Genetic Algorithm for Set Covering Problem

1. Introduction

Set Covering Problem (SCP) is a classical optimization problem where the goal is to select the minimum number of subsets that cover all elements of a universal set. This problem appears in various fields such as network design, scheduling, and resource allocation. Since SCP is computationally difficult to solve exactly for large instances, approximate solutions are typically sought.

Genetic Algorithms are based on the principles of **natural evolution**, where solutions evolve over time. In GA, each solution is encoded as a binary string, where each bit represents the inclusion or exclusion of a subset. The fitness of a solution is determined by how well it covers the universal set and minimizes the number of subsets used.

2. Objective

The primary objective of this experiment is to develop a Genetic Algorithm for solving the SCP by evolving a population of binary strings (each representing a candidate solution) to maximize the coverage of a universal set while minimizing the number of subsets used.

3. Methodology

3.1 Mutation

Used **mutation rate of 1-2 bits per 100 bits**
The function mutate() takes a binary string and a mutation rate as inputs and flips bits probabilistically based on the mutation rate. The mutation is a crucial operator in GA, which introduces diversity into the population. **If the bit is '1', it may mutate based on the mutation rate; for '0', the mutation rate is reduced to balance the process** (giving better output to flip 1 to 0 with more probability than to flip 0 to 1).

3.2 Binary String Generation

The function generate_random_binary_strings() generates a population of binary strings. Each string is of length n (equal to the number of subsets), and each position in the string is either '0' (subset not selected) or '1' (subset selected).
Used **a ratio of 10 ‘0s’ for each ‘1’** which helped taking answer closer to optimal

3.3 Fitness Function

The fitness_score() function evaluates each binary string's fitness. For each '1' in the string, the corresponding subset is included in the coverage set. The fitness score is computed based on the number of elements covered. **If the universal set is fully covered, the solution is rewarded with a bonus proportional to the number of subsets not used** (penalizing overly large solutions). Otherwise, the fitness is simply the number of unique elements covered.

3.4 Crossover and Selection

Crossover value = random again leading to optimal or similar answers
Crossover is performed by selecting two binary strings based on their fitness scores using random.choices() with weighted probabilities. A random crossover point is chosen, and two new offspring strings are created by swapping sections of the parent strings. After crossover, mutation is applied to introduce additional diversity.

3.5 Algorithm Flow

The algorithm proceeds in generations. Initially, a population of 50 binary strings is generated. For each generation, fitness scores are calculated, and **the top 10 solutions are carried over to the next generation (Elitism)**. The remaining solutions are generated using crossover and mutation. This process is repeated for 50 generations.

3.6 Performance Metrics

The mean fitness and standard deviation of fitness scores are recorded at each generation to track the performance of the population over time. Additionally, the number of subsets used in the best solution at the end of each generation is recorded.

Addition of mutation improved performance by 25% on average
Addition of elitism improved performance further by 15% on average

4. Results

4.1 Fitness and Subset Usage

For each experiment (run for 20 iterations), the best fitness value at each generation is recorded. The fitness of the best solution is printed, along with the corresponding binary string and the number of subsets used. This binary string represents the solution for the SCP.

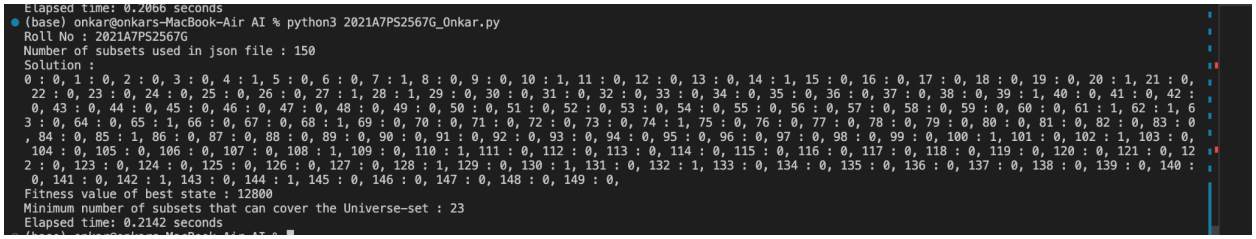
4.2 Mean and Standard Deviation

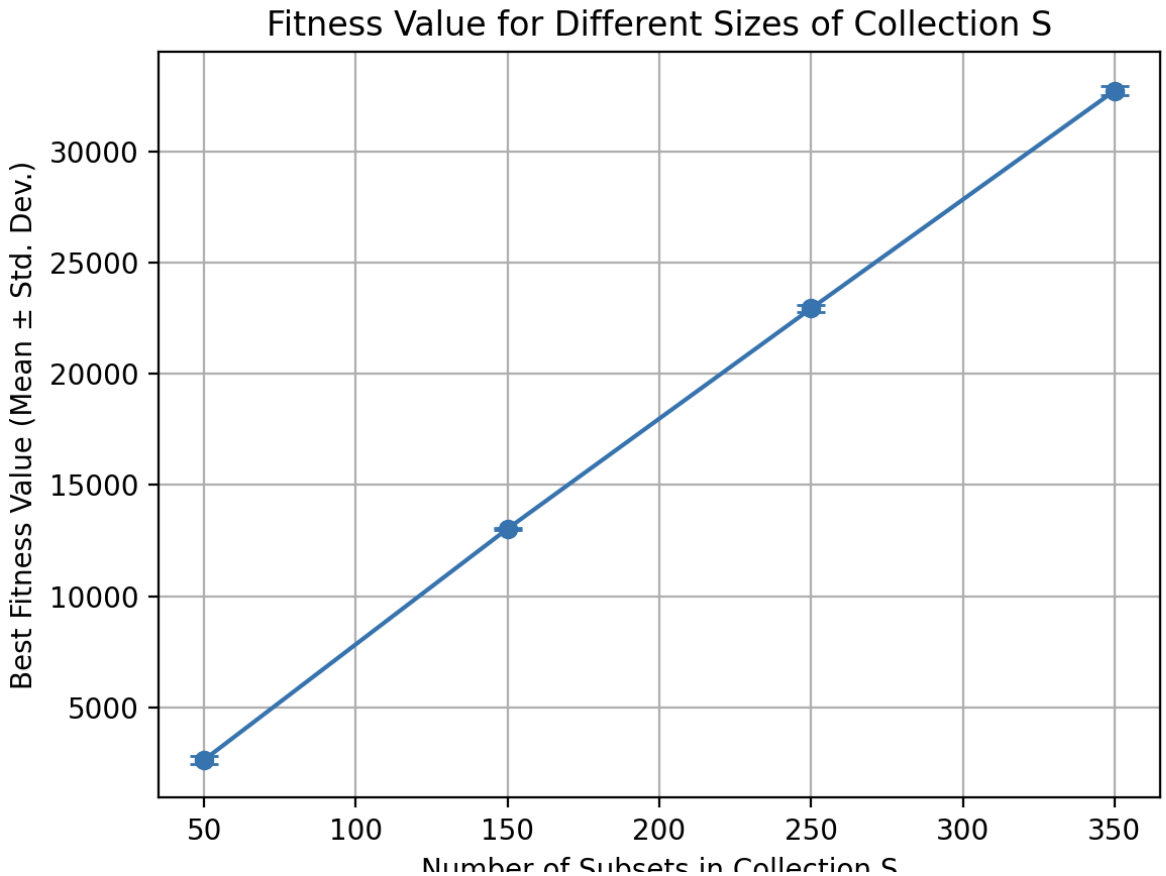
After 50 generations, the mean and standard deviation of the fitness values are calculated and plotted. This allows the assessment of how the population evolves over time and whether it converges to an optimal or near-optimal solution.

5. Conclusion

The Genetic Algorithm demonstrated an effective search over the solution space for the SCP. By balancing coverage of the universal set and minimizing the number of subsets used, the algorithm could evolve populations that produce near-optimal solutions. The mean fitness steadily increased across generations, while the standard deviation decreased, indicating convergence toward better solutions.

Output (Sample json file) :

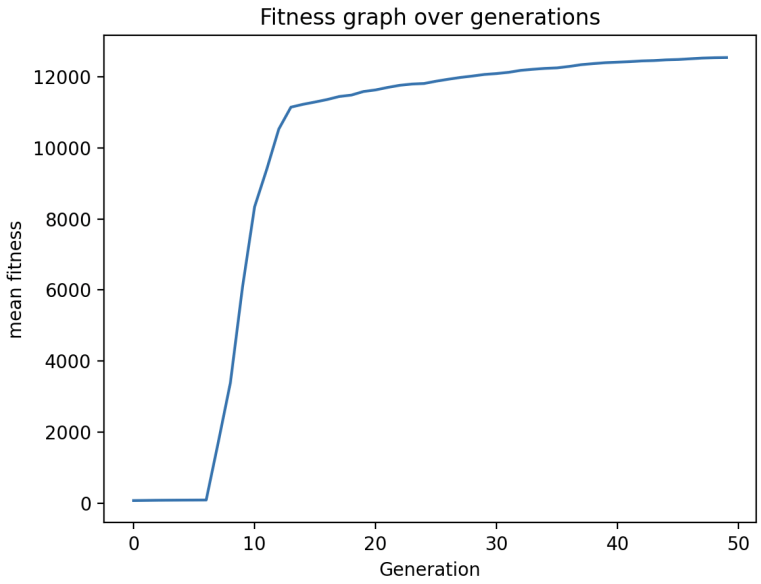




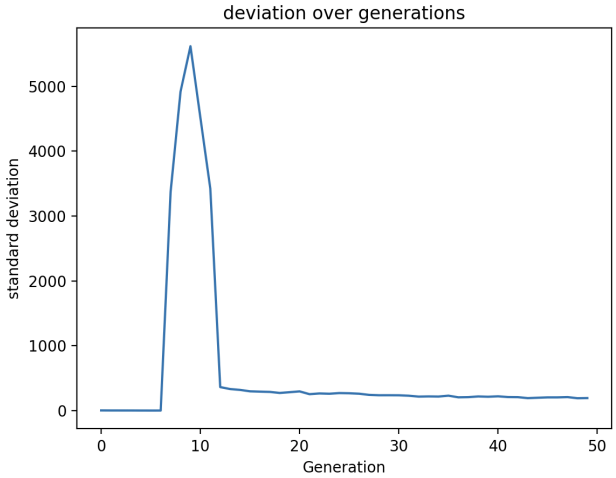
the genetic algorithm runs for each subset size (50, 150, 250, 350) over 10 randomly generated SCPs, and we collect the best fitness values from each run

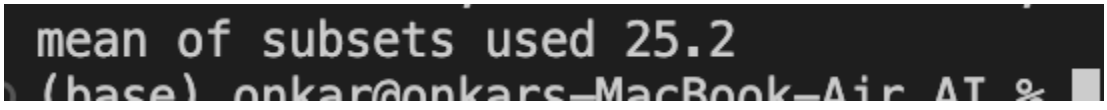
Computing the mean and standard deviation of the best fitness values for each subset size and plot them with error bars using `plt.errorbar`.

Results for the given json test case of population size 150 over 20 trials

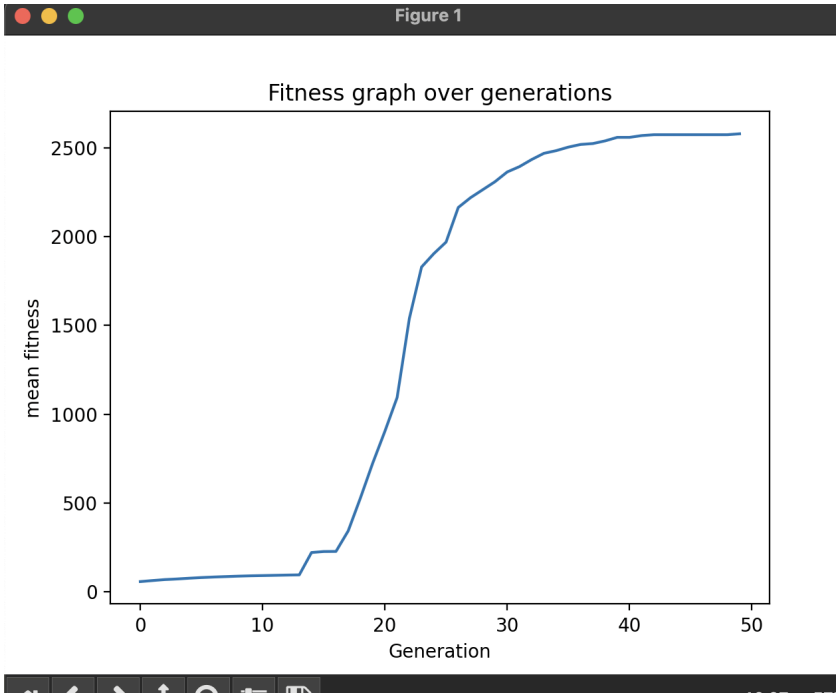


* Sudden increase in fitness is due to fitness function giving more priority to universal fitness

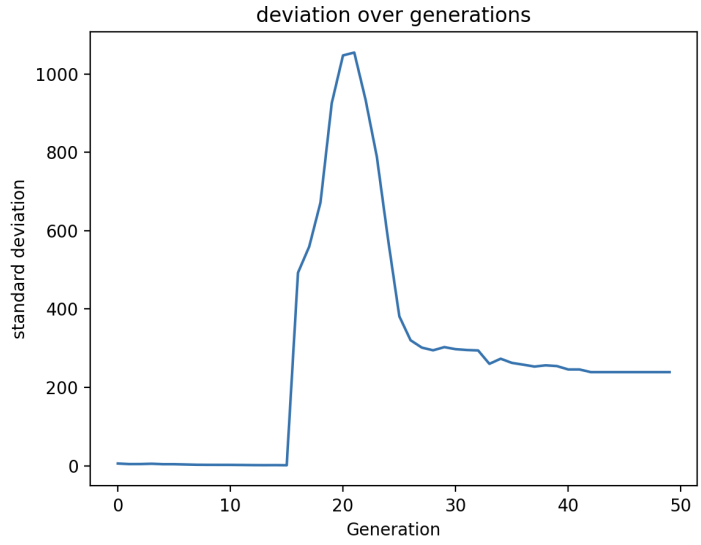




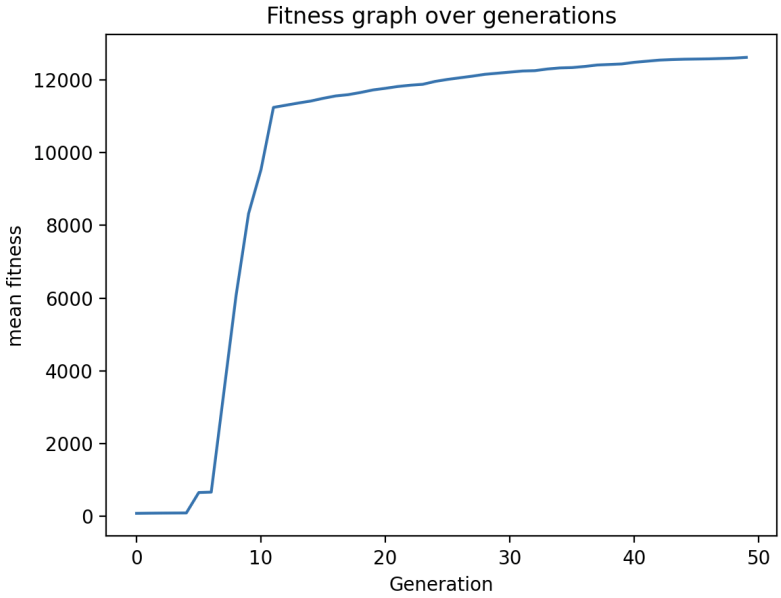
Results for the Randomly generated test case with set count 50

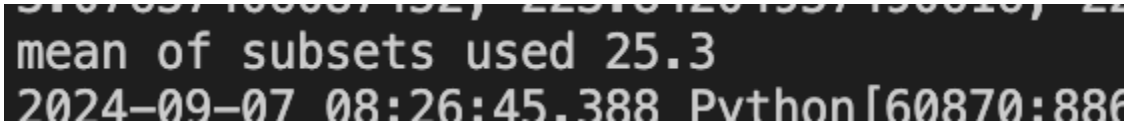


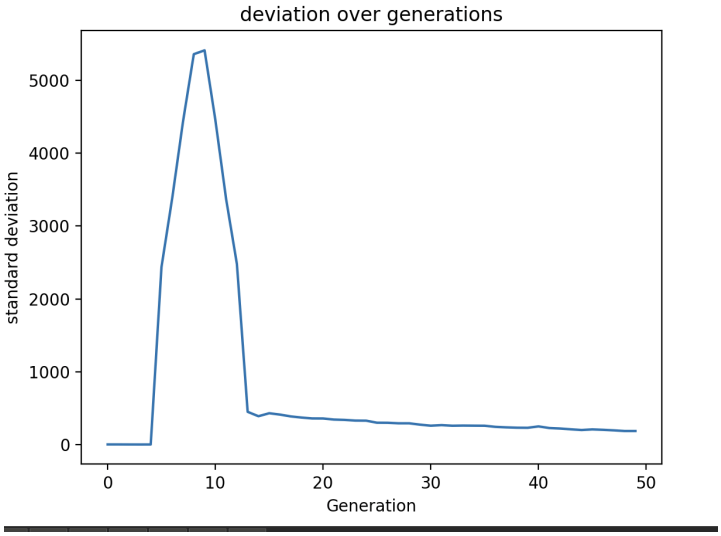

```
999120975515 4, 191.5999120975515 4, 191.5999120975515 4, 191.599
mean of subsets used 25.2
```



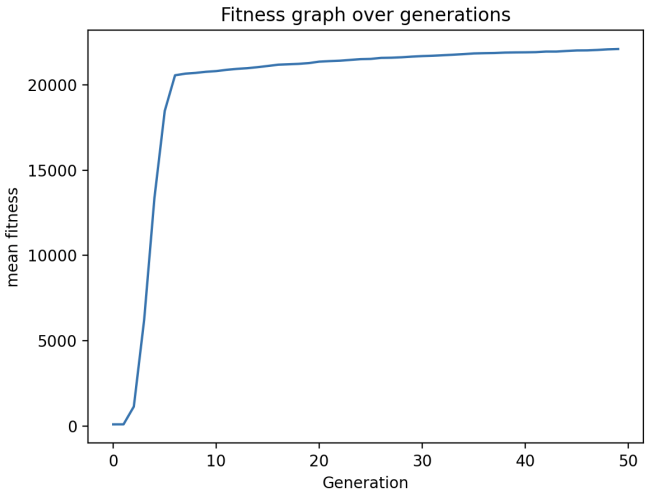
Results for the Randomly generated test case with set count 150



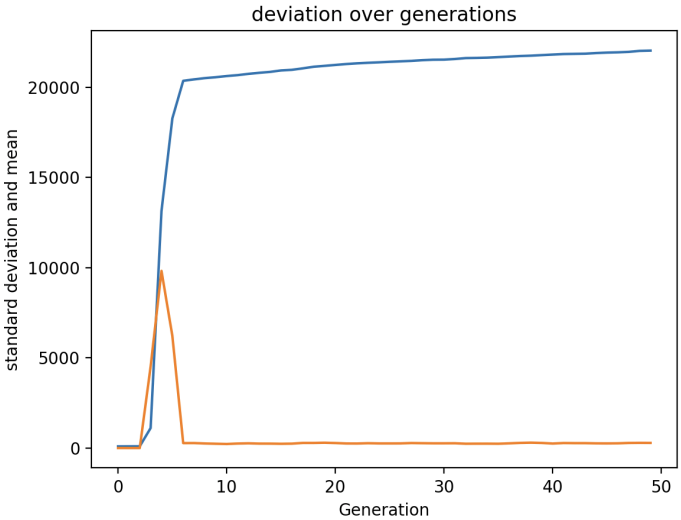




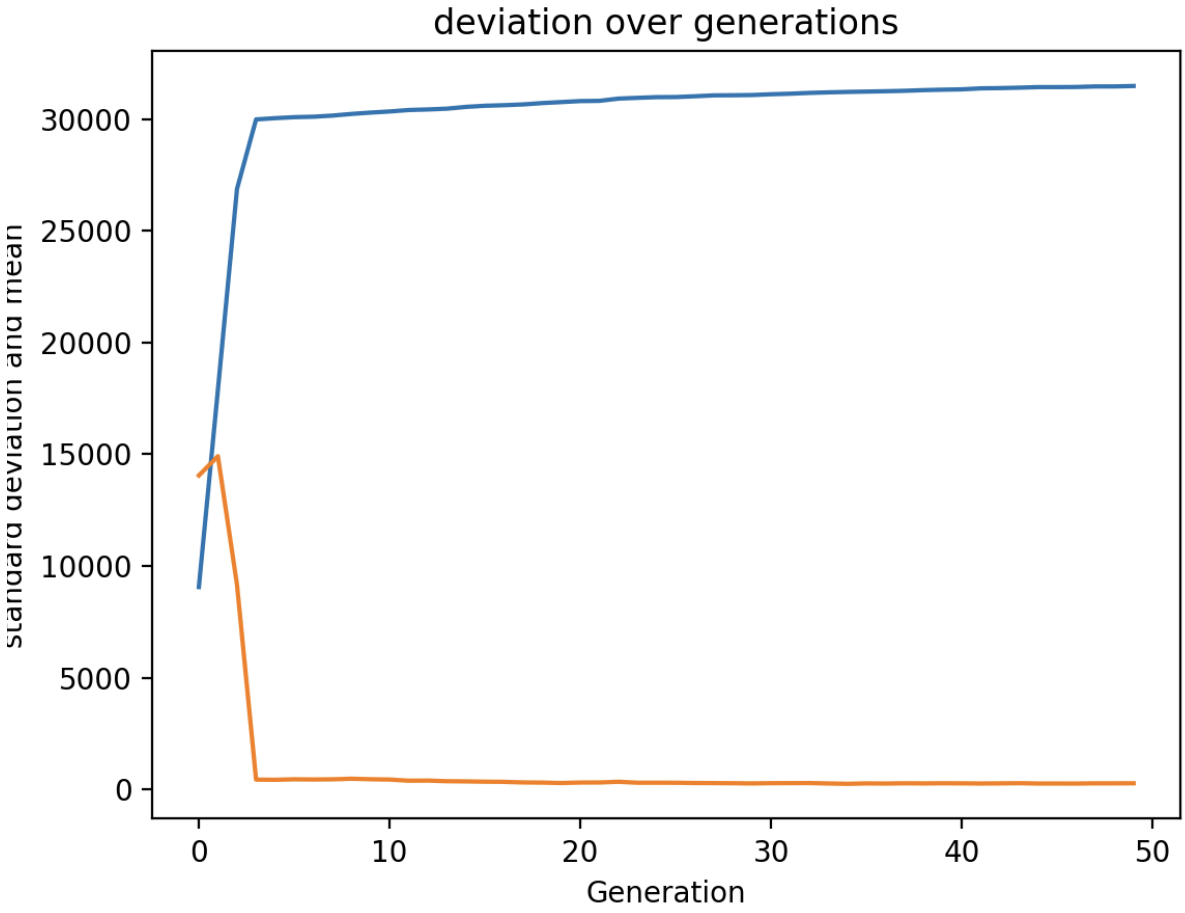
Results for the Randomly generated test case with set count 250




```
531299783, 184.88972531299783, 201.57276340658828, 1
mean of subsets used 30.05
2024-09-07 08:28:32.822 Python[60939:8867124] WARNIN
applicationSupportsSecureRestorableState: and returr
```

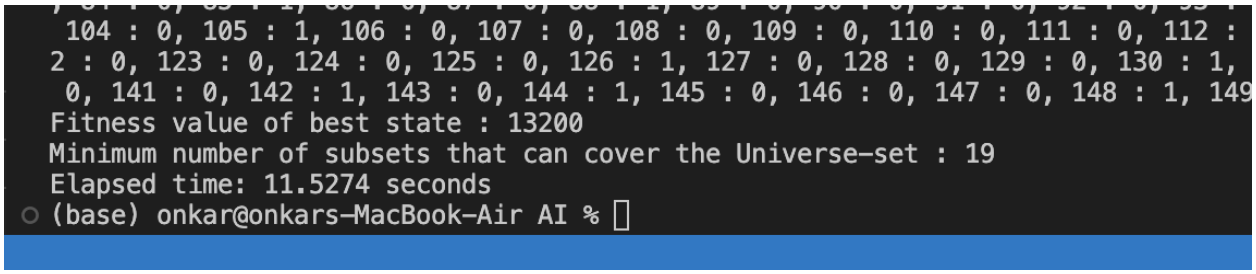


Results for the Randomly generated test case with set count 350




```
897635, 290.1905000440047, 275.0598021100904, 275.059802110
mean of subsets used 36.25
2024-09-07 08:30:56.048 Python[61000:8868831] WARNING: Secu
applicationSupportsSecureRestorableState: and returning YES
```

B) Improvements:

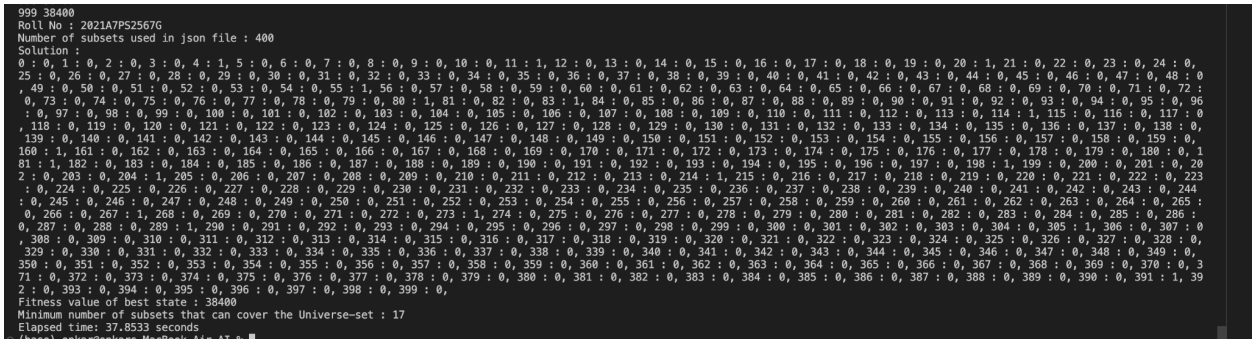


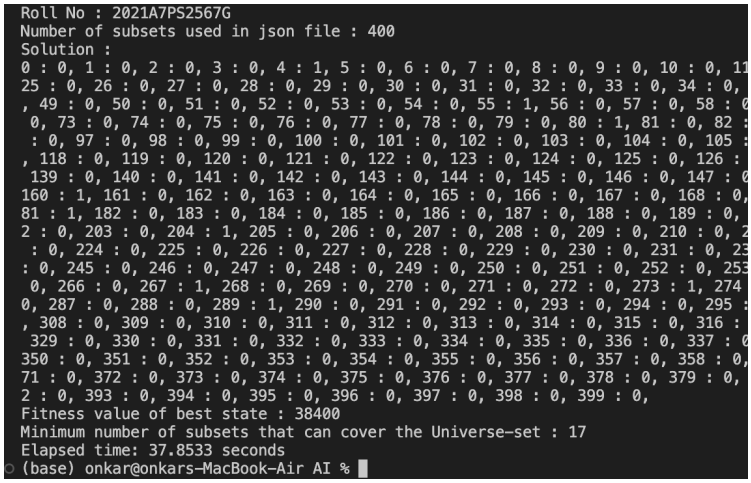
Increase in number of generations and trying out more hybrids

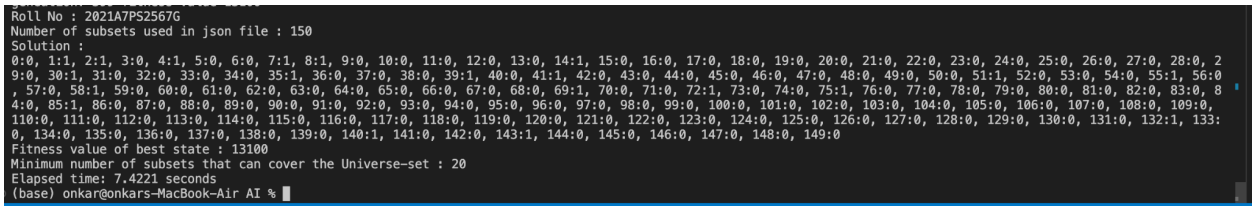
```
values = []
for generations in range (0,1000):
    fitval.clear()
    score=[]
    tfi = []
```

```
# print(fitval)
for hybrid in range(0,300):
    chosen_elements = random.choices(list, score, k=2)
    rand_value=random.randint(0,n-1)
    strr = ""
```

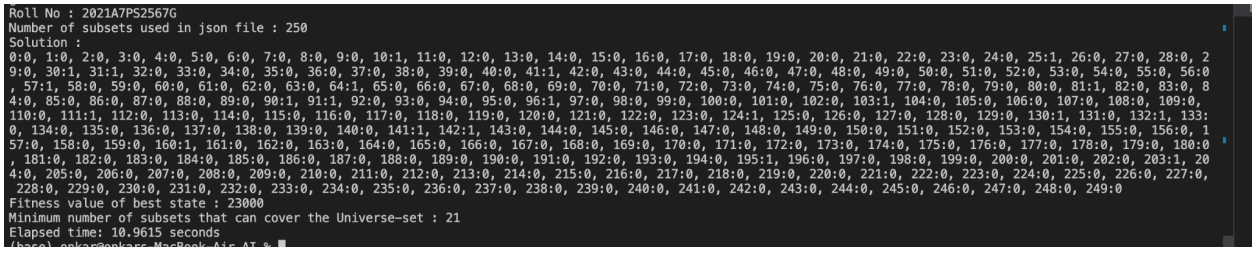
Peak Performance :







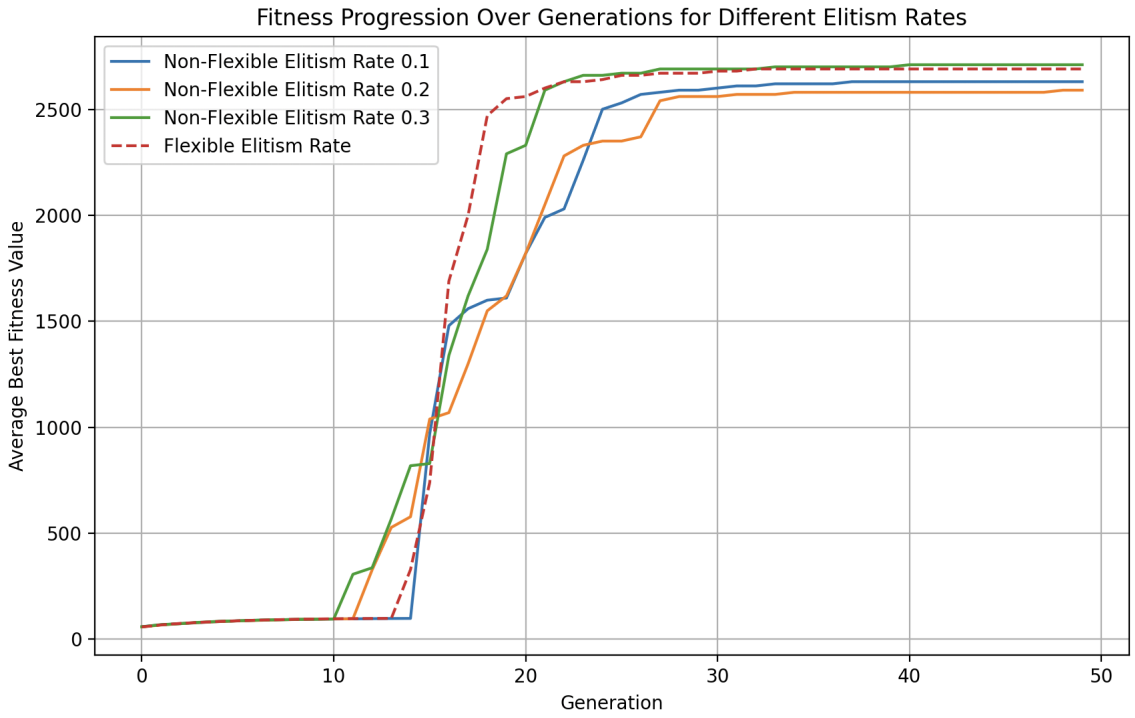
For total set size = 150



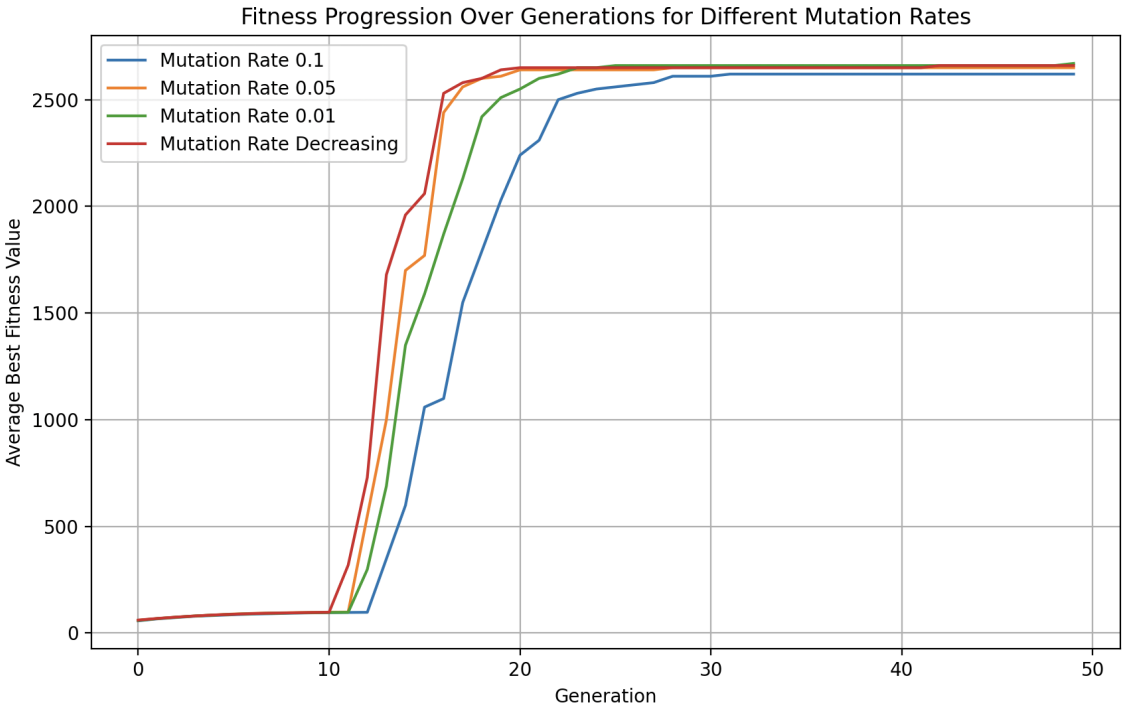
For total set size = 150

Major modifications that stand out :

- Generated multiple offsprings from the chosen 2 parents and considered them further based on their fitness values
- Different mutation rate for included and excluded subsets as our end goal is to reduce the included subsets
- Time efficient code which can be easily extended to multiple generations within the given time limit
- Initial string choice taking it random but keeping 0's and 1's proportionate (optimized at 10('0') : 1('1'))
- Elitism kept flexible (max : 20%) varies based on the quality of the next generation giving 10% better performance average than the conventional elitism algorithm.



- Decrease mutation rate as generations progress (e.g., mutation rate = 1/gen).



Future Aspects (not implemented as the above steps made the code go pretty close to optimal):

- Increase population size after every few generations to add fresh individuals and increase exploration.
- Penalize individuals with high similarity to others to maintain diverse solutions.
- Implement 2 or 3-point crossover to allow more complex recombination between parent solutions.
- Initialize a portion of the population using a greedy heuristic for SCP, ensuring better initial coverage based on greedy approx set covering algorithm.