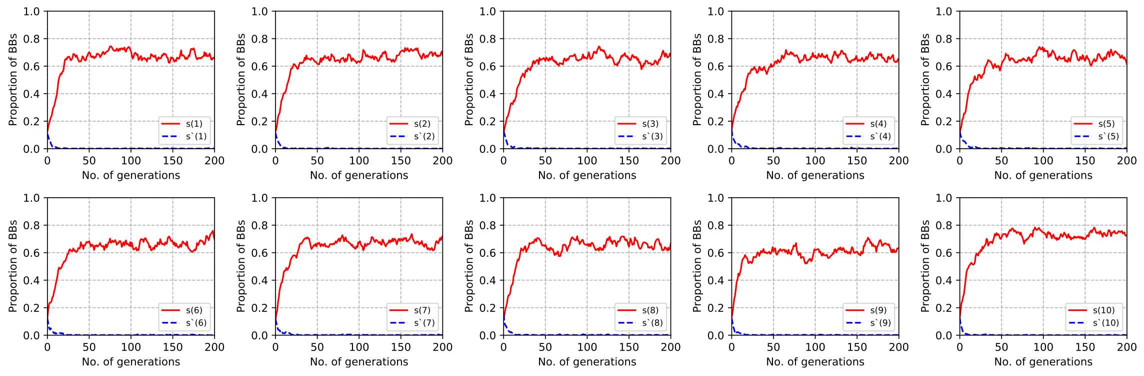

Figure 5: The proportion of 10 three-bit building blocks (BBs) of Problem 1 with Construction 1. The results are averaged over 30 independent runs

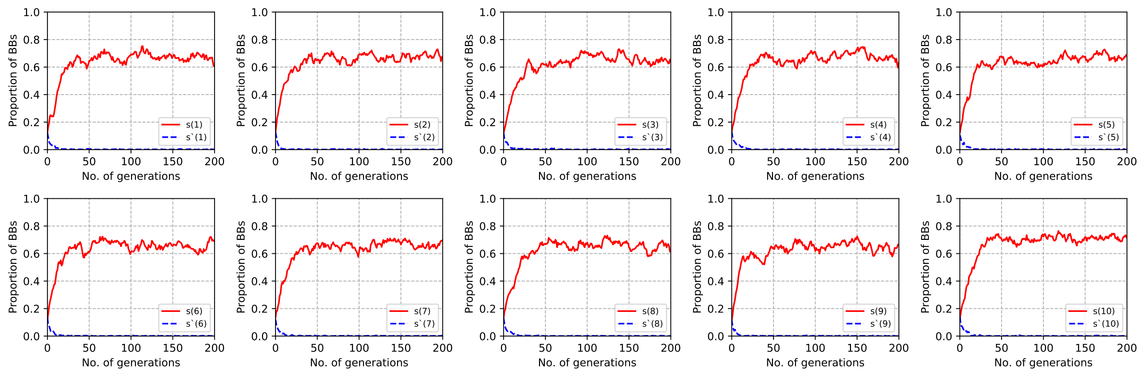

Figure 6: The proportion of 10 three-bit building blocks (BBs) of Problem 1 with Construction 2. The results are averaged over 30 independent runs

Figure 5 and 6 show the successful takeover of the majority of the population by all ten building blocks. Initially, the difference between the proportion of the best BB and its competitor is small. However, due to the given problem's complexity, all ten BBs can completely take over most of the population, leading to the global maximum solution. Interestingly, these good BBs cannot take over the population. This can be explained by the fact that no elite preserving operator is to be used, relatively high crossover disruption, and high p_m.

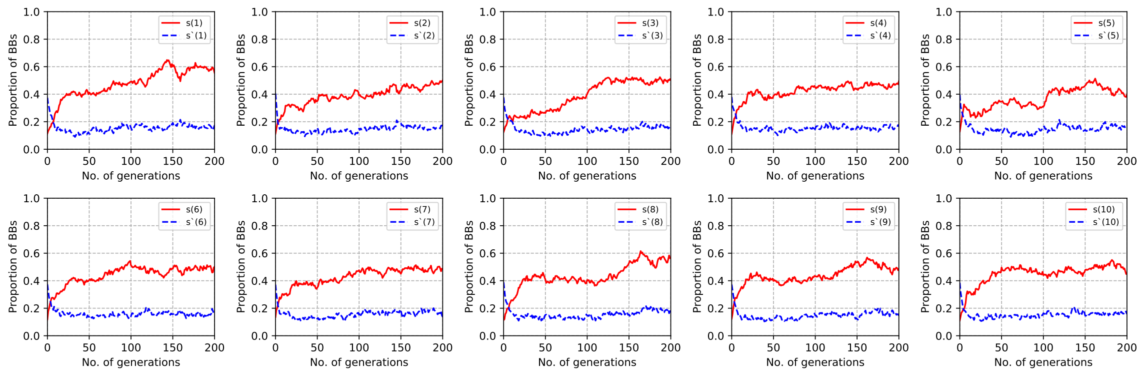

Figure 7: The proportion of 10 three-bit building blocks (BBs) of Problem 2 and Construction 1. The results are averaged over 30 independent runs

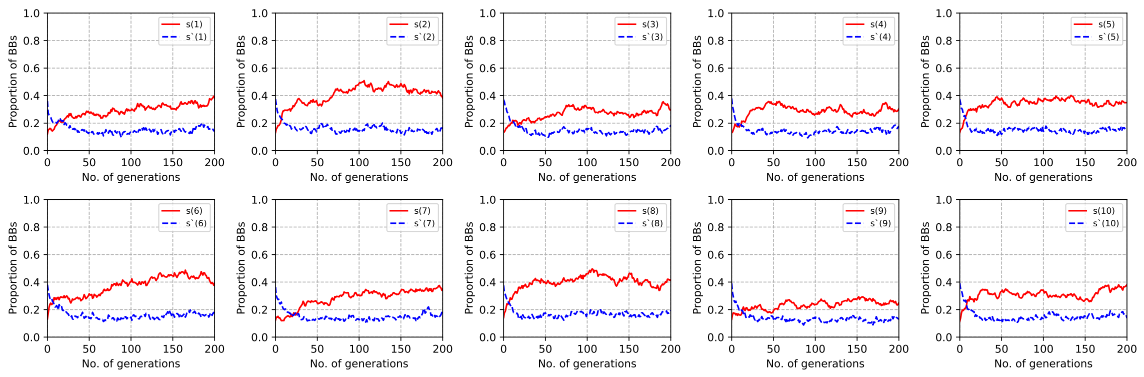
Figure 8: The proportion of 10 three-bit building blocks (BBs) of Problem 2 and Construction 2. The results are averaged over 30 independent runs

Finally, figure 7 and 8 show the proportion of the best BB and their competing BBs of problem 2. For example, the first building block of construction 1 $s(1)$ is (111 *...*) and their competitors $s'(1)$ are (011 *...*), (101 *...*), and (110 *...*). For construction 2, the first building block $s(1)$ is (1*...* 1*...* 1*...*) and their competitors $s'(1)$ are (0*...* 1*...* 1*...*) , (1*...* 0*...* 1*...*) , (1*...* 1*...* 0*...*). The plots shown in 7 and 8 indicate the failure of the best BBs to take over the majority of the population of the complex problem 2. Initially, the competing BBs grow faster, but after some generation, the best BBs can completely take over their competing BBs. The best BBs can beat their competitors in all ith building blocks but stalls at some intermediate proportion value. In all these cases, the proportion of the best BBs stall for many generations and then either it succeeds or fails to find a near-optimal solution.

5 Conclusion

The major findings of this experimental study are listed below:

- The binary-coded GAs with both crossover and mutation operators can find an optimal solution to the simple linear problem (problem 1).
- To solve a complex problem (problem 2) using the binary-coded GAs, the mutation operator has failed in solving the problem. This is because the large probability ($p_m = 1/l$) of mutation operator destroys the good building blocks found in a population. In contrast, the crossover operation is the primary key search operator in solving complex problems. However, the binary-coded GAs still fails to converge to a global maximum. Thus, using a crossover operator with a bigger certain size of the population and more number of allowed generation runs needs to be investigated further.
- The given GA parameters might not be appropriate for solving the complex problem 2 because these parameter settings do not ensure the growth

of the best BBs in the population and do not sustain the growth to takeover the population. Thus, the bigger population size, lower probability of mutation operator p_m, more number of allowed generation runs, and elite-preserving operators may need to be applied to solve the complex problems efficiently.

A Appendices

Main Python Code implementation for this experimental study. Note that the working code is in the zipped file.

```
import numpy as np
import matplotlib.pyplot as plt
from functools import partial
from collections import defaultdict
import random
import os

def perform_tournament_selection(pop, pool_size, k, fitness_func):
    mating_pool = []
    for i in range(pool_size):
        # select k=2 candidates
        idxs = np.random.randint(np.array(pop).shape[0], size=k)
        candidates = [pop[i] for i in idxs]
        top_candidate = np.argmax([fitness_func(s) for s in candidates])
        mating_pool.append(candidates[top_candidate])

    return mating_pool


def perform_singlepoint_crossover(pop_size, mating_pool, pc):
    offsprings = []
    for i in range(pop_size):
        # this is hard-code as only two parents are selected
        idxs = np.random.randint(len(mating_pool), size=2)
        parents = [mating_pool[i].tolist() for i in idxs]
        if np.random.uniform(0, 1, 1)[0] < pc:
            crossover_site = np.random.randint(len(mating_pool), size=1)[0]
            offspring1 = parents[0][:crossover_site] + parents[1][crossover_site:]
            offspring2 = parents[1][:crossover_site] + parents[0][crossover_site:]
            offsprings.append(offspring1)
            offsprings.append(offspring2)
        else:
            offsprings.append(parents[0])
            offsprings.append(parents[1])

    return offsprings

def perform_bitwise_mutation(offspring, pm):
    random_list = np.random.uniform(0, 1, len(offspring))
    offspring = np.array(offspring)

    def mutate(a, b):
        if a < pm:
            return 0 if b == 1 else 1
        else:
            return b

    mutation_func = np.vectorize(mutate)
    new_offspring = mutation_func(random_list, offspring)

    return new_offspring


# Define Objective Functions
def get_fitnesses(pop, fitness_func):
    # evaluate fitness of P
    fitnesses = [fitness_func(s) for s in pop]

    return [(p[0], p[1]) for p in zip(pop, fitnesses)]

def objective_function(s, problem_type, construction):
    if (type(s).__module__ == np.__name__):
        s = s.tolist()

    def problem1(s1):
        # s1 is [0, 1, 1]
        return s1.count(1)/3

    def problem2(s1):
        if s1.count(1) < 3:
            return 0.9 - s1.count(1)/3
        else:
            return 1

    problem = problem1
    if problem_type == 2:
        problem = problem2


    if construction == 1:
        subfunc1_res = [problem(s[3*i:3*i+3]) for i in range(len(s)//3)]
        fitness_val = 10*np.sum(subfunc1_res)
        return fitness_val
    else:
        subfunc2_res = [problem([s[i], s[i+10], s[i+20]]) for i in range(len(s)//3)]
        fitness_val = 10*np.sum(subfunc2_res)
        return fitness_val


# Define experiment settings and Hyperparameters
# Problem information
#====================================
# there are two problems 1 and 2
problem = 2
# there are two constructions 1 and 2
construction = 1
FITNESS_FUNC = partial(objective_function, problem_type=problem, construction=construction)


# Algorithms settings
#====================================
POP_SIZE = 60
T = 200
RUN = 30
BIT_LENGTH = 30
Pc = 0.9
Pm = 1/30
K = 2 # tournament size

def run_ga():
    for run in range(RUN):
        pop = np.random.randint(2, size=(POP_SIZE, BIT_LENGTH))
        pop_gen = [pop]
        for i in range(T):
            mating_pool = perform_tournament_selection(pop, pool_size=POP_SIZE, k=K, fitness_func=FITNESS_FUNC)

            offsprings = perform_singlepoint_crossover(pop_size=POP_SIZE//2, mating_pool=mating_pool, pc=Pc)
            mutated_offsprings = [perform_bitwise_mutation(ofs, Pm) for ofs in offsprings]
            pop_gen.append(np.array(mutated_offsprings))
            pop = mutated_offsprings

run_ga()
```