Høgskolen i Østfold

Machine Learning Project 2

Sanyam Jain

Aarati Shrestha

02.12.2022

Masters in applied computer science

Faculty of Computer Science

Østfold University Collage

You can check the working code repository here - Drive Folder

Summary: We performed several tests and experiments for the following different:

1. **Mutation Rate**: Mutation in a genetic algorithm is used to introduce genetic diversity from one population generation to another. The goal of mutation is to prevent the algorithm from prematurely converging to a suboptimal solution and to explore new areas.
2. **Elite Chromosome Individuals Rate**: The elite group is a set of chromosomes that directly go to the next generation without undergoing any changes. It accelerates the algorithm convergence
3. **Iterations**: As mentioned earlier 30 iterations we need to finish.
4. **Population Size:** Population size is one of the important parameters of GA. It is a collection of chromosomes where each chromosome represents a potential solution to the problem. It has been observed that a large population increases the likelihood of finding the optimal solution for a given problem. A chromosome is formed by a set of parameters known as genes. The chromosome is frequently represented as a binary string, but several alternative data formats are also used.
5. **Selection / Reproduction Criteria:**
    a. Tournament Selection: Tournament selection is one of the selection mechanisms in GA to select chromosomes from the population to perform crossover. It involves running several "tournaments" among a few randomly selected chromosomes. The tournament winner, the one with the highest fitness value, is selected for crossover. The tournament selection method may be described in pseudo-code:

    ref: https://en.wikipedia.org/wiki/Tournament_selection

    - choose k (the tournament size) individuals from the population at random
    - choose the best individual from the tournament with probability p

- choose the second-best individual with probability p*(1-p)
- choose the third best individual with probability p*((1-p)^2) and so on

b. **Roulette Wheel Selection:** Roulette wheel selection is one of the selection strategies used in GA to identify the probable crossover chromosome. It is largely inspired by the roulette wheel in a casino. Each of the potential chromosomes is given a section in the wheel based on its fitness value. A random number is selected, much like in a casino roulette wheel. Chromosomes with a high level of fitness take up more space in the wheel, increasing their probability of being selected for crossover. Pseudocode for roulette selection wheel:
    - Add up the fitness values for each possible chromosome (SoF).
    - Pick a number at random (R) between 0 and SoF.
    - Beginning with the first chromosome in the population, add up the fitness value of the chromosomes (P) until the P >= R.
    - The chromosome chosen for mating is the one for which P exceeds R.

6. **Fitness Function:** How effective a given solution is at resolving the goal problem is assessed using the fitness function. Whether a given chromosome is near the goal problem depends on its fitness level.

Experimentation and Results: A total of 16 different python files and python notebooks have been produced to tweak the parameters mentioned above to get results. All tasks have been performed as mentioned in the project task list. Please refer to the following directory structure:

Given Tasks and respective file names to check

- **Task1:** Given a randomly generated bit string of length 100, create a genetic algorithm that through mutation tries to obtain a string containing only 1s, i.e., 111111...111111.
  - ➢ **From v1 to v16 (all versions)**
- **Task2:** Implement a fitness function that rewards strings containing more 1s, e.g., a counter.
  - ➢ **From v1 to v16 (all versions)**
- **Task3:** Decide initial values for the following parameters and implementation choices for your algorithm: population size, mutation rate, selection/reproduction criteria, number of generations.
  - ➢ **From v1 to v16 (all versions)**
- **Task4:** Execute your algorithm for at least 30 independent runs and collect statistics. Generate fitness plots over the generations for your statistics (best, avg, deviation)
  - ➢ **v11 to v14 (For 30 iterations)**
- **Task5:** Systematically change your parameters (population size, mutation rate) and/or choices for selection/reproduction criteria. Does your algorithm still converge to the global optima? Does it need more or less generations to converge?
  - ➢ **v14 to v16**
- **Task6:** Try to change the length of the randomly generated string to 10 bits and 1000 bits (or more). What do you observe?
  - ➢ **v1 to v10. It is super difficult to run the algorithm for more than 50 bit length. We tried to run it for different bit length sizes, such as 10, 20, 40, 80, 100, 200, 400 and 800. However, the program always stuck to the local suboptimal, that is for example, 100 bit length it stuck at 99 bit convergence. Please check v15 and v16 folders to check results.**
- Discuss and explain your results.
  - ➢ Please follow the report itself.

We will use version numbers to describe the description of files according to the referenced image above: (Please use the respective file name to verify the algorithm and programming code)

1. **v0:** contains basic operations (Brainless Appraoches):
    a. Bit Flip Mutations
    b. Random Resetting Mutations
    c. Swap Mutations
    d. Scramble Mutations
2. **v1:** Tried one iteration of the 100-bit One-Max algorithm with the following setting:
    a. population_size = 10
    b. mutation_rate = 0.25
    c. number_of_generation = 100
    d. numb_of_elite_chromosomes = 1
    e. tournament_selection_size = 4

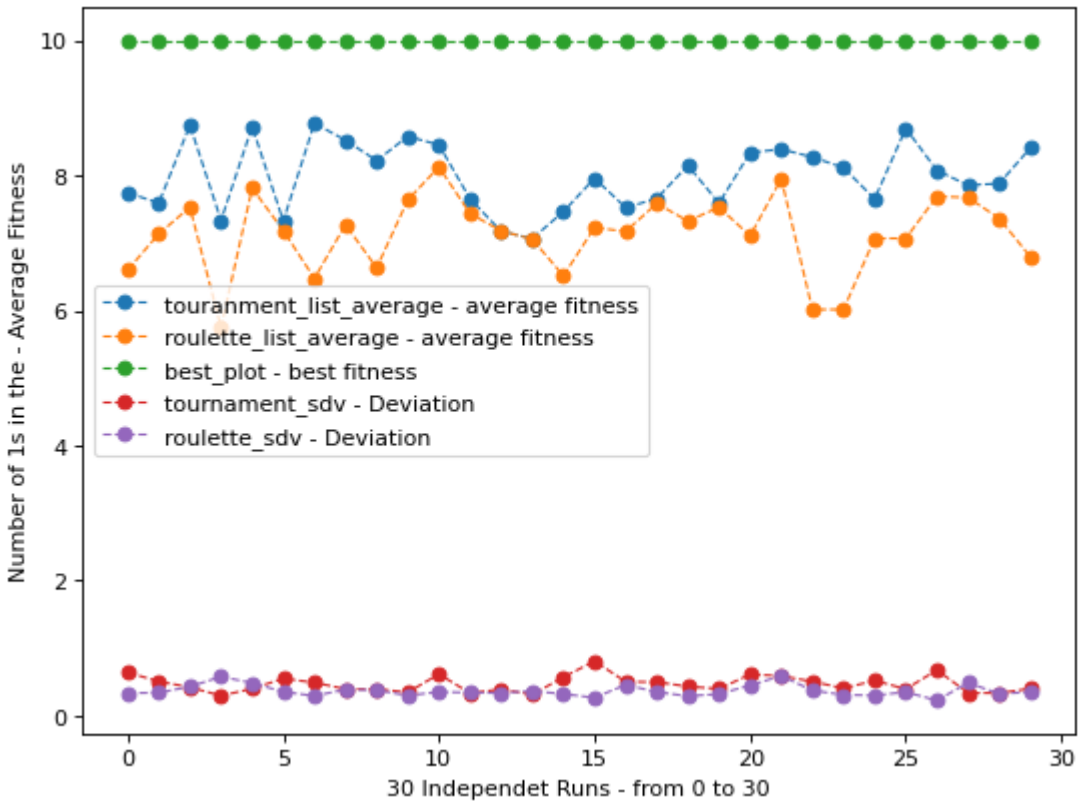
    Last generation output for v1:

    Generation #: 284038 | fittest chromosome fitness: 100 TARGET CHROMOSOMES :

    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] ACTUAL Chromosome :

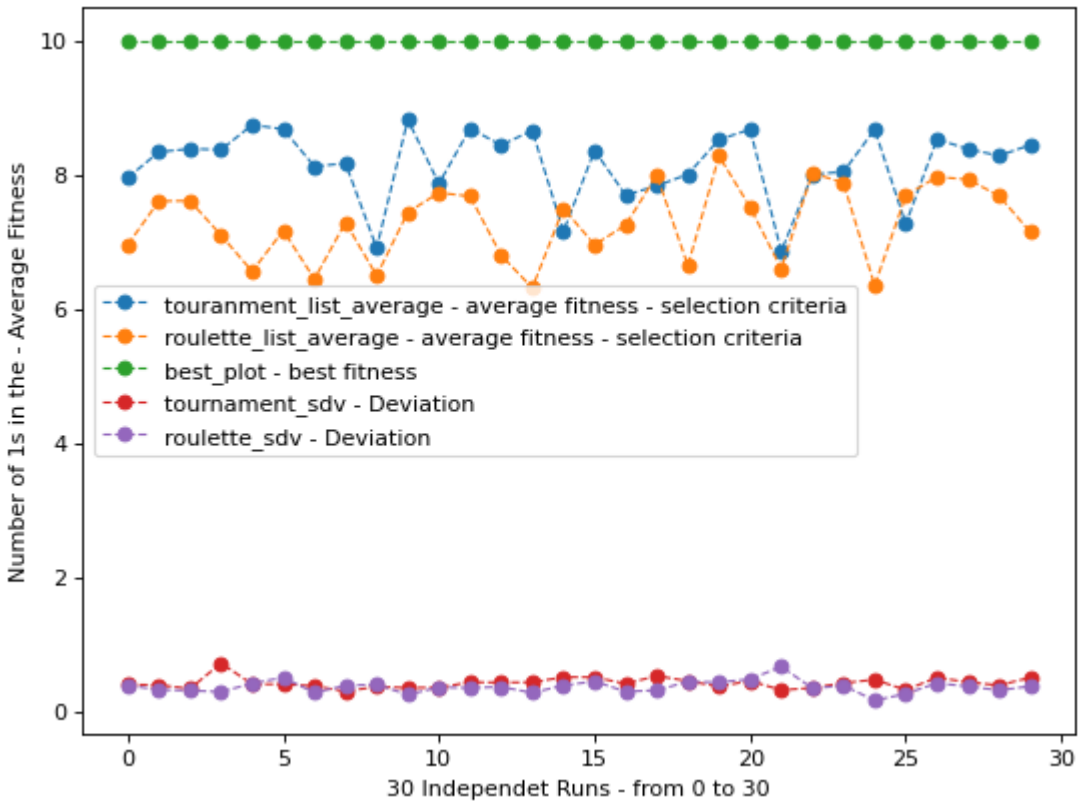
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

3. **v2 and v3: Tasks 1, 2 and 3 are covered in the version 2 of the python notebooks and following. (Checked for 10 bits) Where the output:**

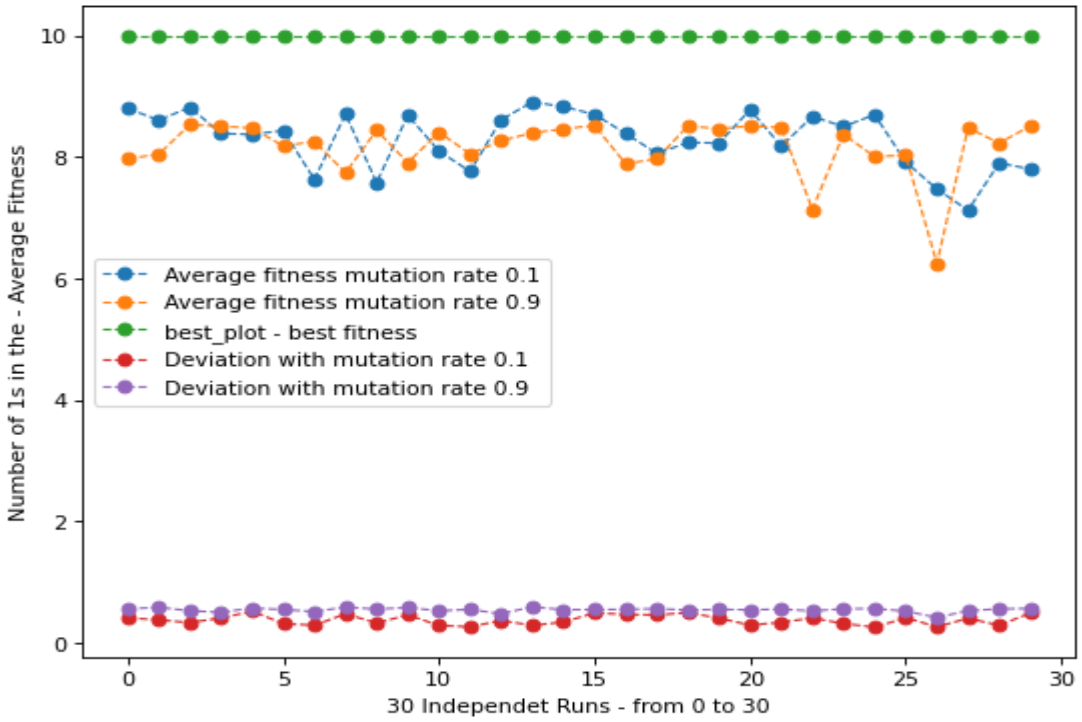


Explanation: Best fitness is of course 10 for 10 bits for all iterations. The green curve shows best fitness, orange and blue curves are average fitnes for all 30 iterations with different selection or reproduction criteria respectively. Finally, red and magenta curves, shows deviation of both the curves respectively. But this is for 10 bits.

4. **Task 4 is implemented in v4 notebook where 30 independent iterations are done for 10 bit length string with the same parameters, Output:**



5. **v5 notebook has changed mutation rate (0.1 & 0.9), output:**



Observation: With the mutation rate 0.1 we see that it starts to stuck in local optimum (for 100 bits it stucked to 99th bit), as genetic diversity decreases, however, as mutation rate becomes 0.9 we see that it starts to produce unwanted mutations and increases genetic diversity (chaos). That is why mutation rate of 0.5 worked best.

In simpler words: Mutation for each solution, we go through all of its chromosomes and with a certain probability we decide if the current chromosome is going to be changed or not how exactly do we change the chromosomes, for example if the chromosome is a bit we flip it from 1 to 0 and from 0 to 1. If the solution is a string then we change the character with another random character and for the traveling salesperson as a problem, for example, we swap the order of two cities between themselves and that's because you can't just change one city as you will see in the coding examples we submit.

A final word about the parameters to play around with them and see which combination gives the best results:

- **If the mutation rate is too high then that produces chaos and meaningless results.**
- **If the mutation rate is too low then you might get stuck in a local minimum or maximum like in the hill climb. You need some randomness to get out of it.**
- **Population size, if it's too high then the program will execute slowly if it's too low then you will get worse quality results.**

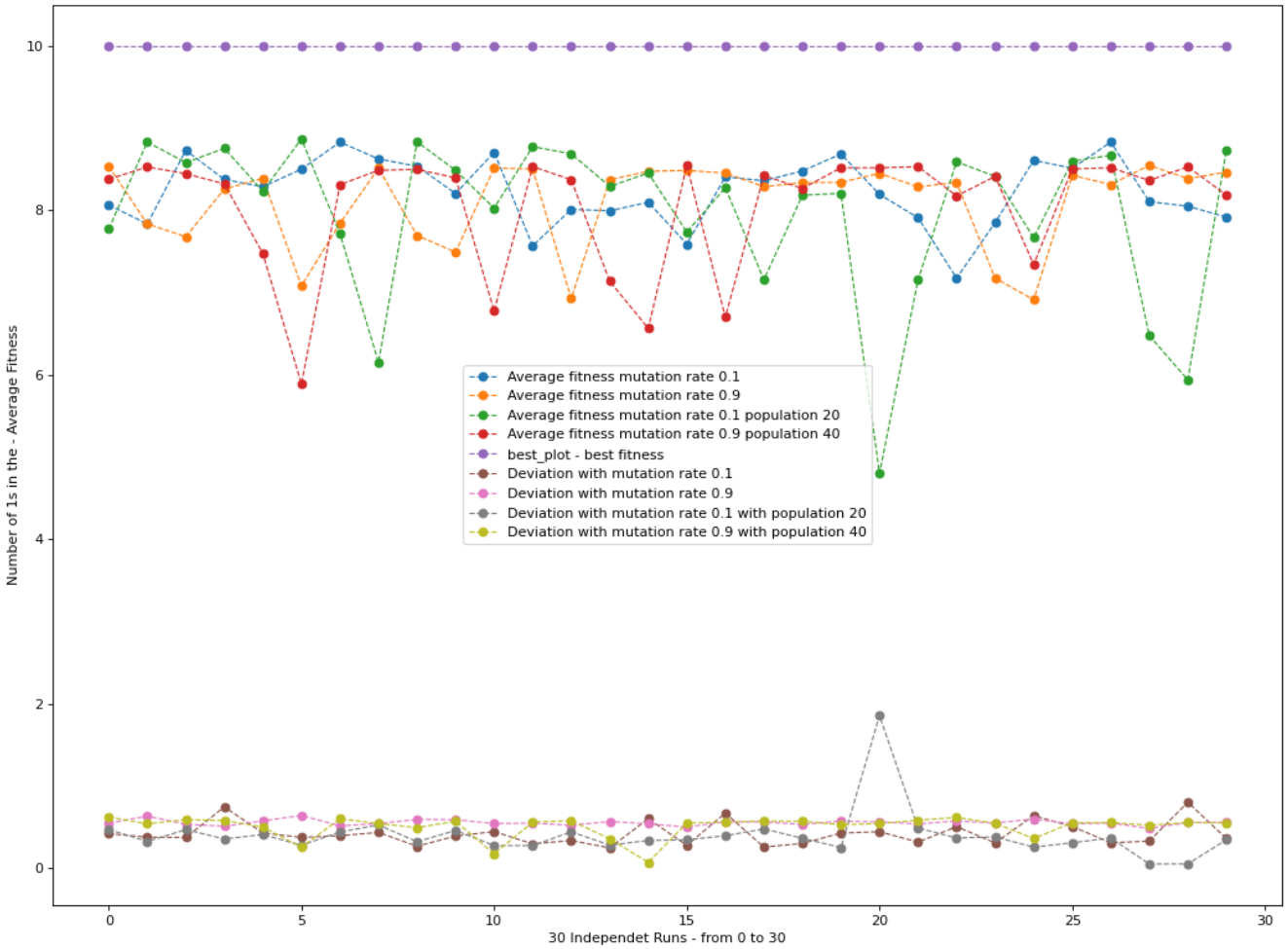
6. **<u>v6 (Imp notebook)</u> has some more results with changning mutation rates and population rates for 10 bit length string. Observations: (10 bit length)**

```
Number of Generations for 30 runs as array for mutation rate 0.1:
[19, 27, 43, 9, 37, 25, 55, 19, 66, 30, 35, 26, 22, 79, 14, 56, 8,
59, 71, 31, 27, 52, 10, 10, 16, 11, 78, 19, 10, 18]
```

```
Number of Generations for 30 runs as array for mutation rate 0.9:
[434, 37, 81, 72, 40, 45, 146, 144, 63, 73, 384, 696, 20, 112,
149, 123, 196, 114, 177, 250, 209, 192, 175, 23, 10, 64, 168, 167,
561, 635]
```

```
Number of Generations for 30 runs as array for mutation rate 0.1
with population 20:  [6, 43, 12, 25, 17, 66, 12, 2, 61, 18, 77,
96, 18, 74, 43, 20, 12, 10, 27, 102, 1, 4, 40, 21, 8, 68, 24, 2,
2, 45]
```

```
Number of Generations for 30 runs as array for mutation rate 0.9
with population 40:  [83, 232, 99, 143, 24, 2, 24, 118, 84, 60, 2,
687, 25, 2, 2, 256, 2, 49, 164, 203, 283, 434, 110, 40, 5, 259,
160, 28, 594, 11]
```



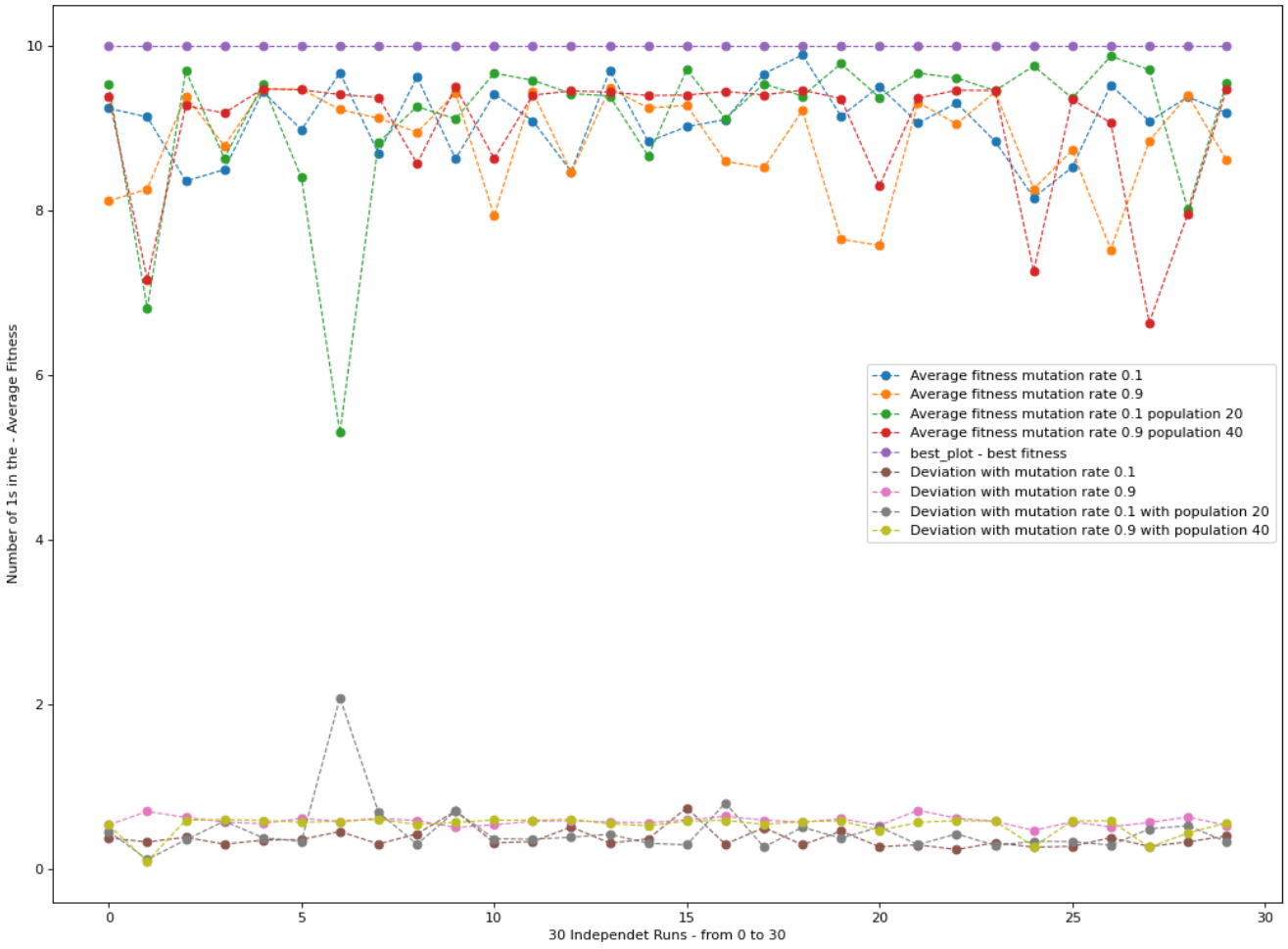
The facts and hypothesis described in point 5 are true for this case too (as final words)!

7. v7 is just another independent run of version 6 (another 30 runs) with output: (10 bit length)

```
Number of Generations for 30 runs as array for mutation rate 0.1:
[36, 54, 34, 73, 55, 53, 46, 74, 50, 9, 46, 31, 18, 93, 28, 7, 44,
36, 101, 46, 73, 62, 115, 63, 8, 39, 40, 78, 57, 75]

Number of Generations for 30 runs as array for mutation rate 0.9:
[267, 39, 216, 118, 288, 333, 285, 66, 169, 165, 28, 252, 92, 241,
105, 492, 126, 176, 55, 20, 5, 32, 262, 133, 12, 146, 16, 121,
294, 88]
```

Number of Generations for 30 runs as array for mutation rate 0.1 with population 20: [19, 2, 61, 3, 47, 10, 1, 5, 64, 6, 40, 62, 52, 17, 69, 94, 7, 110, 17, 35, 11, 110, 21, 69, 91, 39, 66, 30, 3, 42]

Number of Generations for 30 runs as array for mutation rate 0.9 with population 40: [41, 2, 137, 100, 1707, 326, 914, 40, 73, 606, 170, 500, 221, 96, 36, 370, 97, 52, 155, 44, 13, 47, 123, 194, 2, 248, 57, 2, 3, 310]

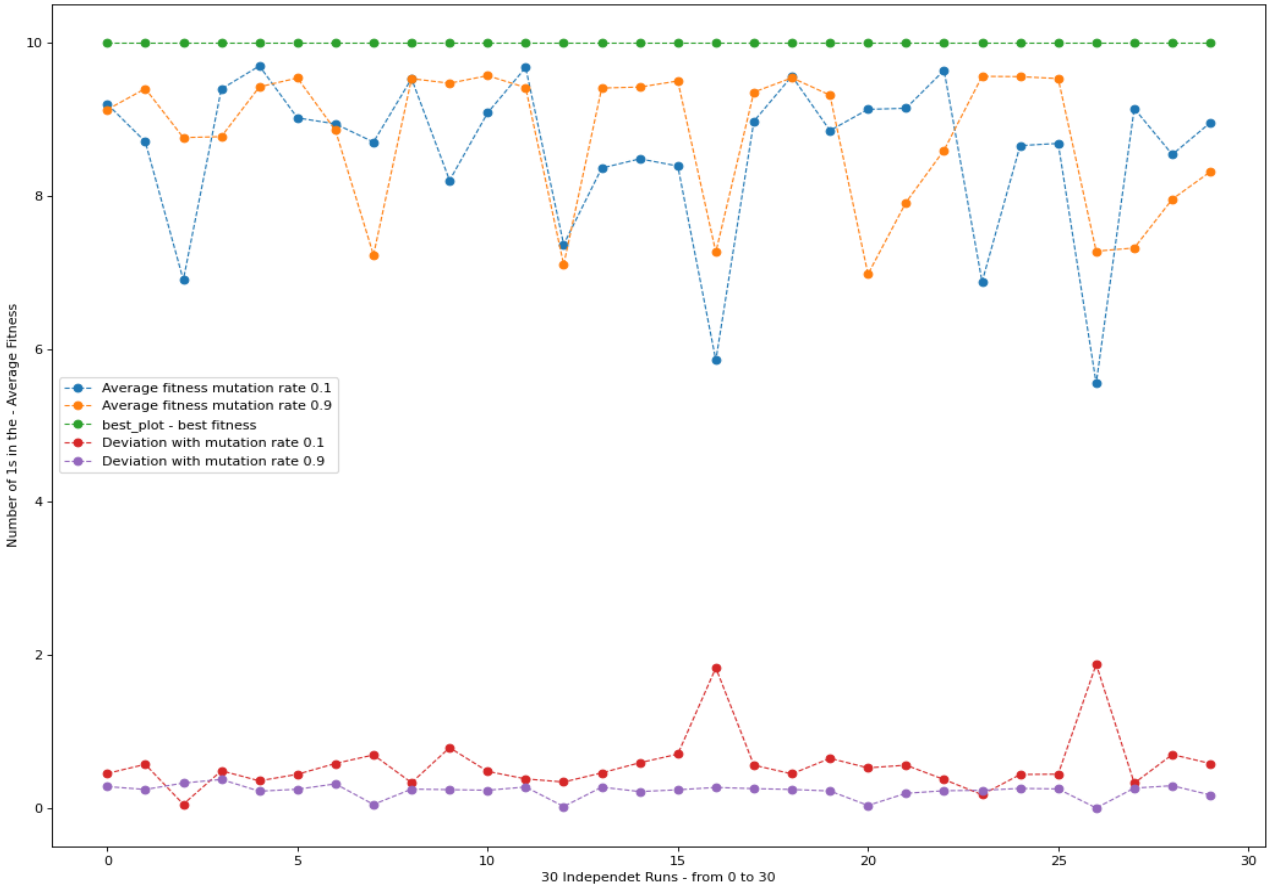


8. v8 has following parameters:

```
population_size = 40

mutation_rate_val = 0.25

target_individual = [1,1,1,1,1,1,1,1,1,1,1]

if(int(0.1 * population_size) ) >= 1:

  elite_individuals = int(0.1 * population_size)

else:
```

```
  elite_individuals = 1


if(int(0.5 * population_size) ) >= 1:
  tournament_selection_size = int(0.5 * population_size)
else:
  tournament_selection_size = 1


if(int(0.4 * len(target_individual)) ) >= 1:
  crossover_point =  int(0.4 * len(target_individual))
else:
  crossover_point = 1
```

The Output:



9. versions 9 to 10 are just same experiments with different parameters, please open and check both the notebooks, and for 10 bits lengths.

10. v11 we increased 100 bits but it did not converge, even 1 iteration (out of all 30 iterations). Even after changing the parameters:

- **a. Mutation rate: 0.1 and 0.9 did not work**
- **b. Population size = 50%**
- **c. Crossover point = 0.4**

The last iteration has following information:

GENERATION Number : 59296

BEST Individual FITNESS : 57

TARGET Individuals : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

BEST Individuals : [1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1] Elite population : 9

len of pop indi : 10

ITERATION NUMBER 0

---

11. Now versions 15th and 16th has some interesting convergence trials because of the handcrafted and fine tuned parameters. Version 15th has 40 bit length and version 16th has 50 bit length. However the major parameter that helped to speed up the processing as mutation rate as 0.5 mutation rate. We tried to log the outputs too. Just the increase of 10 bits in bit length, the size of logs increased from 63.4MB for bit length 40 to 445MB for bit length 50. And the outputs are attached as follows:

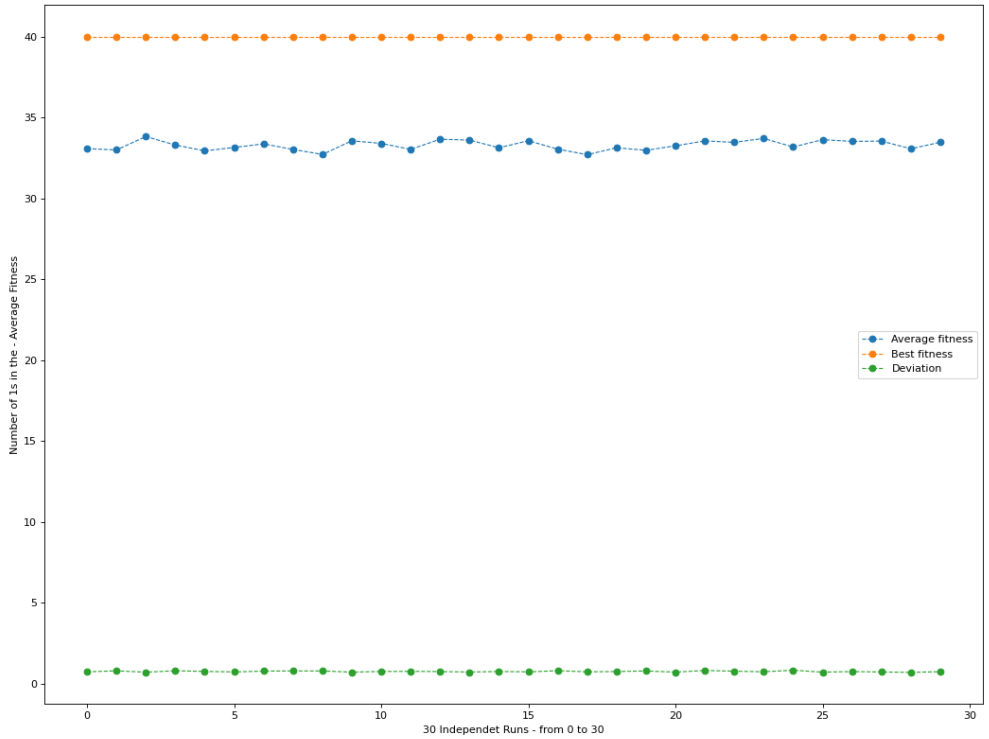


Diagram Caption:

genetic_algorithm_v15_running_for_40bits_0.50mutationrate_10populationsize_0.5eliteindividuals_0.5touranmentselection

refer v15 notebook

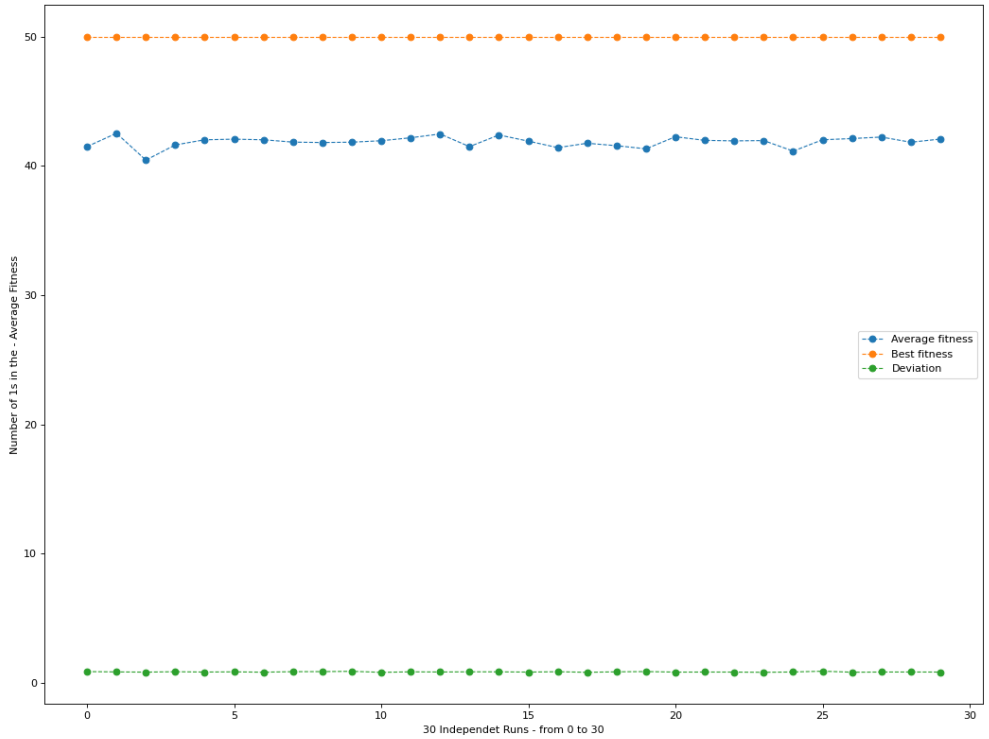


Diagram Caption:

genetic_algorithm_v16_running_for_50bits_0.50mutationrate_10populationsize_0.5eliteindividuals_0.5touranmentselection

refer v16 notebook