2021A7PS2627G

Kislay Ranjan Nee Tandon

Brief:

This implementation of a genetic algorithm tackles the Set Covering Problem (SCP) by representing potential solutions as binary strings. Each bit in the string corresponds to a subset in the collection, with a one indicating inclusion of the subgroup and a 0 indicating exclusion.

The algorithm starts by initializing a population of random individuals. Each individual is evaluated using a fitness function that balances two goals: maximizing the coverage of the universe (i.e., ensuring that the selected subsets cover all elements from the universe) and minimizing the number of subsets used.

Selection is based on fitness, where fitter individuals are likely to be chosen as parents for the next generation. The crossover function combines two parent solutions to create offspring, while mutation introduces small changes, maintaining genetic diversity and helping the algorithm escape local optima.

The algorithm runs for several generations or until a time limit is reached. Throughout the process, the best solution and its fitness are tracked. The approach is repeated for different collection sizes, demonstrating the algorithm's performance under various conditions. This implementation effectively demonstrates how genetic algorithms can be adapted to solve complex combinatorial problems like SCP.

The Default mutation rate is 0.01.

Question 1 :

Implementing Genetic Algorithm for SCP

```python
def fitness_function(individual, subsets):
    covered = set()
    count = 0
    for i, bit in enumerate(individual):
        if bit:
            covered.update(subsets[i])
            count += 1
    coverage = len(covered) / 100  # Assuming universe size is always 100
    return (coverage - (count / len(subsets))) * 100
```

Fitness Function Summary:

The fitness function evaluates how effectively a subset selection covers the universe while minimizing the number of subsets used. It balances coverage (the proportion of the universe covered by the selected subsets) with efficiency (using the fewest subsets possible). The goal is to maximize this balance, resulting in higher fitness values for better solutions.

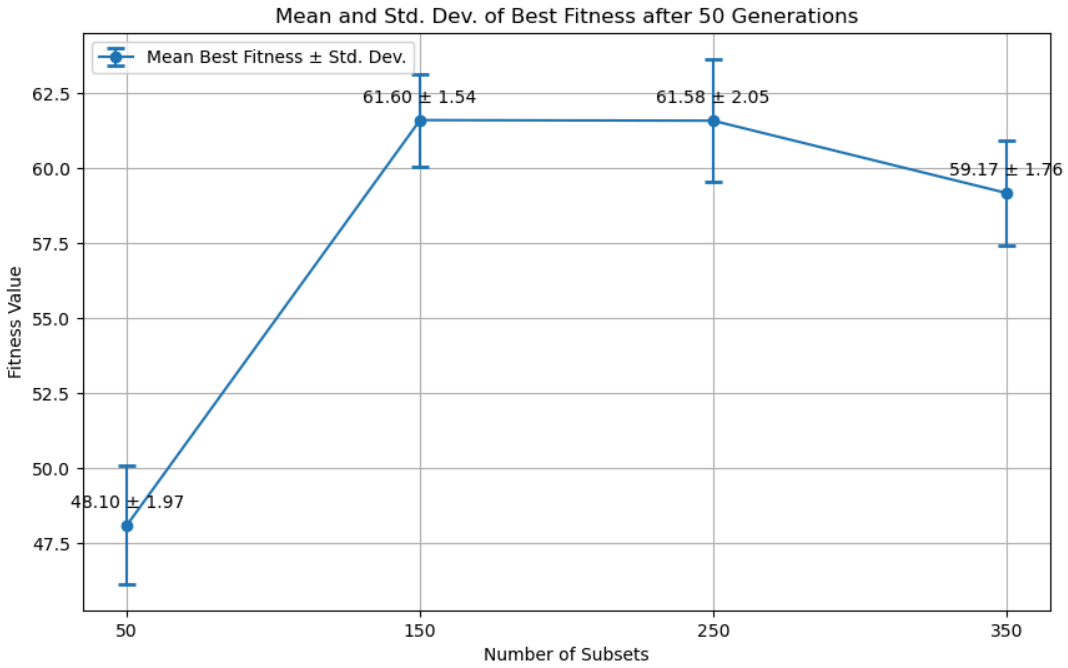


Figure 1: Mean Best Fitness Value over 50 generations

This graph illustrates the performance of the genetic algorithm for different subset sizes (50, 150, 250, 350) over 50 generations.

Key observations:

- Mean fitness improves significantly as the number of subsets increases from 50 to 150
- Peak performance is observed around 250 subsets
- A slight decrease in performance occurs when increasing to 350 subsets
- Standard deviation decreases up to 150 subsets, indicating more consistent results
- Beyond 250 subsets, the standard deviation rises slightly, suggesting reduced consistency

.

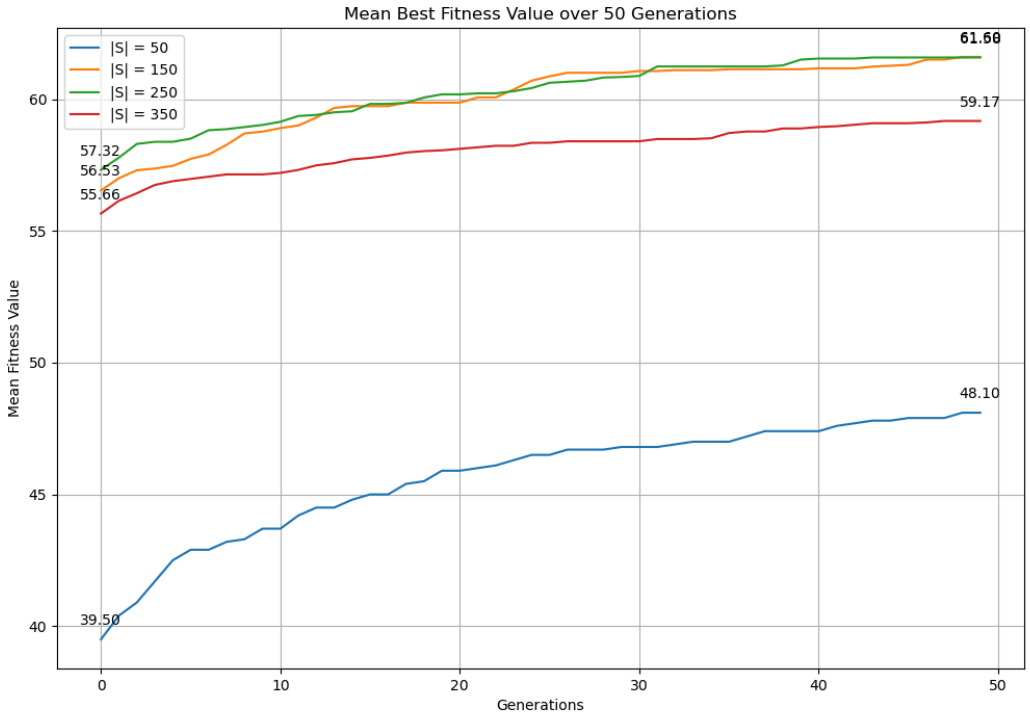


Figure 2: Mean and Standard Deviation of best Fitness values after 50 generations.

This figure provides a snapshot of the algorithm's performance after 50 generations for various collection sizes.

Key findings:

- Optimal performance is achieved with |S| = 150 subsets (mean fitness: 61.60 ± 1.54)
- Performance plateaus and slightly decreases for larger collections
- Consistent performance across multiple runs, as indicated by small standard deviations
- Significantly lower fitness for |S| = 50 (48.10 ± 1.97), likely due to insufficient subsets

In conclusion, the genetic algorithm shows effective optimization behavior for the SCP, with performance peaking at a collection size of around 150 subsets. The algorithm consistently improves solutions over generations, with most improvement occurring in early generations. Future work could explore tuning algorithm parameters or alternative genetic operators to enhance performance, especially for more significant problem instances.

Figure 2

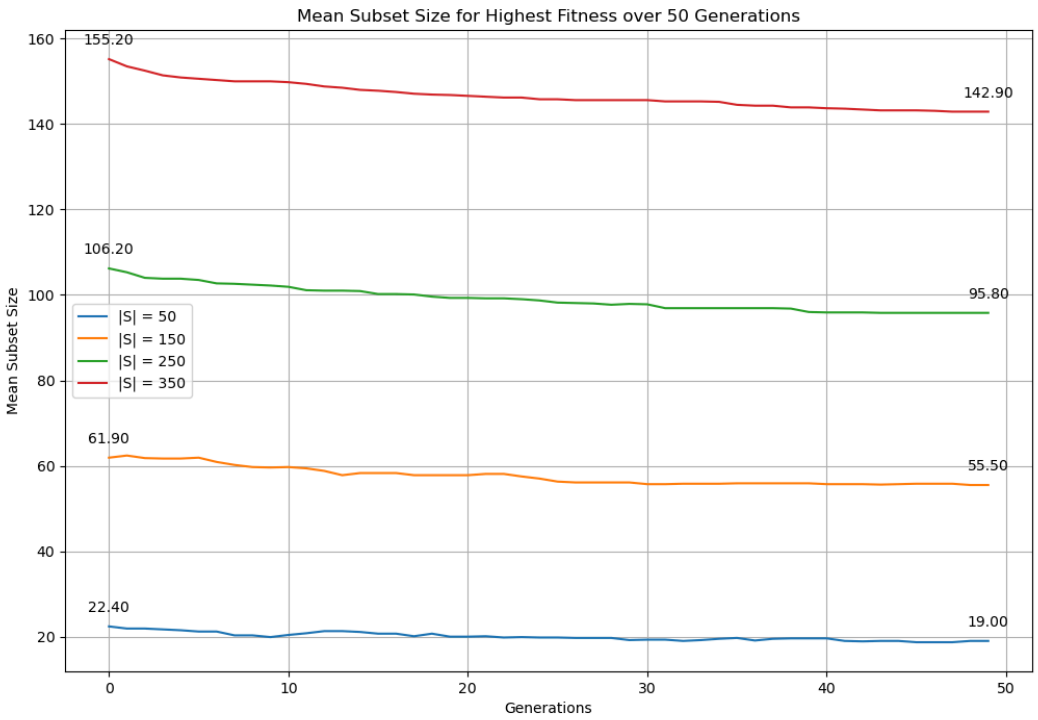


Figure 3: Mean Subset size over a generation

This graph shows how the mean subset size for the highest fitness solution changes over 50 generations for different collection sizes.

Observations:

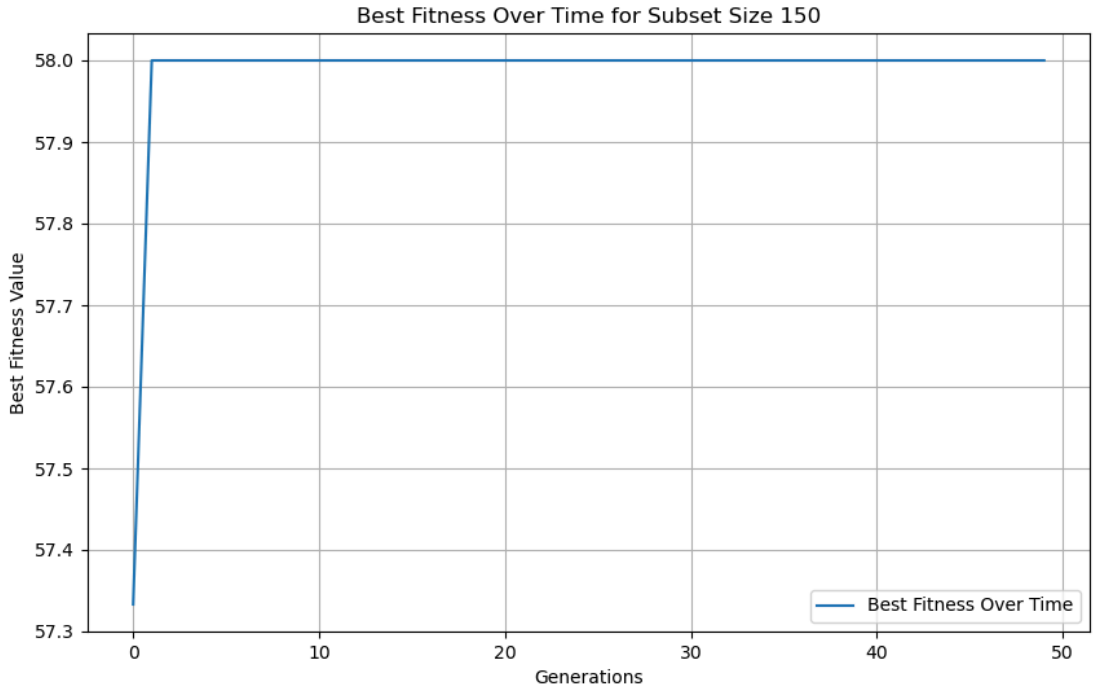
1. Larger collection sizes result in solutions with larger subset sizes
2. Mean subset size decreases over generations for all collection sizes
3. The improvement rate is fastest in early generations, slowing down later
4. Smaller collection sizes show more stability in subset size across generations

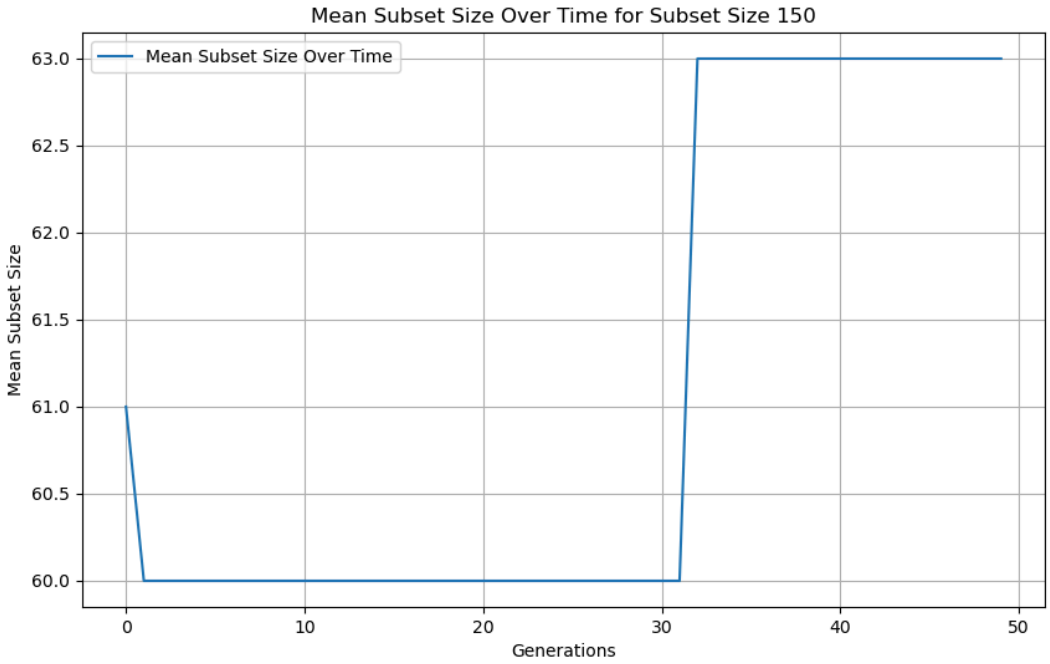
Question 2 :

Improving the Algorithm

Before making any improvements, we need to assess the performance of our current implementation. Currently, it yields a fitness value of 58 and a minimum number of subsets around 63.

```
(base) kislayranjanneetandon@Kislays-Laptop AI_assignment_1_Sem_1_24-25 % python -u "/Users/kislayranjanneetandon/Documents/Code/AI/AI_assignme
nt_1_Sem_1_24-25/2021A7PS2627G_KISLAY.py"
Number of sets: 150
Solution: 0:1, 1:0, 2:1, 3:1, 4:0, 5:1, 6:0, 7:1, 8:0, 9:0, 10:0, 11:1, 12:1, 13:0, 14:0, 15:1, 16:0, 17:0, 18:0, 19:0, 20:0, 21:1, 22:0, 23:0,
 24:0, 25:1, 26:0, 27:1, 28:0, 29:0, 30:0, 31:1, 32:0, 33:1, 34:1, 35:0, 36:1, 37:1, 38:0, 39:0, 40:0, 41:1, 42:1, 43:0, 44:1, 45:1, 46:1, 47:0
, 48:0, 49:0, 50:0, 51:1, 52:1, 53:1, 54:0, 55:0, 56:1, 57:1, 58:1, 59:0, 60:0, 61:0, 62:1, 63:1, 64:0, 65:0, 66:0, 67:1, 68:0, 69:1, 70:1, 71:
1, 72:0, 73:0, 74:0, 75:0, 76:0, 77:1, 78:1, 79:0, 80:1, 81:0, 82:0, 83:0, 84:0, 85:1, 86:1, 87:0, 88:1, 89:0, 90:0, 91:0, 92:0, 93:1, 94:0, 95
:0, 96:0, 97:1, 98:1, 99:0, 100:0, 101:1, 102:0, 103:1, 104:1, 105:0, 106:1, 107:0, 108:1, 109:0, 110:0, 111:1, 112:1, 113:1, 114:0, 115:0, 116
:0, 117:1, 118:0, 119:0, 120:0, 121:0, 122:1, 123:0, 124:1, 125:1, 126:0, 127:0, 128:0, 129:0, 130:1, 131:0, 132:1, 133:1, 134:0, 135:0, 136:0,
 137:0, 138:1, 139:1, 140:0, 141:0, 142:1, 143:0, 144:1, 145:1, 146:1, 147:0, 148:0, 149:0,
Fitness value of best state: 58.00000000000001
Minimum number of subsets that can cover the universe set: 63
Time taken: 0.032 seconds
```



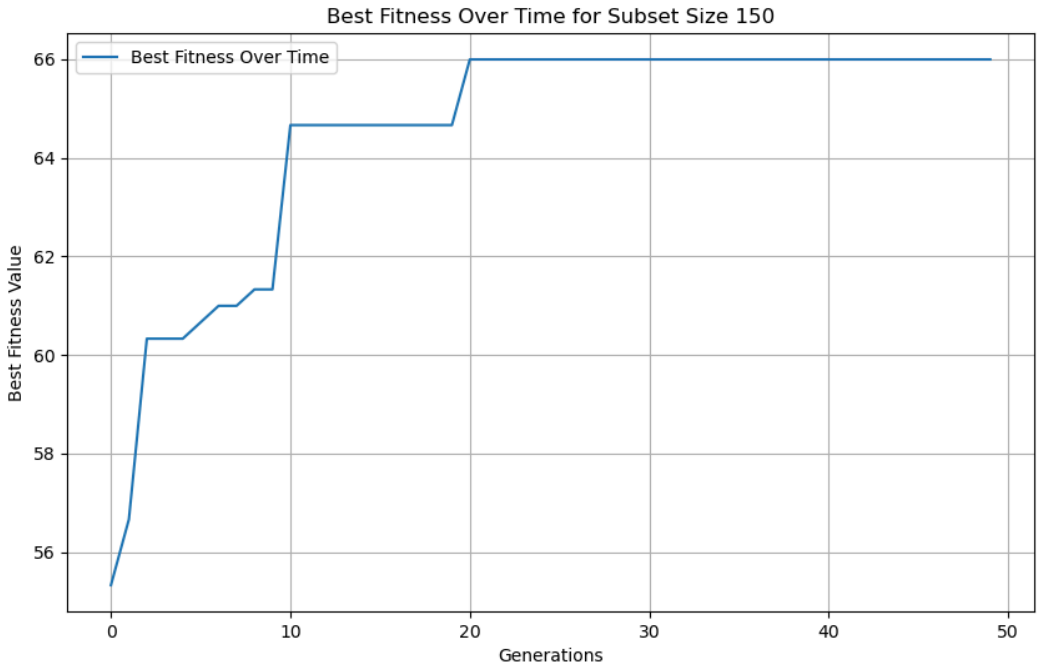


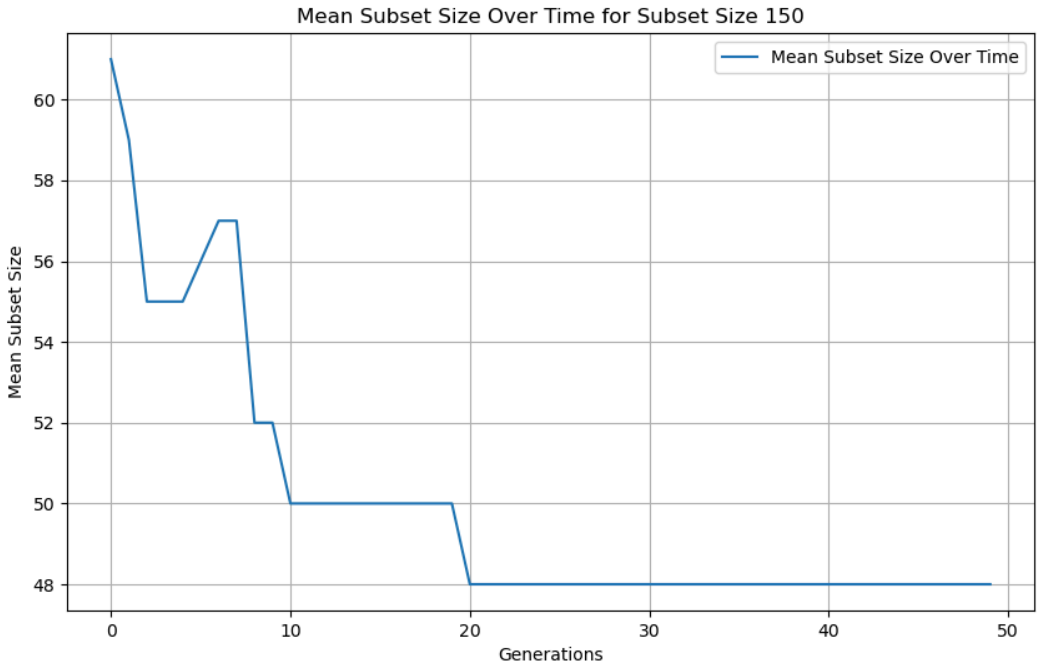
Inserting both Child

In genetic algorithms, the crossover operation is crucial for combining the genetic information of two parent solutions to produce new offspring. The modification to return two children instead of one enhances the diversity of the population, which can lead to more effective exploration of the solution space. By selecting a random crossover point, the algorithm creates two new individuals: one by combining the first segment of the first parent with the second segment of the second parent and the other by doing the reverse. This approach ensures that both offspring inherit different traits from each parent, increasing the likelihood of discovering optimal solutions. Such modifications can significantly improve the performance of genetic algorithms in solving complex optimization problems by maintaining a healthy balance between exploration and exploitation.

After incorporating both children into the genetic algorithm, we observed improvements in our results. Initially, the fitness value was 58, with a minimum number of subsets around 63. Following the addition of the new children, the fitness value increased to approximately 65, and the minimum number of subsets was reduced to 49.

```
(base) kislayranjanneetandon@Kislays-Laptop AI_assignment_1_Sem_1_24-25 % python -u "/Users/kislayranjanneetandon/Documents/Code/AI/AI_assignme
nt_1_Sem_1_24-25/2021A7PS2627G_KISLAY.py"
Number of sets: 150
Solution: 0:0, 1:1, 2:0, 3:0, 4:0, 5:0, 6:0, 7:0, 8:0, 9:1, 10:0, 11:0, 12:1, 13:0, 14:0, 15:0, 16:0, 17:0, 18:0, 19:0, 20:1, 21:0, 22:1, 23:0,
 24:0, 25:0, 26:1, 27:0, 28:0, 29:0, 30:1, 31:0, 32:0, 33:1, 34:0, 35:0, 36:1, 37:0, 38:0, 39:0, 40:0, 41:1, 42:0, 43:0, 44:0, 45:0, 46:0, 47:0
, 48:0, 49:0, 50:0, 51:0, 52:0, 53:0, 54:0, 55:0, 56:1, 57:0, 58:0, 59:0, 60:1, 61:1, 62:0, 63:1, 64:1, 65:1, 66:0, 67:1, 68:1, 69:0, 70:1, 71:
0, 72:1, 73:0, 74:0, 75:1, 76:1, 77:1, 78:0, 79:0, 80:0, 81:0, 82:1, 83:0, 84:1, 85:0, 86:0, 87:0, 88:1, 89:0, 90:1, 91:1, 92:1, 93:1, 94:0, 95
:1, 96:0, 97:1, 98:0, 99:1, 100:0, 101:0, 102:0, 103:0, 104:1, 105:0, 106:0, 107:0, 108:0, 109:1, 110:1, 111:1, 112:0, 113:0, 114:0, 115:1, 116
:0, 117:0, 118:0, 119:1, 120:1, 121:1, 122:0, 123:0, 124:0, 125:0, 126:0, 127:0, 128:0, 129:0, 130:0, 131:1, 132:1, 133:0, 134:0, 135:0, 136:0,
 137:1, 138:0, 139:1, 140:0, 141:1, 142:0, 143:0, 144:1, 145:0, 146:0, 147:0, 148:1, 149:0,
Fitness value of best state: 65.99999999999999
Minimum number of subsets that can cover the universe set: 48
Time taken: 0.027 seconds
```



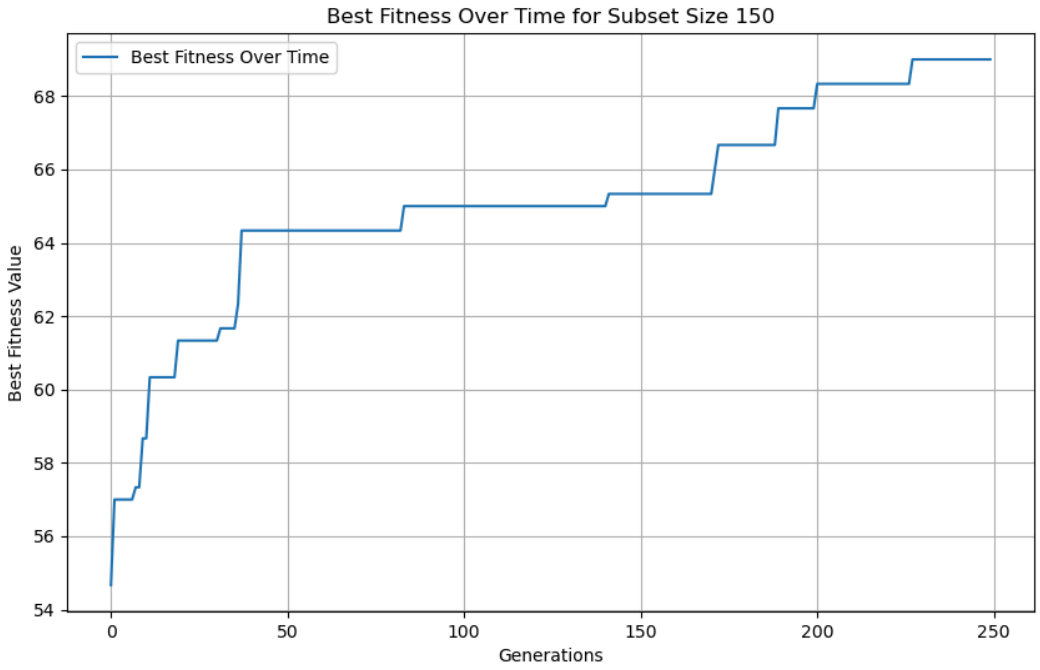


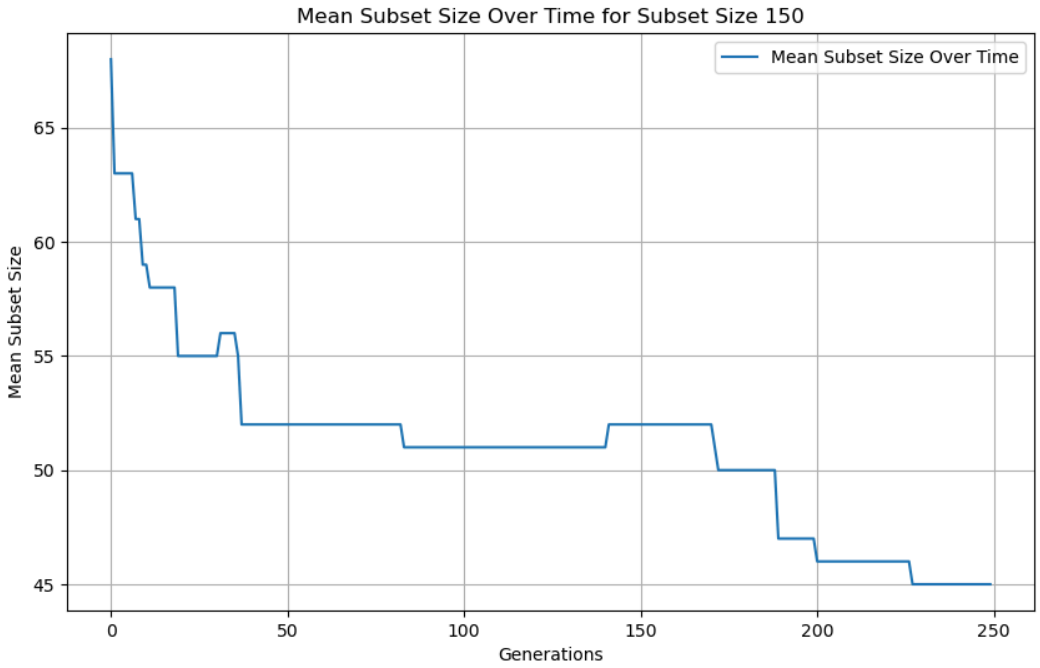
Changing Mutation Rate and Number of Generation

In genetic algorithms, the mutation rate is crucial for maintaining genetic diversity and exploring the solution space. Initially, a higher mutation rate, such as 0.3, encourages exploration by introducing more variability. This helps the algorithm discover new solutions and avoid local optima. As the algorithm progresses, gradually decreasing the mutation rate to around 0.01 shifts the focus from exploration to exploitation; also, to see this over significant observable distance, we are increasing generations from 50 to 250.

After implementing these changes, the algorithm achieved a fitness value of 69 and a minimum number of subsets of 45. This outcome indicates that the adjustments in mutation rate and the increase in the number of generations contributed to a more effective search and optimization process. The higher fitness value and reduced number of subsets reflect improved solution quality and efficiency.

```
(base) kislayranjanneetandon@Kislays-Laptop AI_assignment_1_Sem_1_24-25 % python -u "/Users/kislayranjanneetandon/Documents/Code/AI/AI_assignment_1_Sem_1_24-25/2021A7PS2627G_KISLAY.py"
Number of sets: 150
Solution: 0:0, 1:0, 2:1, 3:1, 4:0, 5:1, 6:1, 7:1, 8:1, 9:0, 10:0, 11:1, 12:0, 13:0, 14:0, 15:0, 16:0, 17:0, 18:1, 19:1, 20:0, 21:1, 22:0, 23:1, 24:0, 25:1, 26:1, 27:1, 28:0, 29:1, 30:1, 31:0, 32:1, 33:0, 34:0, 35:0, 36:1, 37:0, 38:1, 39:0, 40:1, 41:0, 42:0, 43:0, 44:0, 45:0, 46:0, 47:0, 48:1, 49:0, 50:0, 51:1, 52:0, 53:0, 54:0, 55:0, 56:0, 57:0, 58:0, 59:1, 60:0, 61:1, 62:0, 63:0, 64:0, 65:1, 66:0, 67:0, 68:0, 69:0, 70:0, 71:1, 72:0, 73:1, 74:0, 75:1, 76:1, 77:0, 78:0, 79:0, 80:0, 81:0, 82:0, 83:0, 84:0, 85:1, 86:0, 87:0, 88:0, 89:0, 90:0, 91:0, 92:1, 93:1, 94:0, 95:0, 96:0, 97:0, 98:0, 99:0, 100:0, 101:0, 102:0, 103:0, 104:0, 105:1, 106:1, 107:0, 108:1, 109:0, 110:0, 111:0, 112:1, 113:1, 114:0, 115:0, 116:0, 117:0, 118:1, 119:1, 120:0, 121:0, 122:0, 123:0, 124:0, 125:0, 126:0, 127:1, 128:1, 129:0, 130:0, 131:0, 132:0, 133:0, 134:0, 135:0, 136:0, 137:1, 138:0, 139:0, 140:0, 141:0, 142:0, 143:0, 144:1, 145:0, 146:1, 147:0, 148:1, 149:0,
Fitness value of best state: 69.0
Minimum number of subsets that can cover the universe set: 45
Time taken: 0.113 seconds
```



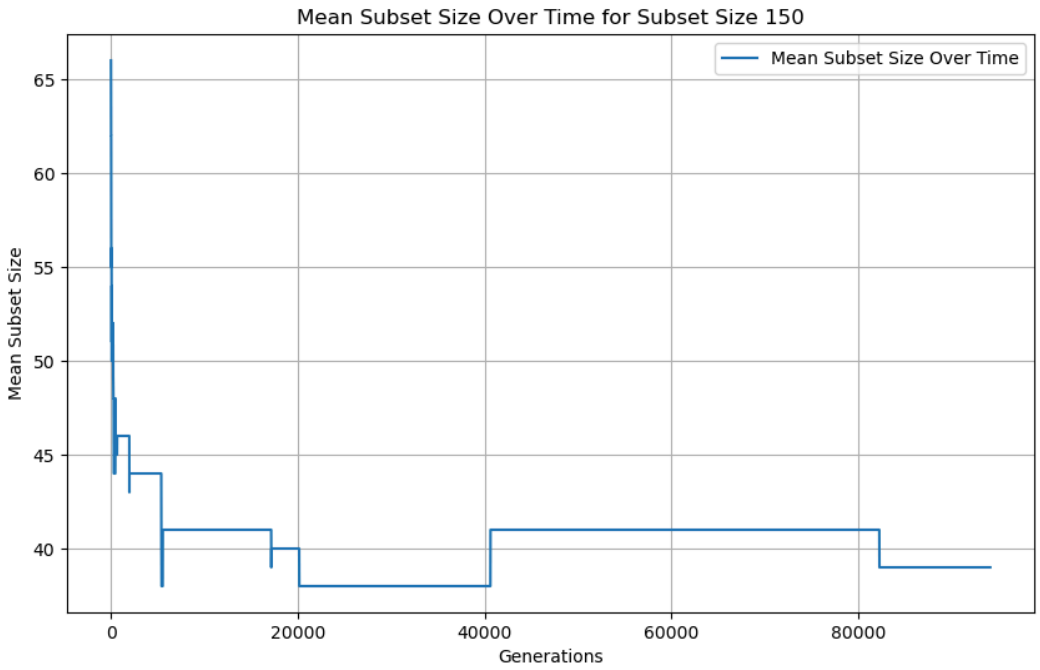


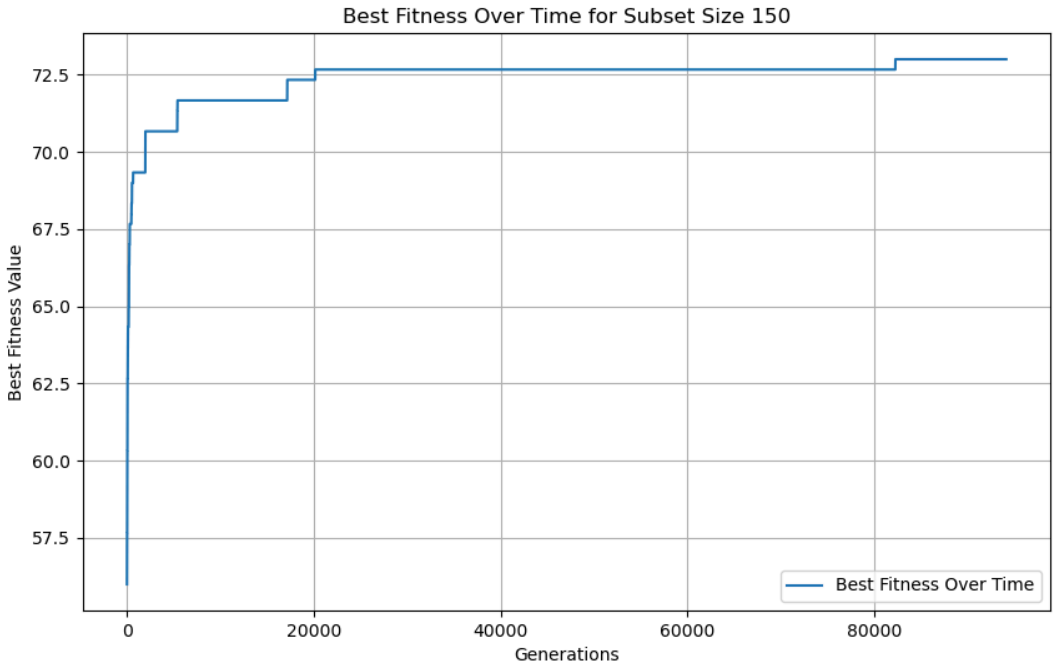
Utilizing a time limit of 40 seconds

In the experiment, the genetic algorithm was modified to run each instance for a fixed duration of 40 seconds. This adjustment aimed to evaluate the impact of a time constraint on the algorithm's performance. It was observed that the 40-second runtime influenced the quality of the solutions. The following results were given for each of these subsets.

As a result of this time constraint, the algorithm achieved a fitness value of 73 and a minimum number of subsets of 39. This indicates that the shorter runtime still allowed the algorithm to find solutions that were both more optimal and efficient compared to previous runs. The increase in fitness value and the reduction in the number of subsets suggest that the algorithm was able to perform effectively even under a fixed time limit.

```
nt_1_Sem_1_24-25/2021A7PS2627G_KISLAY.py"
Time limit reached.
Number of sets: 150
Solution: 0:0, 1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:1, 8:0, 9:0, 10:0, 11:0, 12:1, 13:0, 14:0, 15:1, 16:0, 17:1, 18:0, 19:0, 20:0, 21:1, 22:0, 23:1,
 24:0, 25:0, 26:0, 27:0, 28:1, 29:1, 30:0, 31:1, 32:0, 33:1, 34:0, 35:0, 36:0, 37:0, 38:0, 39:0, 40:0, 41:0, 42:0, 43:0, 44:0, 45:0, 46:0, 47:0
, 48:1, 49:0, 50:1, 51:0, 52:1, 53:0, 54:0, 55:0, 56:0, 57:0, 58:0, 59:0, 60:0, 61:1, 62:0, 63:0, 64:0, 65:1, 66:0, 67:0, 68:1, 69:0, 70:0, 71:
0, 72:0, 73:0, 74:0, 75:0, 76:0, 77:0, 78:0, 79:0, 80:0, 81:0, 82:1, 83:0, 84:1, 85:0, 86:1, 87:0, 88:0, 89:0, 90:0, 91:1, 92:0, 93:0, 94:1, 95
:0, 96:1, 97:0, 98:1, 99:0, 100:1, 101:0, 102:1, 103:0, 104:1, 105:0, 106:1, 107:0, 108:0, 109:0, 110:1, 111:0, 112:0, 113:0, 114:0, 115:1, 116
:0, 117:0, 118:0, 119:1, 120:1, 121:0, 122:0, 123:0, 124:0, 125:0, 126:0, 127:0, 128:0, 129:1, 130:0, 131:1, 132:1, 133:0, 134:0, 135:0, 136:0,
 137:1, 138:1, 139:0, 140:1, 141:0, 142:0, 143:0, 144:1, 145:0, 146:0, 147:1, 148:0, 149:0,
Fitness value of best state: 73.0
Minimum number of subsets that can cover the universe set: 39
Time taken: 40.000 seconds
```





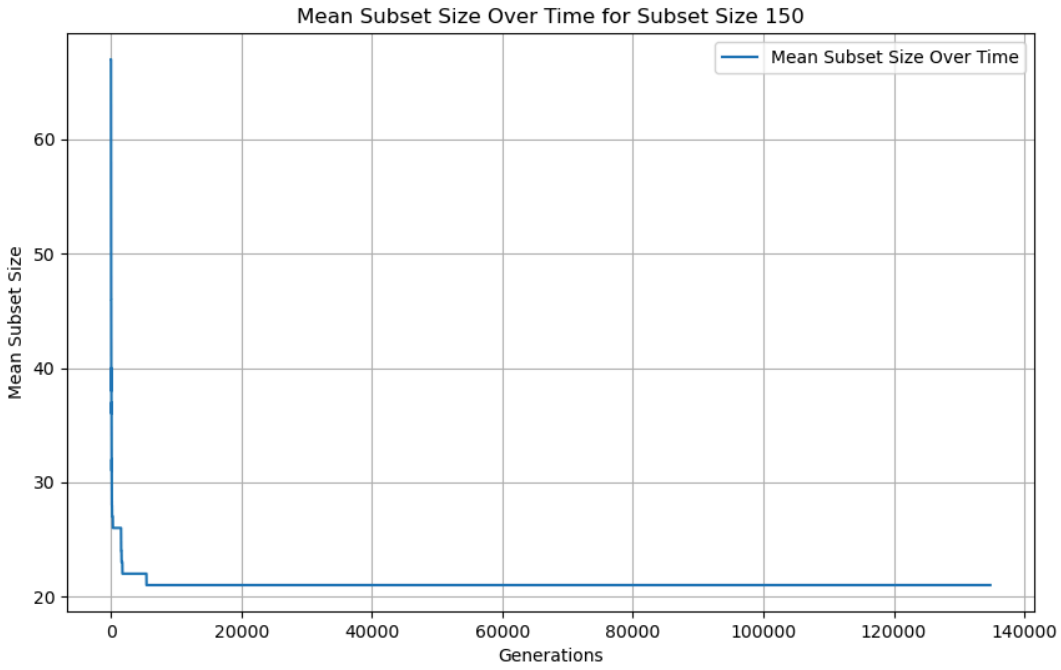
Introducing Elitism

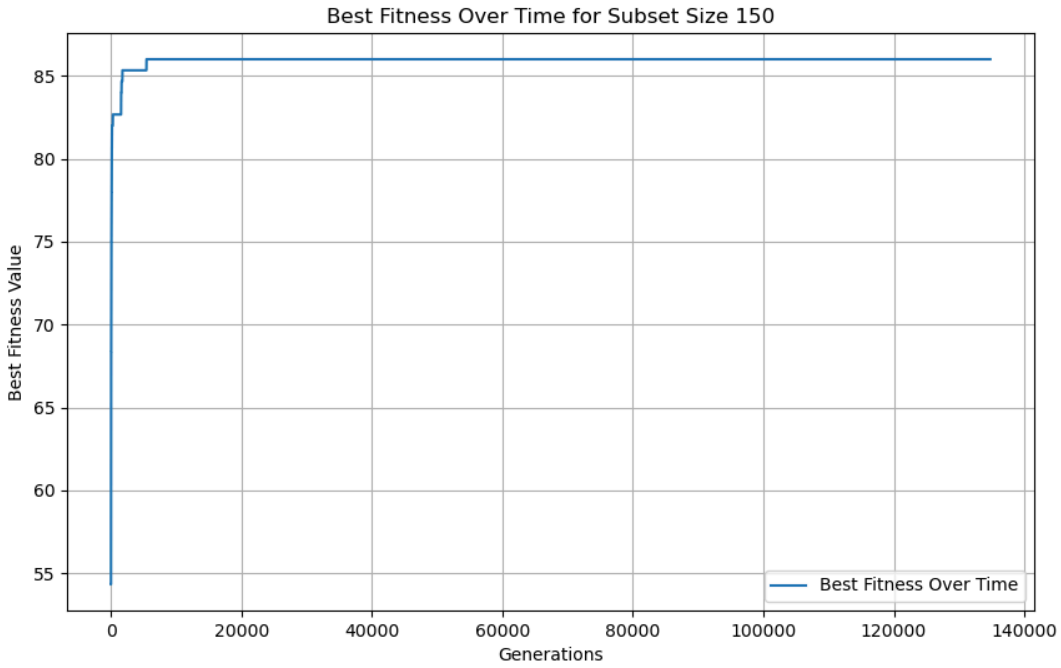
Elitism is a technique used in genetic algorithms to preserve the best-performing solutions from one generation to the next. By ensuring that the best individuals (or elites) are carried over unchanged, elitism helps maintain high-quality solutions and improves convergence rates. This approach can prevent the loss of optimal or near-optimal solutions due to the randomness of crossover and mutation processes. Introducing elitism involves selecting a portion of the population with the highest fitness and directly passing it to the next generation. In contrast, the rest undergoes genetic operations like crossover and mutation.

In practice, introducing elitism involves selecting a portion of the population with the highest fitness and directly passing these elite solutions to the next generation. Meanwhile, the remaining individuals undergo genetic operations such as crossover and mutation.

After incorporating elitism into the genetic algorithm, the results showed a significant improvement, with a fitness value of 86 and a minimum number of subsets of 21. This outcome indicates that elitism effectively enhanced the algorithm's performance, leading to higher solution quality and greater efficiency in the number of subsets.

```
nt_1_Sem_1_24-25/2021A7PS2627G_KISLAY.py"
Time limit reached.
Number of sets: 150
Solution: 0:0, 1:1, 2:0, 3:0, 4:0, 5:1, 6:1, 7:0, 8:0, 9:1, 10:0, 11:0, 12:0, 13:0, 14:0, 15:0, 16:0, 17:1, 18:0, 19:1, 20:1, 21:0, 22:0, 23:0,
 24:0, 25:1, 26:0, 27:0, 28:0, 29:0, 30:0, 31:0, 32:0, 33:0, 34:0, 35:0, 36:0, 37:0, 38:0, 39:1, 40:0, 41:0, 42:0, 43:0, 44:1, 45:0, 46:1, 47:0
, 48:0, 49:0, 50:0, 51:0, 52:0, 53:0, 54:0, 55:0, 56:1, 57:0, 58:0, 59:0, 60:0, 61:0, 62:0, 63:0, 64:1, 65:1, 66:1, 67:0, 68:0, 69:0, 70:0, 71:
0, 72:0, 73:0, 74:0, 75:0, 76:0, 77:0, 78:0, 79:0, 80:0, 81:0, 82:0, 83:0, 84:0, 85:0, 86:0, 87:0, 88:0, 89:0, 90:0, 91:0, 92:1, 93:1, 94:0, 95
:0, 96:1, 97:1, 98:0, 99:0, 100:0, 101:0, 102:0, 103:0, 104:0, 105:1, 106:0, 107:0, 108:0, 109:0, 110:0, 111:0, 112:0, 113:0, 114:0, 115:0, 116
:0, 117:0, 118:0, 119:0, 120:0, 121:0, 122:0, 123:0, 124:0, 125:0, 126:1, 127:0, 128:0, 129:0, 130:0, 131:0, 132:0, 133:0, 134:0, 135:0, 136:0,
 137:0, 138:0, 139:0, 140:0, 141:0, 142:0, 143:0, 144:0, 145:0, 146:0, 147:0, 148:0, 149:0,
Fitness value of best state: 86.0
Minimum number of subsets that can cover the universe set: 21
Time taken: 40.000 seconds
```





Conclusion

The genetic algorithm effectively solves the Set Covering Problem (SCP), with notable improvements in performance due to modifications in the inserting mutation rates and the introduction of elitism. Enhancements like returning two children per crossover and adjusting mutation rates contribute to better fitness values and reduced subset sizes. Introducing elitism further refines solution quality and efficiency, proving its effectiveness in preserving high-quality solutions. The algorithm's performance benefits from allowing more runtime, with significant fitness and efficiency improvements observed across various subset sizes. Future work could focus on tuning algorithm parameters and exploring alternative genetic operators to optimize the solution for more significant problem instances.