Genetic Set Cover

Naive Solution

I started by implementing the simple genetic algorithm mentioned in the textbook Artificial Intelligence: A Modern Approach.

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
    weights ← WEIGHTED-BY(population, fitness)
    population2 ← empty list
    for i = 1 to SIZE(population) do
      parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
      child ← REPRODUCE(parent1, parent2)
      if (small random probability) then child ← MUTATE(child)
      add child to population2
    population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

Figure 4.8 A genetic algorithm. Within the function, *population* is an ordered list of individuals, *weights* is a list of corresponding fitness values for each individual, and *fitness* is a function to compute these values.

Fitness Function

I used a fitness function which checks for coverage and gives a large bonus for full coverage. This property of providing a large bonus for full coverage makes it very likely that our final solution will have full coverage. We can also infer from the fitness function value if the coverage is full or not. If the value of the fitness function is greater than 100, we know that we fully cover the universe.

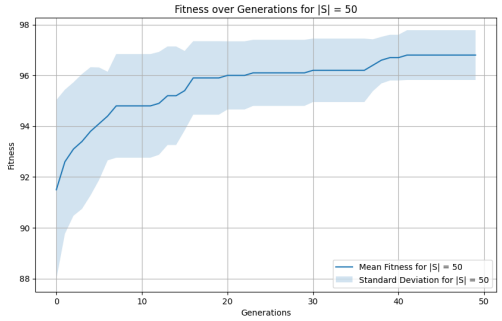
$$f(\text{individual}) = \begin{cases} 100 \times \left(\frac{\text{coverage}}{\text{universe\_size}}\right) & \text{if coverage} < \text{universe\_size} \\ 100 + 10000 \times \left(1 - \frac{\text{num\_selected}}{\text{len(subsets)}}\right) & \text{if coverage} = \text{universe\_size} \end{cases}$$

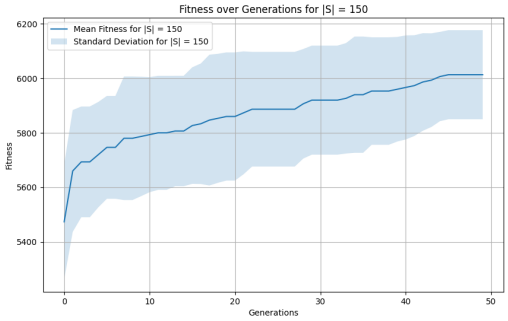
Where:

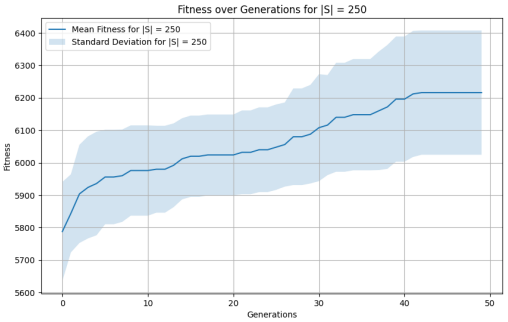
- `coverage` is the number of elements covered by the individual's selected subsets.
- `universe_size` is the total number of elements in the universe.
- `num_selected` is the number of subsets selected by the individual.
- `len(subsets)` is the total number of subsets available.

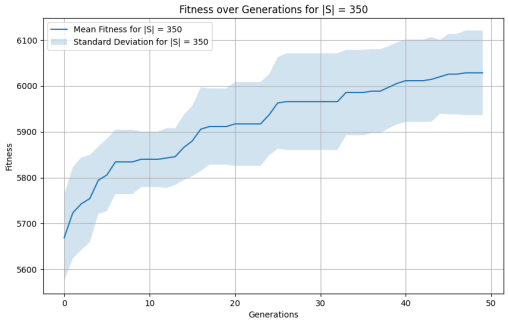
Plots

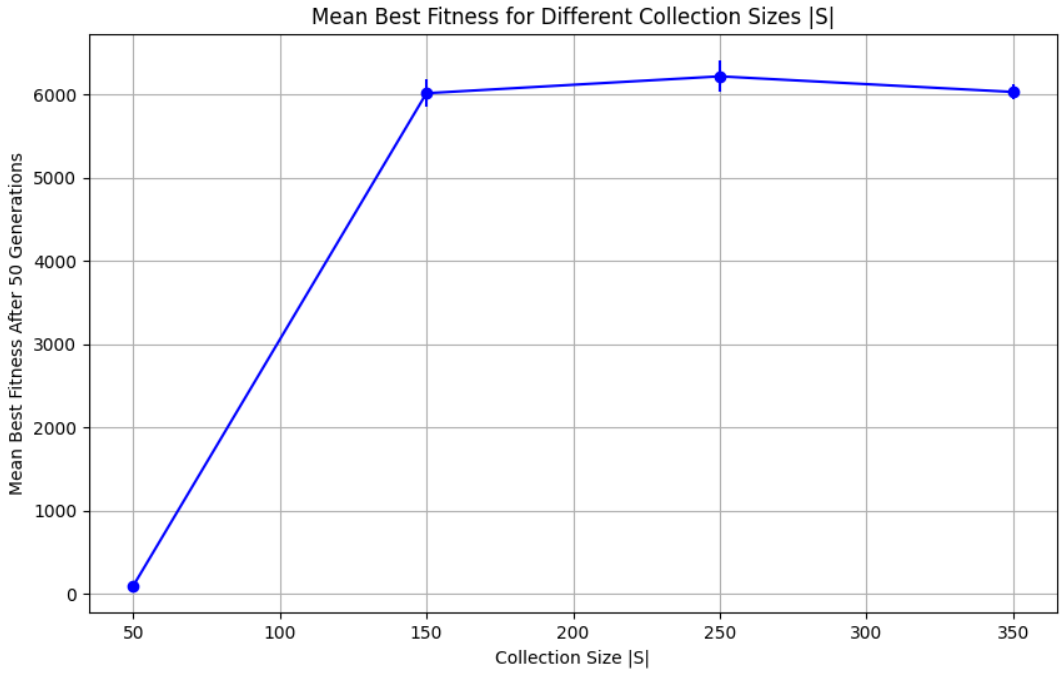
The plots of the fitness function over generations can be found for the sets of sizes 50, 150, 250 and 350 below. We can observe a large variation since the fitness function changes drastically as an individual moves from partial coverage to full coverage. This large bonus for full coverage makes it likely that we arrive at a solution which covers the entire universe.











Mean and Standard Deviation of the best fitness function after 50 generations. The standard deviation can be seen as a vertical line at the points.

Adding Improvements

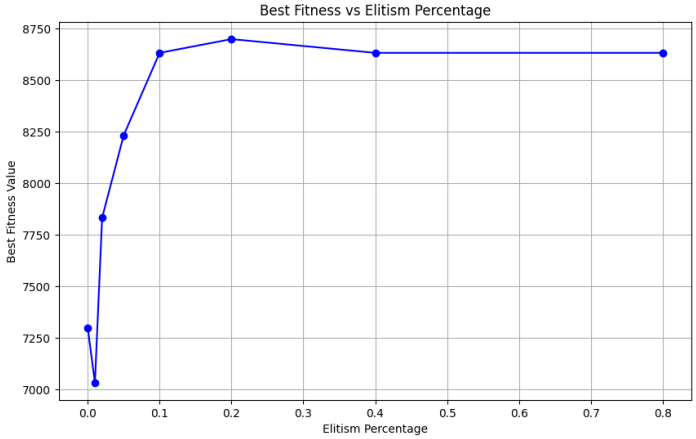
Elitism

As stated in Part A, I started with the naive approach. The first optimization which I used was to implement elitism.

This technique was chosen because:

- It allows us to preserve high-quality solutions
- Improves convergence speed

Below are the graphs for the best fitness achieved with varying percentages for elitism (starting with 0, which indicates no elitism).



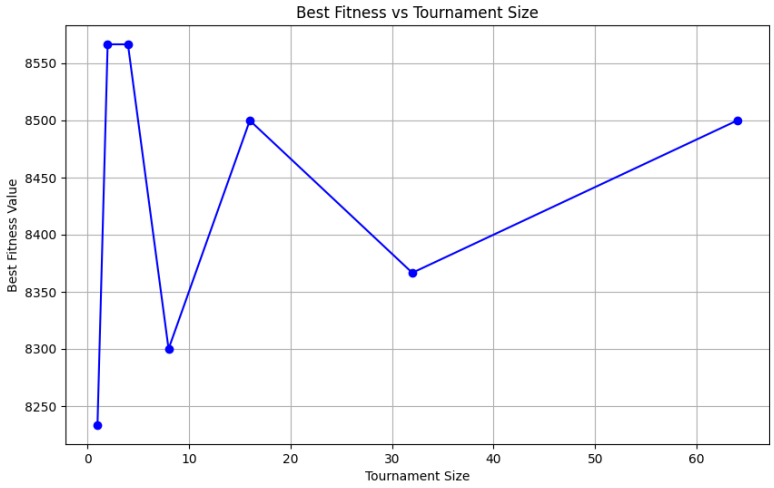
We can observe that elitism leads to an overall improvement in the best fitness we obtain. The results worsen after we reach the elitism value of 0.2; this can be potentially because we do not get enough diversity later on to improve.

Tournament Selection

I started using tournament selection because:

- Increased selection pressure: This increases convergence speed
- Preserving Diversity: Small tournament sizes help us explore the search space since we are less likely to consistently pick the best individuals
    - This works well in combination with elitism if we select a small tournament-size value

This has a similar effect on fitness as elitism if we select a large value. If we select a large tournament size, we might decrease diversity, and hence, we decrease fitness in the long run. This fact is reflected in our plot. We see a peak at 2 and 4 with a decline later on.

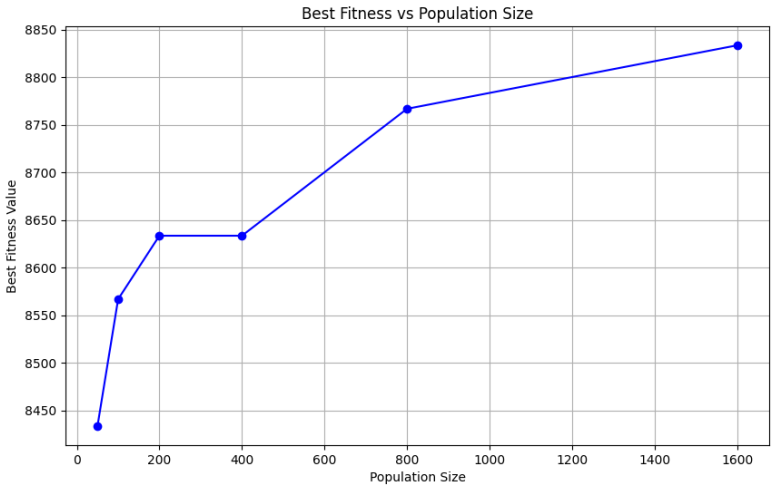


Population Size

I later increased the population of the genetic algorithm for the following reasons:

- Exploring the search space better
    - This was especially important since I observed my early stopping was getting triggered quite frequently, stopping the exploration.
    - The increased diversity after the increasing population helped combat the stagnation caused by early stopping

We see an increase in our final fitness function with an increase in population size. There is an obvious tradeoff between population size, computational resources required, and time taken for the program to run.

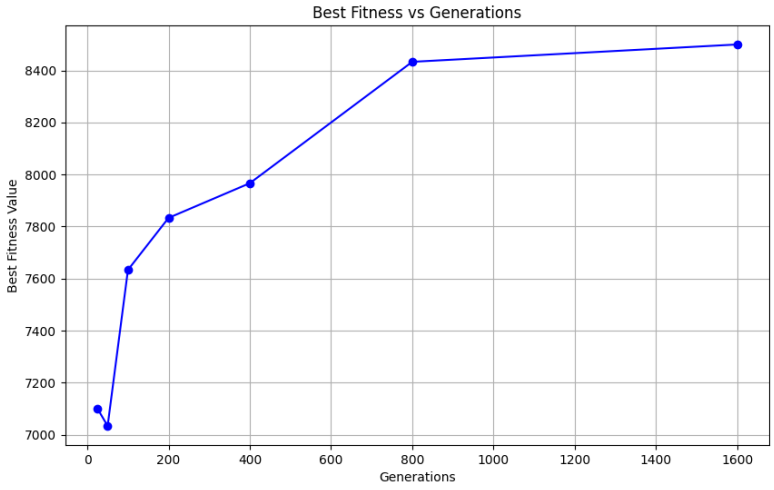


Uniform Crossover

I replaced the crossover method recommended in the textbook with a uniform crossover approach. Unlike the traditional method, which combines genes from parents at a specific cut point, uniform crossover selects each gene independently from either parent with equal probability. This approach enhances gene mixing and promotes greater diversity within the population.

In my test runs, adopting uniform crossover led to an average increase of 200 in the best fitness value compared to the traditional crossover method.

Generations



I next increased the number of generations I was running the algorithm for, this allows us to:

- Explore the search space better and obtain better solutions (especially if the mutation rate is high)

As expected, increasing the number of generations allows us to come to better quality solutions. The tradeoff we have to make when increasing the number of generations is the amount of time taken for the algorithm to converge. I also implemented early stopping if we do not see an improvement after 30% of the total generations.

Final Result

Finally, after incorporating all these results, I tweaked the parameters slightly. I settled on using the below parameters:

- `population_size`: 1000
- `generations`: 1000
- `mutation_rate`: 0.02
- `elitism_percentage`: 0.2
- `tournament_size`: 4

I was able to obtain the set cover by using **19 sets** using `scp_test.json`. I have added a random seed for reproducibility.