IDAI610-PS2-Report

Khawaja Abaid Ullah

October 10, 2024

Problem 1: Genetic Algorithm Implementation

Extra Credit: Custom Multiplicative Penalty-based fitness function for 0-1 Knapsack Problem

The fitness function for the multiplicative penalty-based method in the 0-1 Knapsack Problem can be expressed as:

$$F(\mathbf{x}) = V(\mathbf{x}) \cdot \phi(W(\mathbf{x})) \tag{1}$$

Where:

- $F(\mathbf{x})$ is the fitness of individual $\mathbf{x}$
- $V(\mathbf{x})$ is the total value of individual (solution) $\mathbf{x}$
- $\phi(W(\mathbf{x}))$ is the penalty factor based on the weight of solution $\mathbf{x}$

The components are calculated as follows:

$$V(\mathbf{x}) = \sum_{i=1}^{n} v_i x_i \tag{2}$$

$$\phi(W(\mathbf{x})) = \frac{1}{1 + \alpha \cdot \max(0, W(\mathbf{x}) - W_{max})} \tag{3}$$

$$W(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i \tag{4}$$

Where:

- n is the number of items
- v_i is the value of item i
- w_i is the weight of item i
- x_i is a binary variable (0 or 1) indicating whether item i is included

- W_{max} is the maximum weight capacity of the knapsack
- α is the penalty factor coefficient (can be tuned)

When there are solutions that violate the weight constraints, our given basic sum of products fitness function either struggles or straight up fails badly as can be seen practically with the data given in the *config_2.txt* file. But this multiplicative penalty based fitness function is quite robust to that failure. It ensures that solutions violating the weight constraint are penalized, but they are not completely discarded. This allows genetic algorithms to explore potentially good solutions that might violate constraints slightly but can be adjusted through further evolution. Additionally, the penalty is proportional to the severity of the violation, enabling a balance between maximizing value and minimizing constraint violations.

The results and comparisons are provided under Q2.

Extra Credit: Early stopping function for 0-1 Knapsack Problem

Taking the inspiration from the deep neural networks training processes, where if the model doesn't improve it's score for a certain number of epochs, the training is halted (or sometimes some other action is taken), I've implemented a similar criterion, which can be found as a function named **stop_if_not_improving** (not the most creative name). It's most important argument is the patience, which waits for patience number of generation, and if the best overall score doesn't improve it halts the evolution process. But, there can be cases where an algorithm keeps making improvements and we may not have enough time or computation resources to run it for an infinite amount of time, so it also takes into account a maximum (or final) generation parameter which comes into clutch in such scenario and halts training.

The results and comparisons are provided in under Q3.

Q1: Implementation Description

The backbone of implementation is the numpy library. It implements all the required operators as outlined in the problem description. Here are some defining features about my implementation:

- It modifies the given get_initial_population function. The resulting method is called load_config. The population generation has been moved out from this method to a more relevant initialize_population method, which makes use of numpy's binomial method for sampling population. I use a probability of 0.05 for config 2 because the items have huge weights and the number of items is also quite large for the size of the constraints and hence if we keep the probability 0.5 as for config 1 (which is how the random.randint works) then our algorithm just runs into zeros and NaNs and never recovers.

- In addition to the required methods like roulette wheel selection, tournament selection, simple fitness computation method, and simple stopping criteria that runs till the specified final generation number, it offers a penalty-based fitness method and a better stopping criteria that stops the evolution if the score doesn't improve for specified number of generations.
- There are bunch of properly documented command line arguments that make the implementation quite flexible and allow user to simply change the arguments to achieve functionality like visualization (using matplotlib) of the values across generations, change number of generations, stopping criteria, selection method, fitness method, population size and much more.
- The evolve function runs the evolution based on a user specified stopping criteria, and returns a tuple containg last population as well as a dictionary containing the best overall solution.
- Additionally, although not required, I've implemented elitism in my implementation, which by default is not active, i.e. it's rate is set to 0. But you enable it, user can pass –elitism-rate argument a value in interval [0, 1].

Q2: Selection Alone

The algorithm does well despite only relying on a single operator. It moves towards better and better solutions but does eventually plateaus. It achieves a score of 4886 for config 1 with Roulette Wheel Selection and achieves the same score with Tournament selection though through different intermediate populations. For config 2, the story is same. The maximum score of 384 is achieved by both selection methods. Roulette Wheel selection keeps the algorithm searching for a new solution which is evident by fluctuations in the average score of the population at each generation, while the tournament selection converges after just 4 generations and doesn't explore different solutions afterwards. The best solution for each of these scenarios occurs in generation 0, obviously, because there's no mutation or crossover so the initial population stays the same.

Please that when sampling population for config 2, I'm using probability of 0.05 for each gene to limit the number of active genes in order to make the solutions a little feasible, as otherwise, as mentioned above, we run into zeros and NaNs.

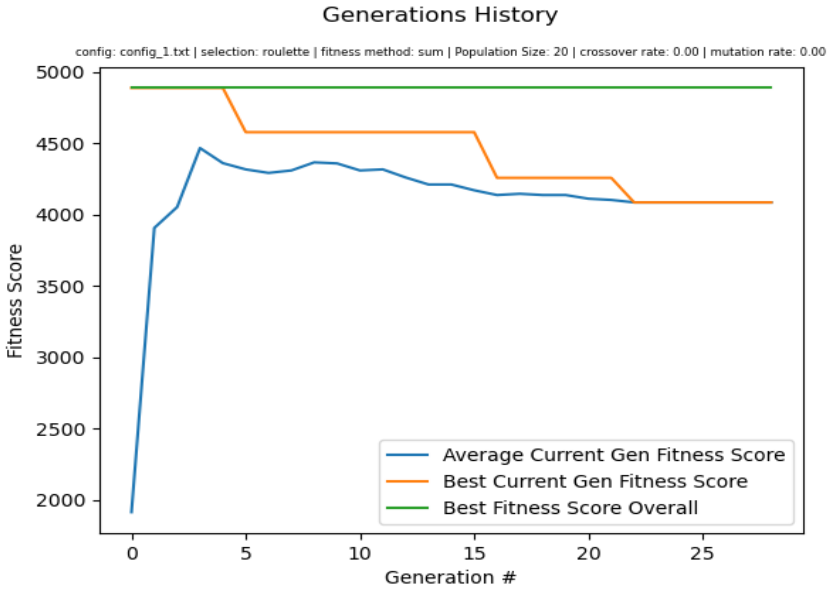
Q2: (a), (b), (c)

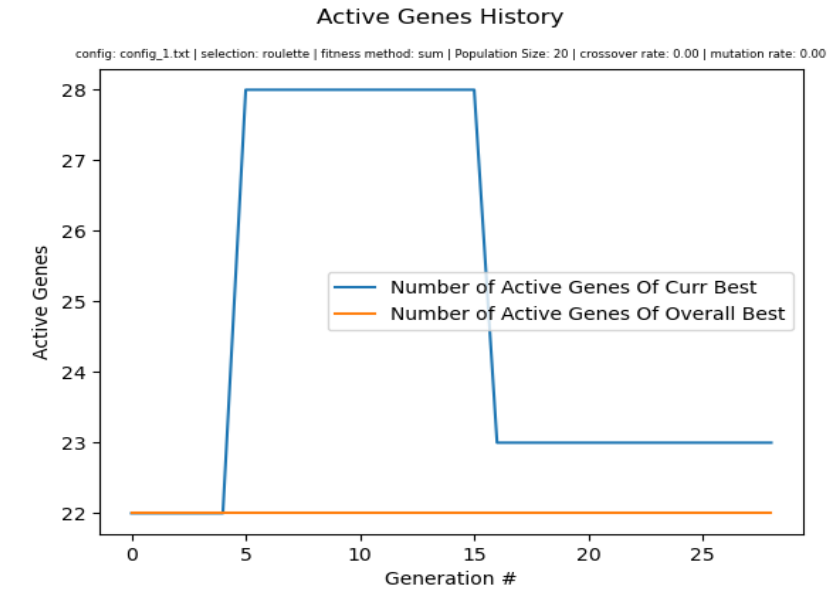
Following are some visualized plots for each configuration that provide a nice visual summary of performance of different settings for both provided configurations.

config 1:

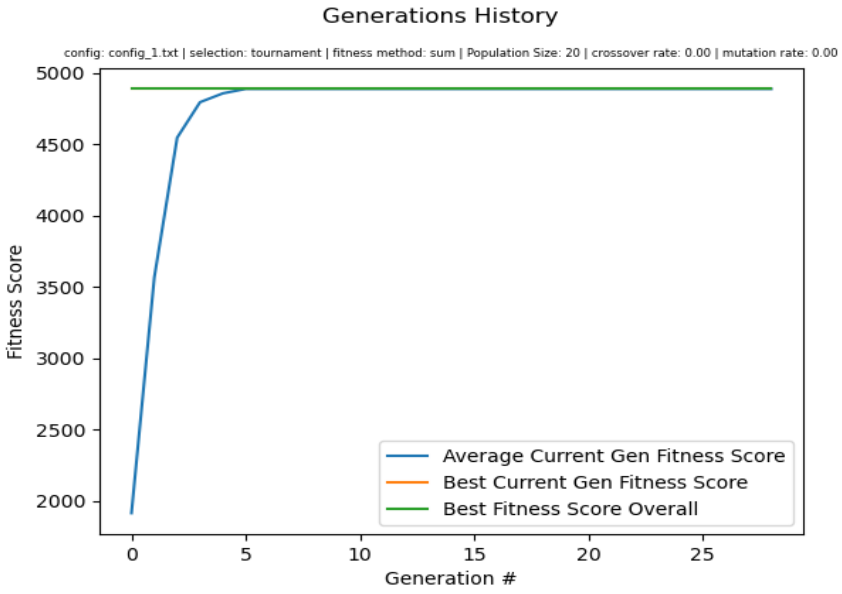
Run the following command to reproduce results:

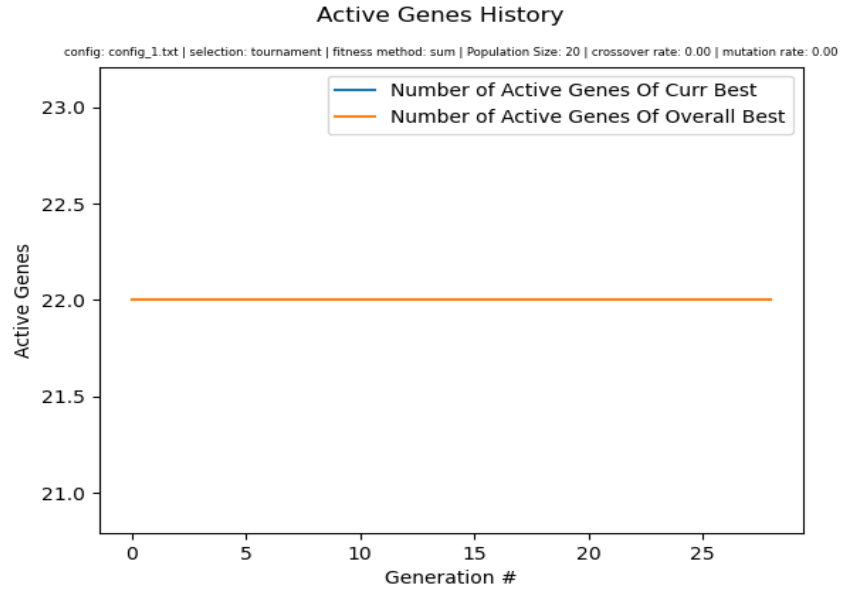
```
python ./knapsack_01_genetic.py --config config_1.txt
--visualize --selection roulette --disable-crossover
--disable-mutation
```





```
python ./knapsack_01_genetic.py --config config_1.txt
--visualize --selection tournament --disable-crossover
--disable-mutation
```

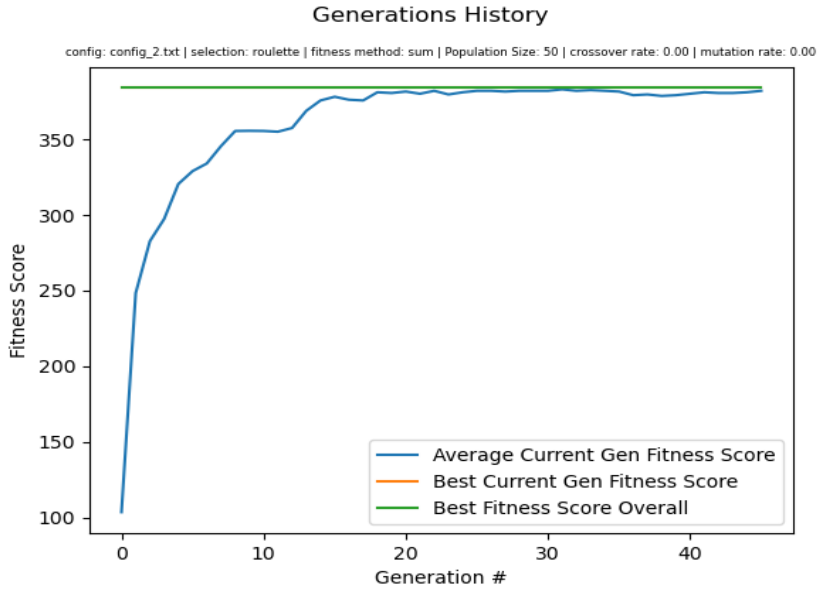


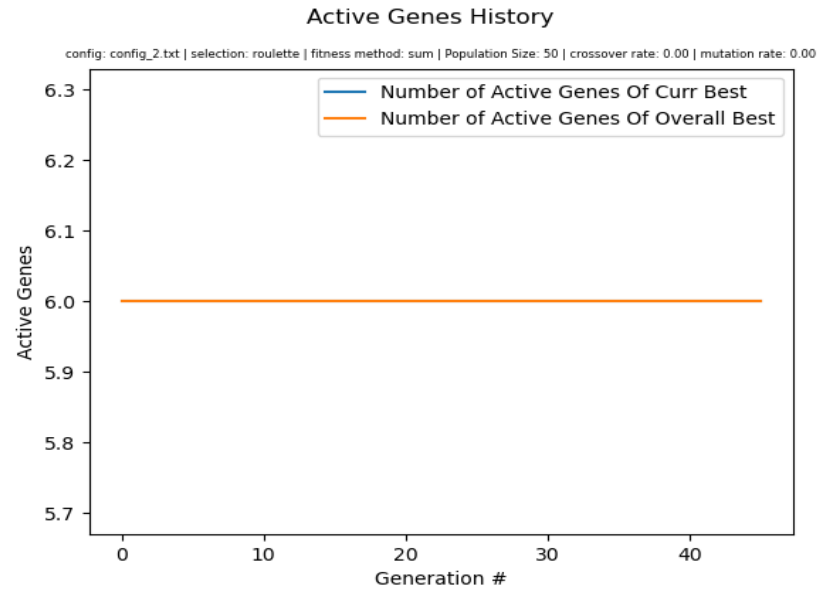


config 2:

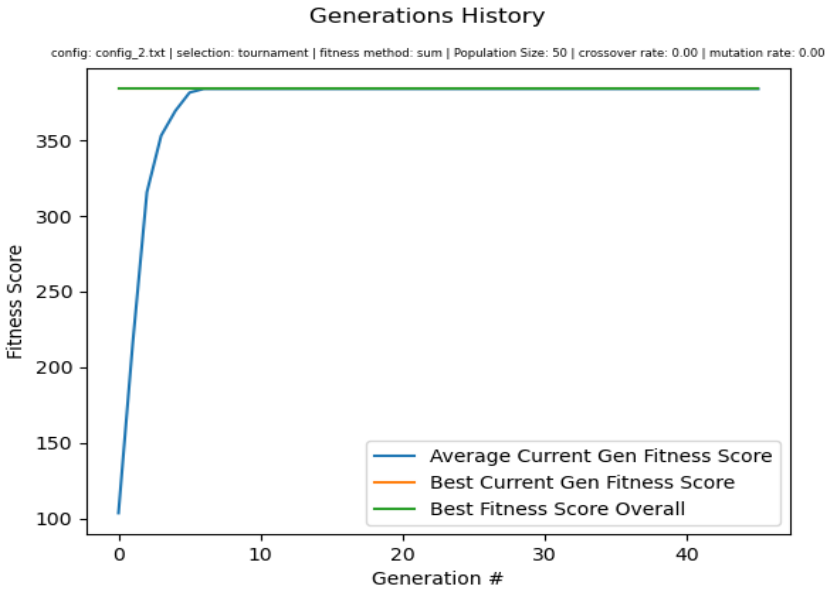
Run the following command to reproduce results:

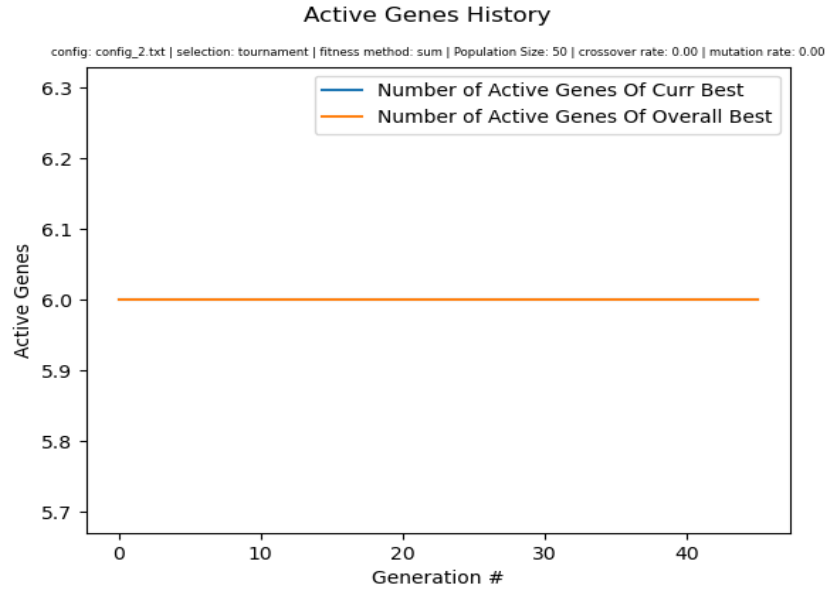
```
python ./knapsack_01_genetic.py --config config_2.txt
--visualize --selection roulette --disable-crossover
--disable-mutation
```





```
python ./knapsack_01_genetic.py --config config_2.txt
--visualize --selection tournament --disable-crossover
--disable-mutation
```





Q2: (d) What I learned:

For these particular configurations, the tournament selection is far more efficient especially if we couple it with an early stopping function as it arrives at (local or global) optimal solution, the same one that is found by roulette wheel selection in half as many steps (almost) and then it essentially never considers another solution while roulette wheel keeps on searching different solutions.

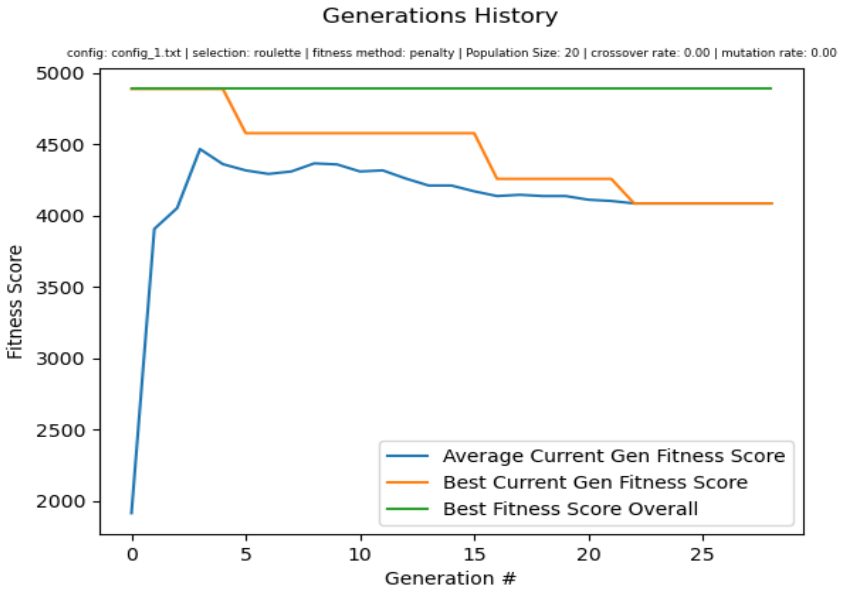
Q2 Extra Credit: Comparison with custom fitness function

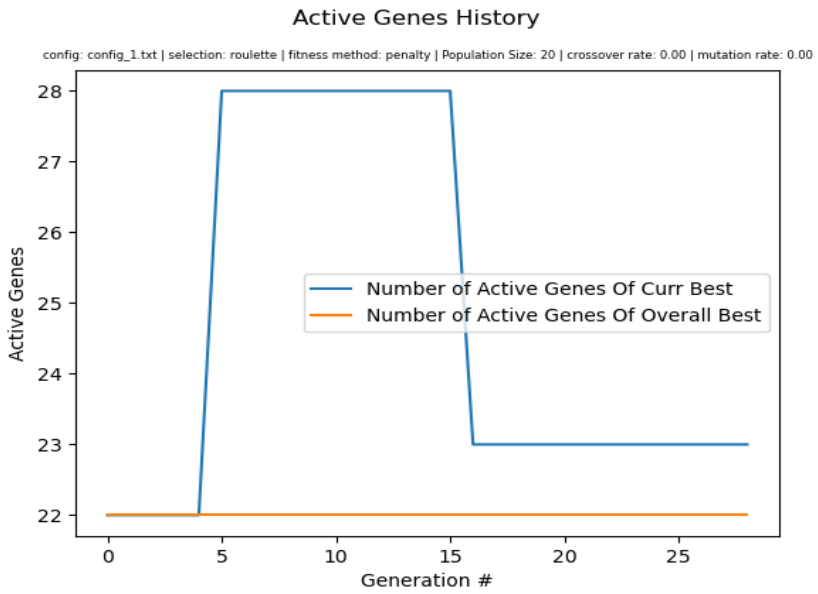
The results are identical to the simple fitness function that is given by in the assignment. I reckon it is due to the small population size and the lack of mutation and crossover operators (hence less diversity) which is the main bottleneck here that is not allowing this fitness function, that should improve results in theory at least (could perform worse as well), to utilize it's potential. I'm curious to see how it fares with the given fitness function with bigger, more diverse populations and extra genetic operators like mutation and crossover.

config 1:

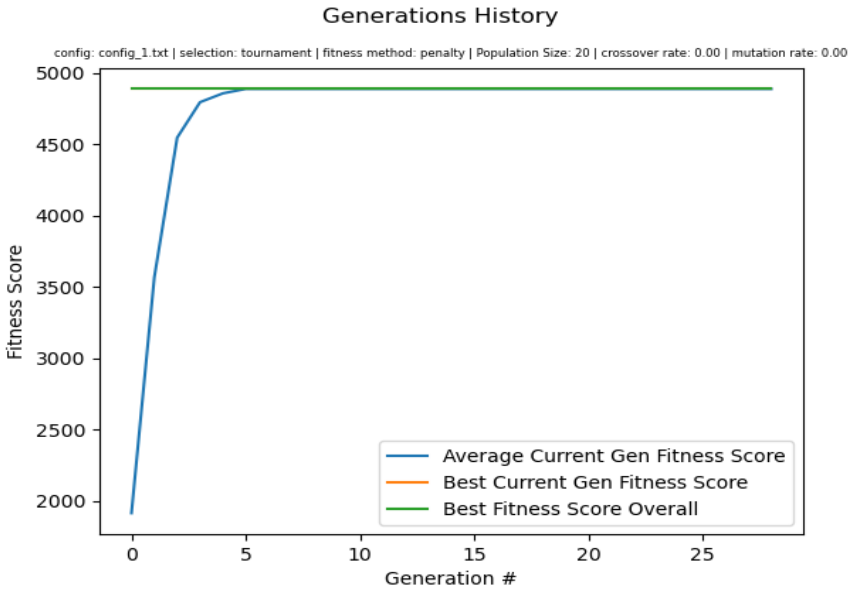
Run the following command to reproduce results:

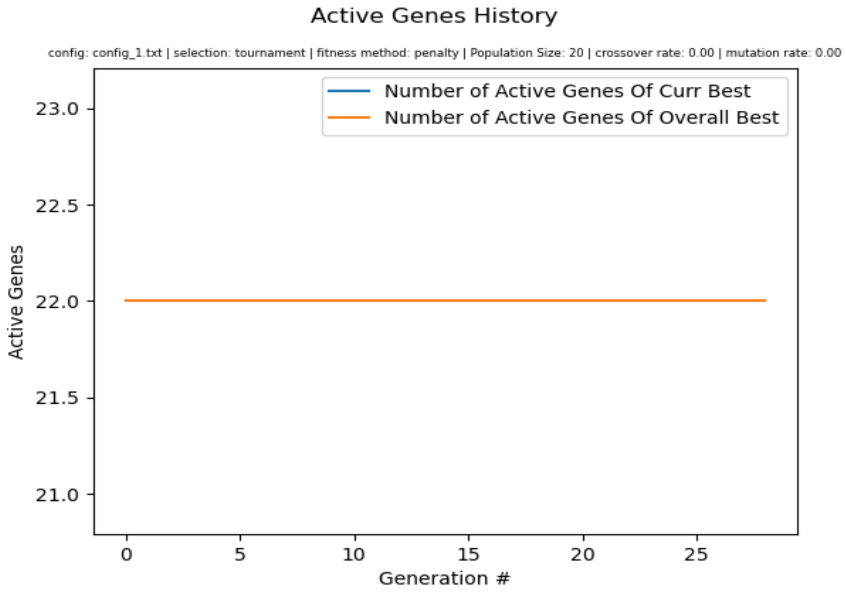
```
python ./knapsack_01_genetic.py --config config_1.txt
--visualize --selection roulette --disable-crossover
--disable-mutation --fitness penalty
```






```
python ./knapsack_01_genetic.py --config config_1.txt
--visualize --selection tournament --disable-crossover
--disable-mutation --fitness penalty
```



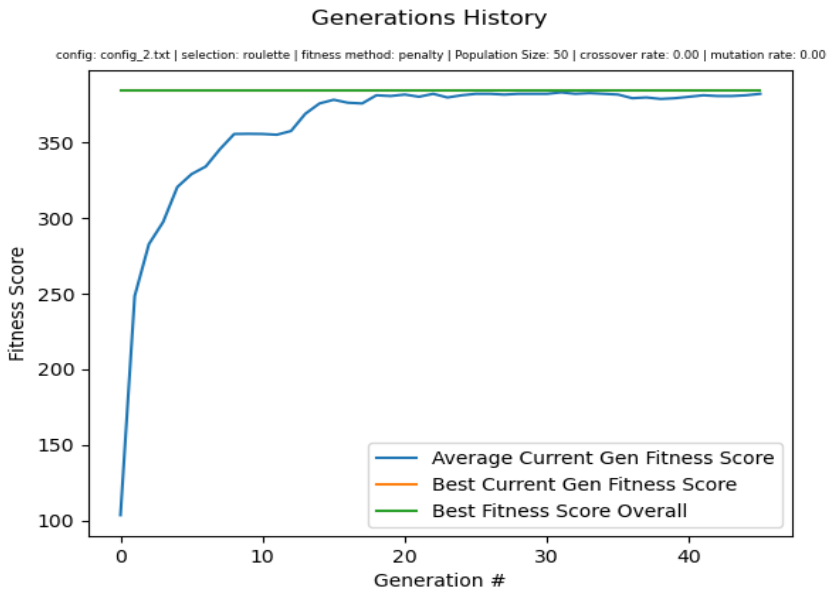


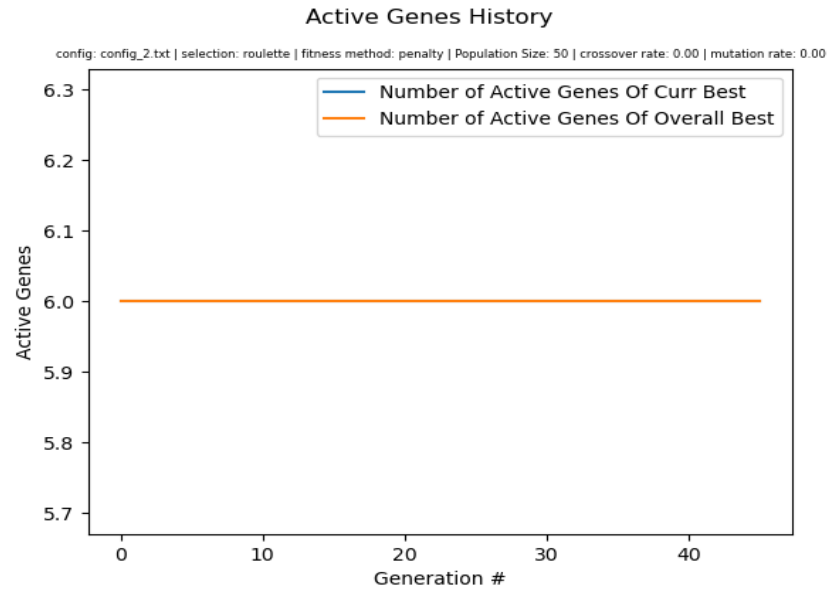
please see the next page

config 2:

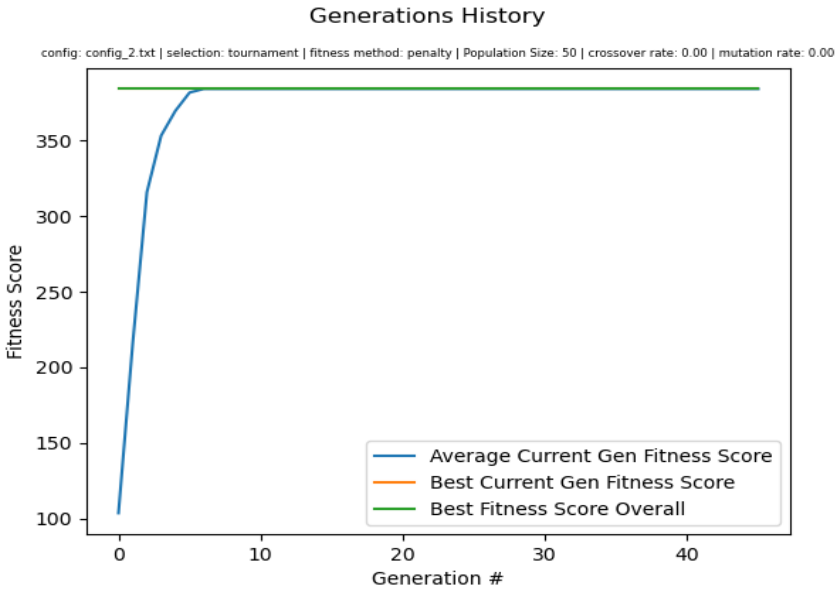
Run the following command to reproduce results:

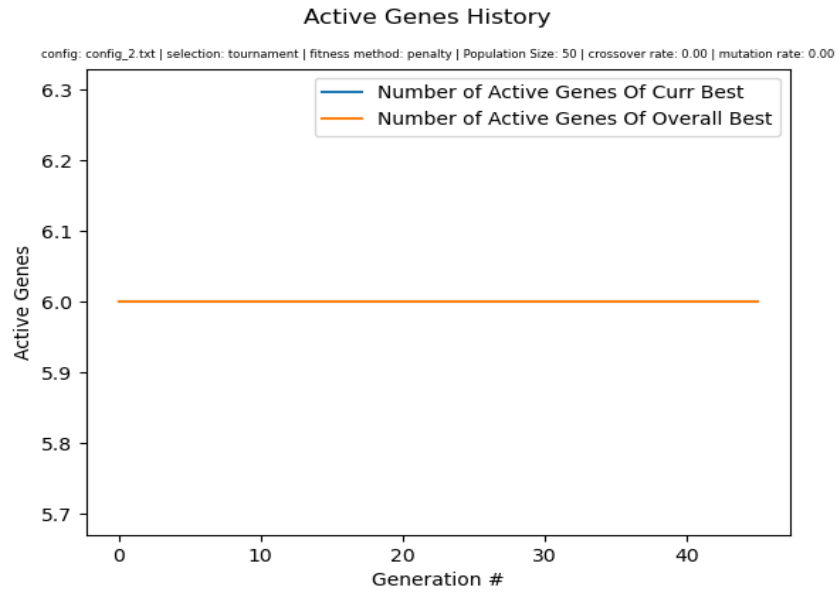
```
python ./knapsack_01_genetic.py --config config_2.txt
--visualize --selection roulette --disable-crossover
--disable-mutation --fitness penalty
```





```
python ./knapsack_01_genetic.py --config config_2.txt
--visualize --selection tournament --disable-crossover
--disable-mutation --fitness penalty
```





Q3: Integrate Cross-over and Mutation

The performance of algorithm improves by a lot as a result of integrating these two operators. For instance, for config 1 with everything else the same, and selection method of roulette wheel, the algorithm now finds a solution with a score of 6095 with crossover rate 1.0 and dynamic mutation rate which increases from 0.05 to 0.15 with 0.05 increments as compared to Q2 where the the best solution only scored 4886. Similarly, the for config 2, for roulette wheel selection, the best overall score now jumps from 384 to 750, almost double. Though, for tournament selection, the story is a little different. With a tournament size of 3, the best overall score with these two new operators enabled stays the same, at 384, but when we increase the tournament size to 6, it jumps to 538, though if we increase the size more the score starts to drop again.

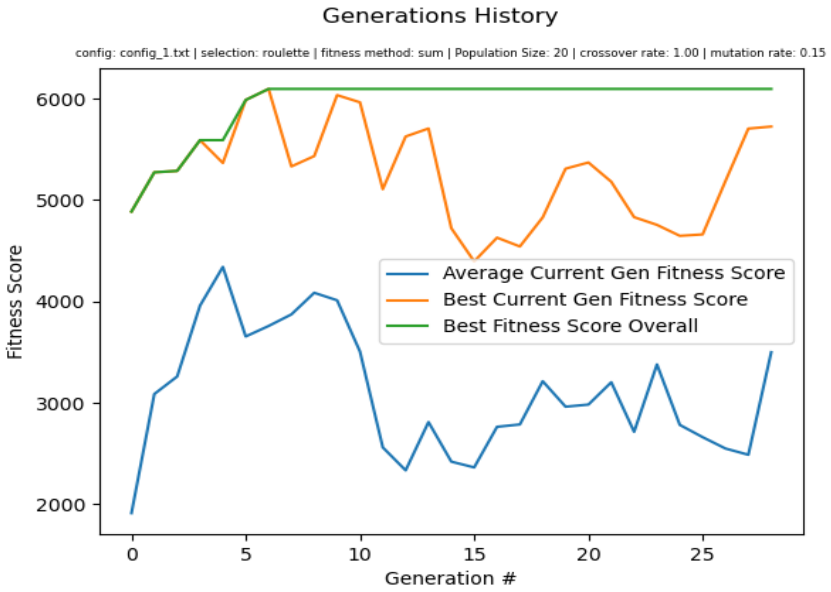
Please once again note that when sampling population for config 2, I'm using probability of 0.05 for each gene to limit the number of active genes in order to make the solutions a little feasible, as otherwise, as mentioned above, we run into zeros and NaNs.

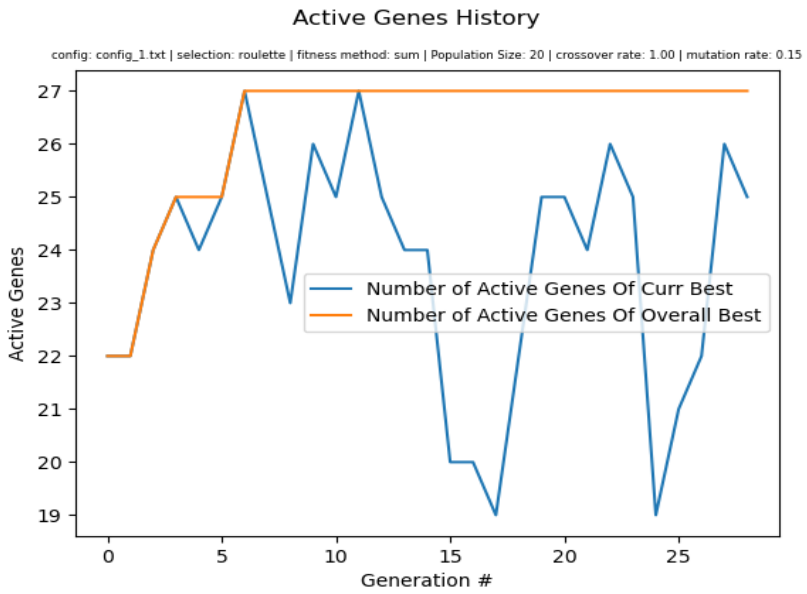
The following plots provide a visual look into the performance improvement.

config 1:

Run the following command to reproduce results:

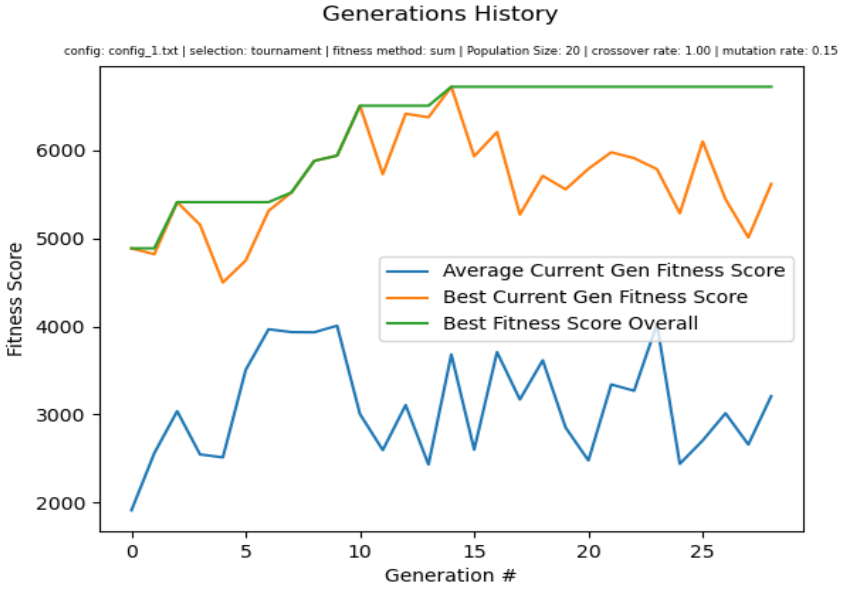
```
python ./knapsack_01_genetic.py --config config_1.txt
--visualize --selection roulette
```

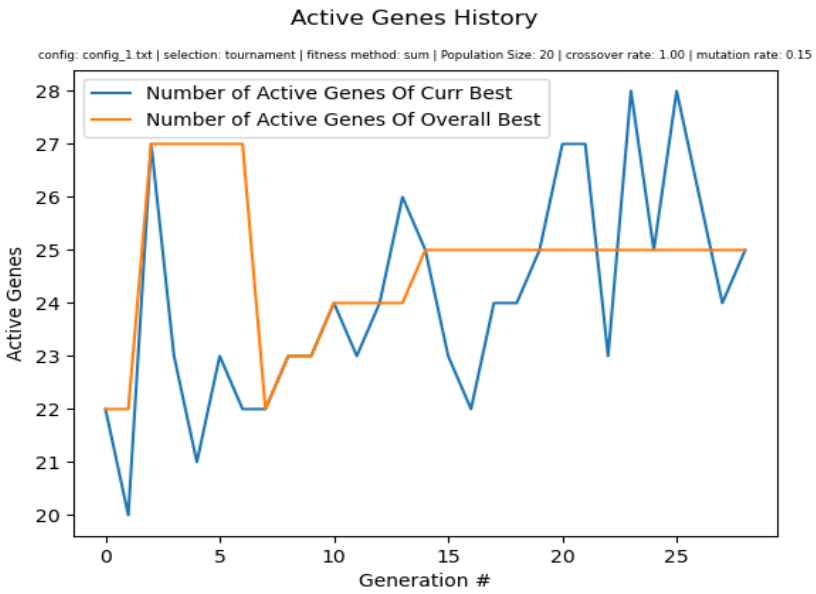




The best overall solution with value 6095 occurs in generation 6.

```
python ./knapsack_01_genetic.py --config config_1.txt
--visualize --selection tournament
```





The best overall solution with value 6719 occurs in generation 14.

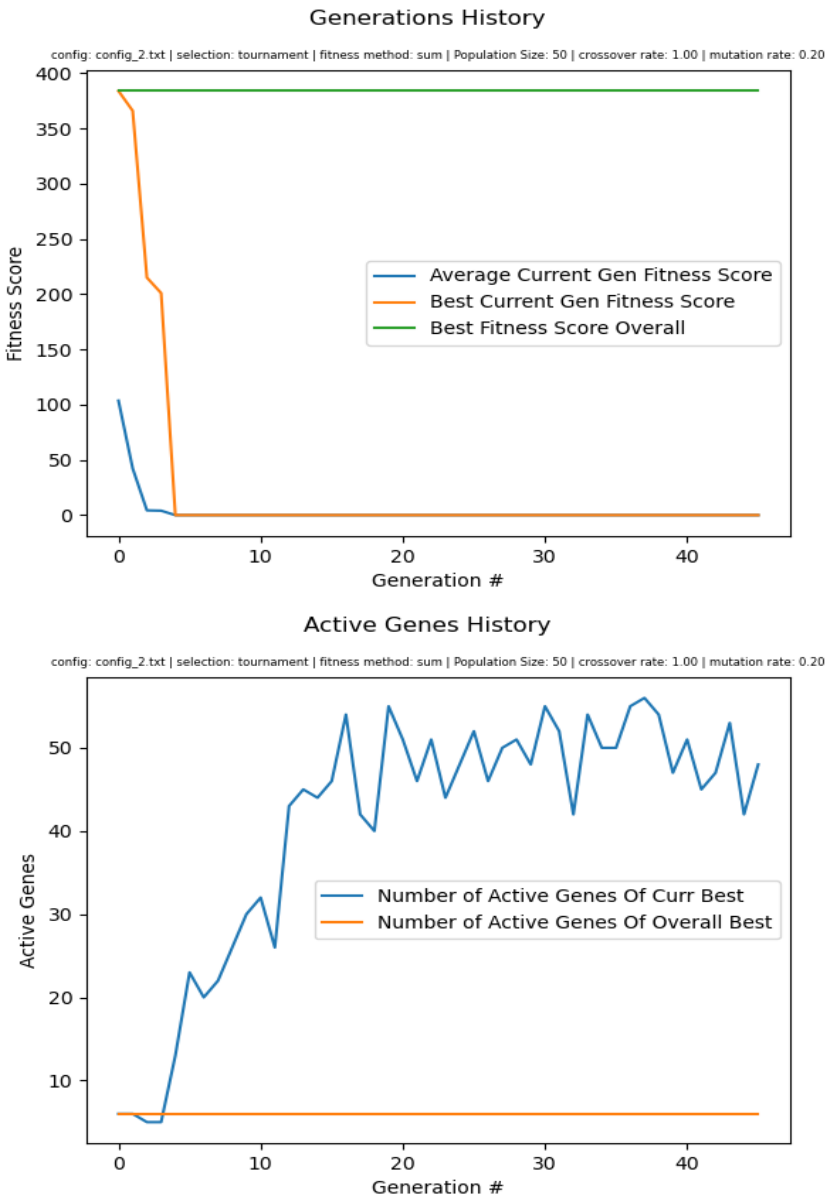
config 2:

Run the following command to reproduce results:

```
python ./knapsack_01_genetic.py --config config_2.txt
--visualize --selection roulette
```

The best overall solution with value of 750 occurs in generation 8.

```
python ./knapsack_01_genetic.py --config config_2.txt
--visualize --selection tournament
```



The best solution with value 384 occurs in generation 0 with tournament size of 3, and occurs in generation 3 with tournament size of 6 with a value of 538. So there's quite some room for improvement. The results shown are for tournament size 3.

Q3 Extra Credit: Custom Stopping Criteria

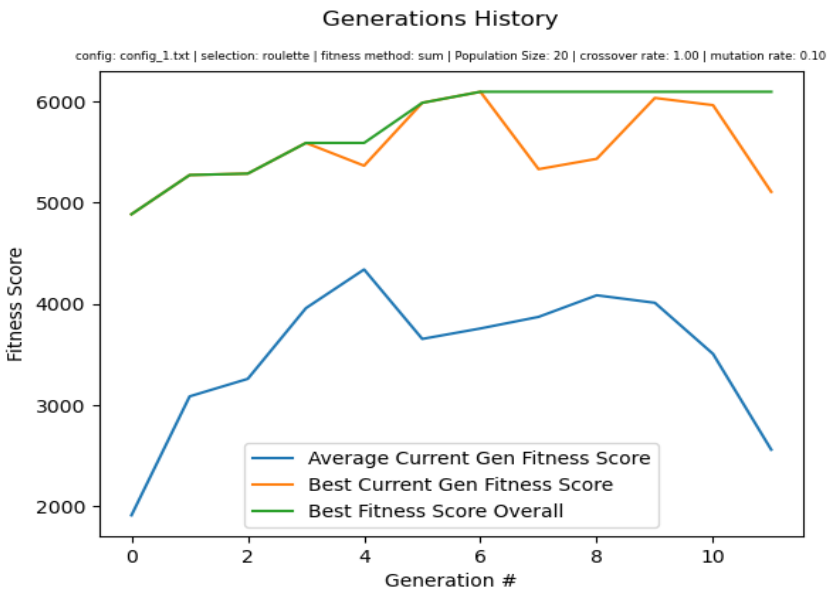
The stopping criteria I've implemented is just early stopping found in neural networks training. I've explain it above. The results here aren't groundbreaking in terms of score (one wouldn't expect it either), it just saves us computation resources by terminating early when the algorithm isn't improving.

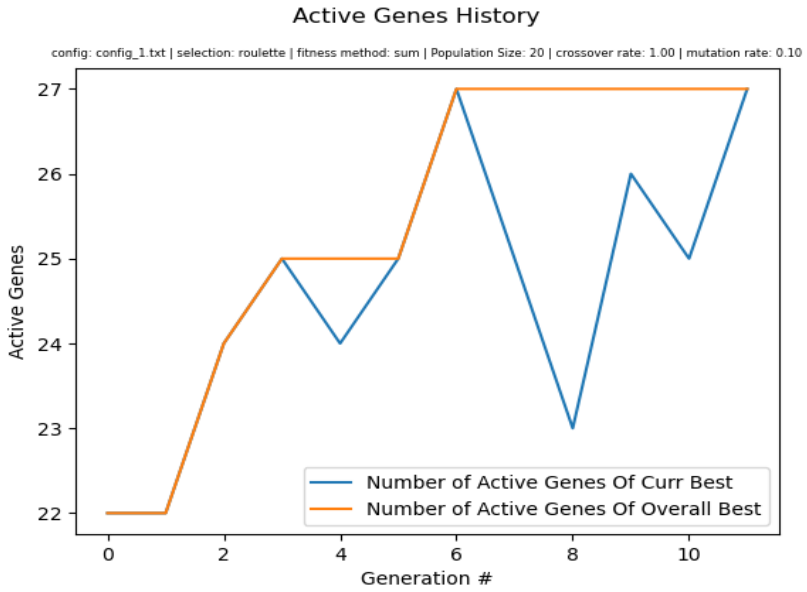
I'm attaching the visualizations below which show basically the same results but now the number of generations are drastically reduced.

config 1:

Run the following command to reproduce results:

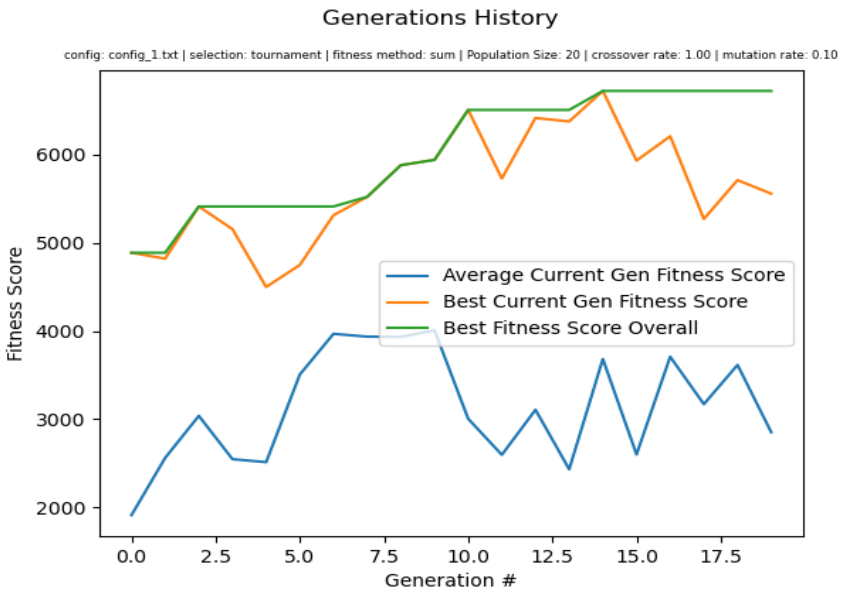
```
python ./knapsack_01_genetic.py --config config_1.txt
--visualize --selection roulette --stop notimproving
```

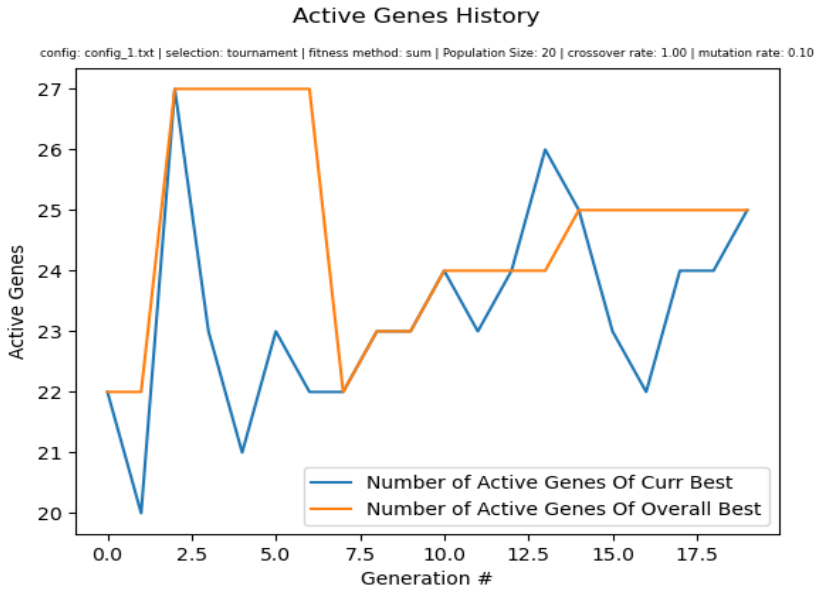




The best overall solution with value 6095 occurs in generation 6, so the evolution stops at generation 11 after waiting for 5 generation (i.e. patience).

```
python ./knapsack_01_genetic.py --config config_1.txt
--visualize --selection tournament --stop notimproving
```



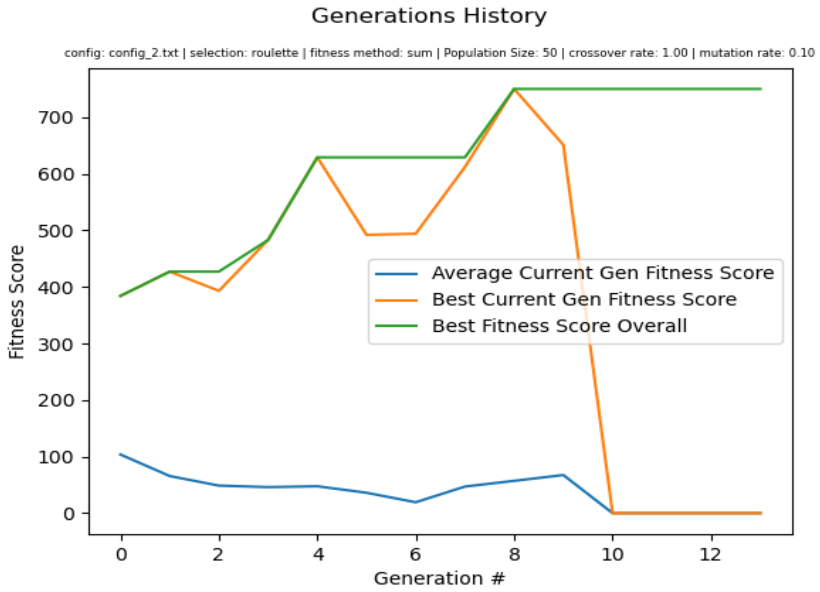


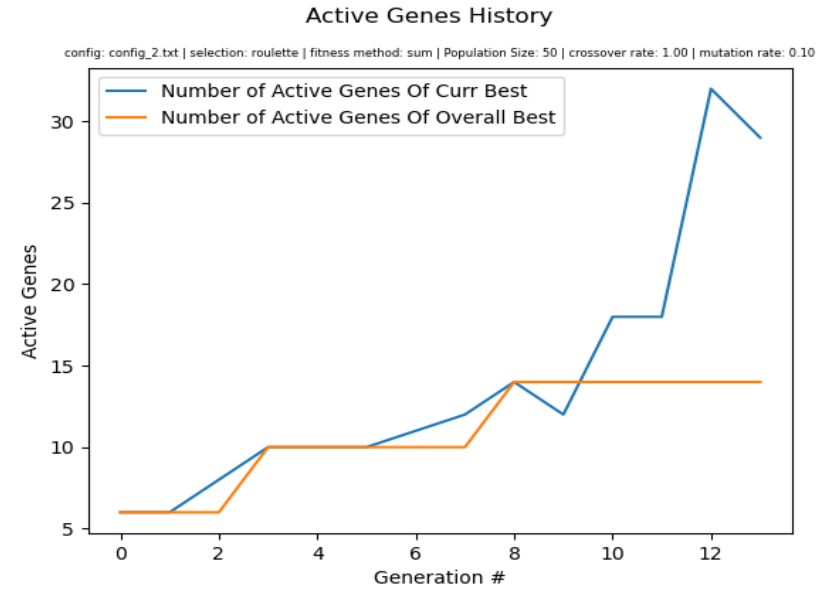
The best overall solution with value 6719 occurs in generation 14 so the evolution stops at generation 19 after waiting for 5 generation (i.e. patience).

config 2:

Run the following command to reproduce results:

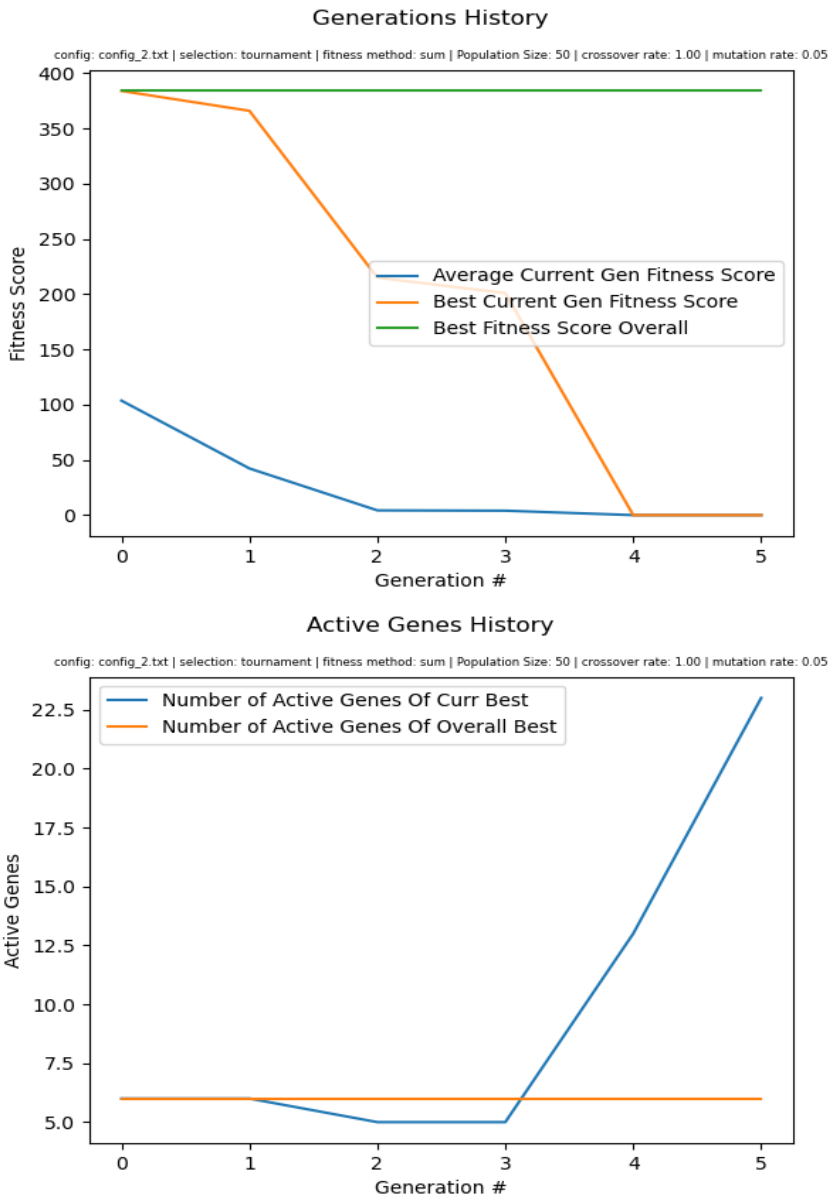
```
python ./knapsack_01_genetic.py --config config_2.txt
--visualize --selection roulette --stop notimproving
```





The best solution occurs in generation 8 hence evolution stops at gen 13.

```
python ./knapsack_01_genetic.py --config config_2.txt
--visualize --selection tournament --stop notimproving
```



The best overall solution with value occurs in generation 0 hence evolution stops at generation 5.

Q4: Explore Population Sizes

As we increase the population size, we notice an expected trend - the algorithm is being able to find better solutions thanks to more diversity in terms of individuals.

Population_Size	Total_Value $\overline{u} \pm \sigma$	Max_Weight	N_Items_in_Best
50	5549.60 ± 298.41	846	24
100	5678.07 ± 328.91	848	25
150	5718.57 ± 317.42	850	25
200	5767.99 ± 330.19	850	25
250	5794.53 ± 325.93	850	25
300	5825.11 ± 320.47	850	25
350	5858.50 ± 321.25	850	25
400	5874.96 ± 315.66	850	25
450	5898.46 ± 310.22	850	25
500	5913.21 ± 306.97	850	25
550	5930.59 ± 303.16	850	25
600	5945.77 ± 299.37	850	25
650	5966.15 ± 302.97	850	25
700	5976.27 ± 297.01	850	25
750	5988.22 ± 293.26	850	25
800	5999.96 ± 292.12	850	25
850	6008.88 ± 289.15	850	25
900	6022.39 ± 290.46	850	25
950	6034.52 ± 292.62	850	25
1000	6044.24 ± 291.58	850	25

Problem 2: Compare GA with a Non-population-based Search Algorithm

For this problem, I've implemented Tabu Search, which although relies on neighbors, but isn't considered a population based algorithm as at any given time we only consider a single current solution. It was also quite fun to implement and experiment with.

The code can be found in the accompanying **knapsack_01_tabu.py** file.
Following are some important features about my implementation:

- It uses the simple sum of included items values as a criteria to compare solutions, the same as described in the assignment but without the constraint check which is assumed to be handled by the user downstream.
- It allows user to use either a random initial solution or a greedy initial solution. The greedy initial solution uses value to weight ratio to choose the

initial (feasible) solution. Compared to genetic algorithm, which initializes a whole population, it only initializes a single solution to begin.

- It selects all the feasible neighbors of a given solution and finds the best that is not in tabu list. The only exception to this rule is when there is a solution that is better than the overall best solution found so far. It is very different from the crossover operation in genetic algorithm in that we don't have a pair, and we don't really want to create new children, we just explore the "neighbourhood" around the current solution by flipping one bit at a time.
- The tabu_search function encapsulates the core algorithm of Tabu Search.
- Similar to Genetic algorithm based implementation, command line arguments are available for flexibility.

Q5: Empirical Comparison

For the sake of comparison, I ran tabu search for the same number of iterations as number of generations given in each config.

The results were quite surprising for me.

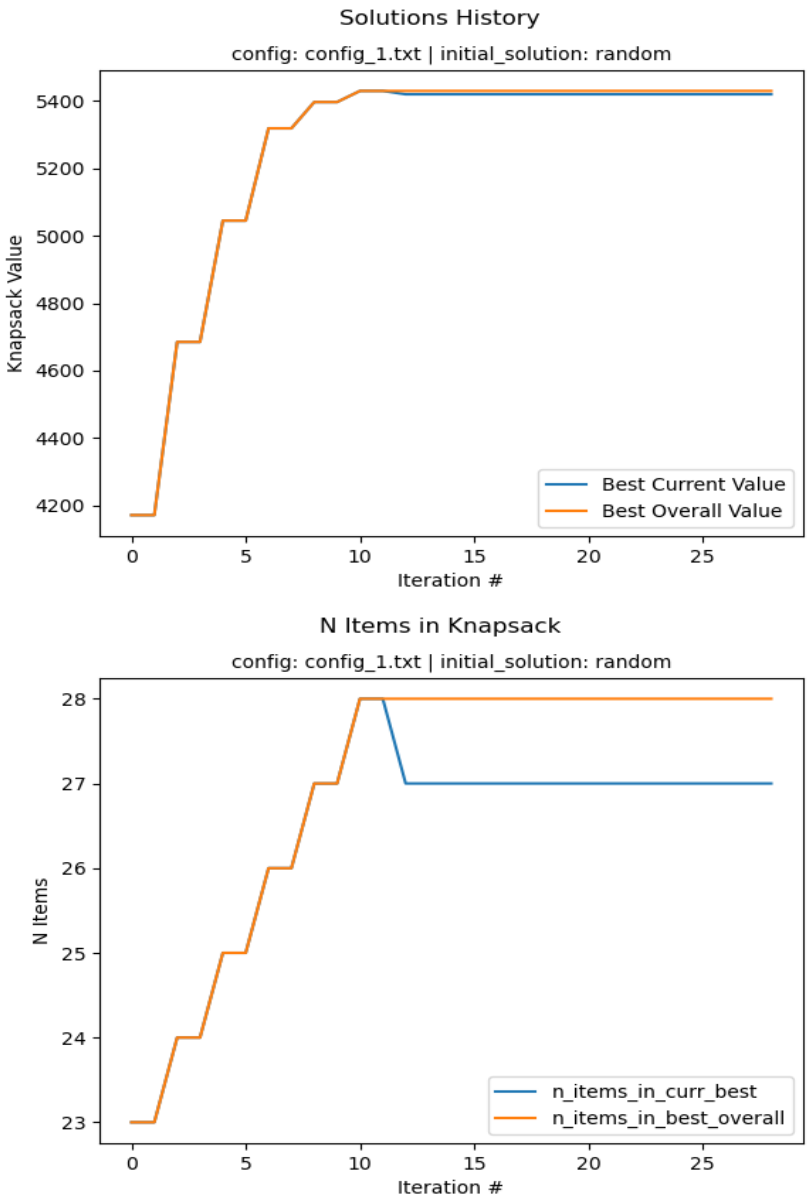
Tabu search performed relatively poorly with random initial solution but performed great with greedy initial solution.

Let's see it in action for each config where i also compare numbers with genetic approach.

config 1:

Run the following command to reproduce results:

```
python ./knapsack_01_tabu.py --verbose --visualize --initial-solution random
```



Tabu search finds a best solution of 5430 at iteration 10, compared to the best genetic setting which found a solution with 6719 score at generation 14. So here

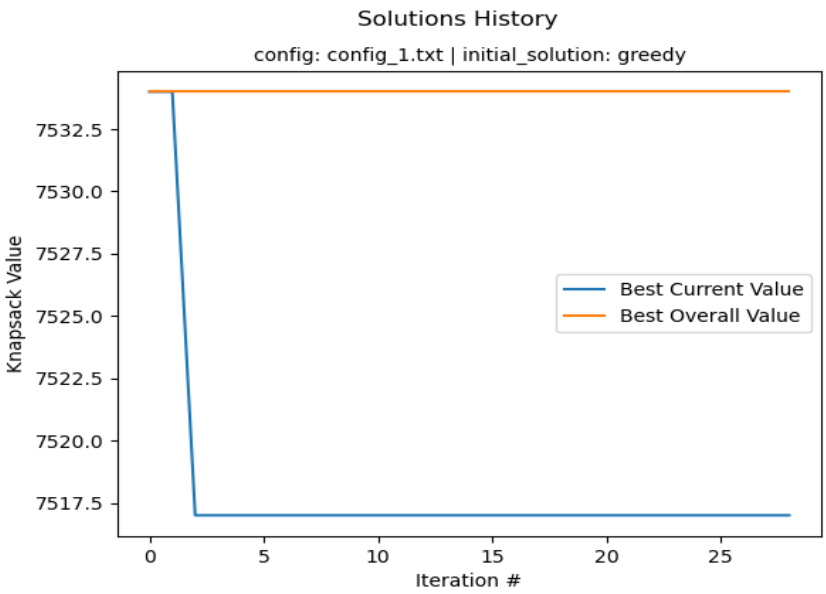
it does worse.

One interesting thing to notice is that, tabu search incorporates more items 28, compared to genetic counterpart which included 25 items only.

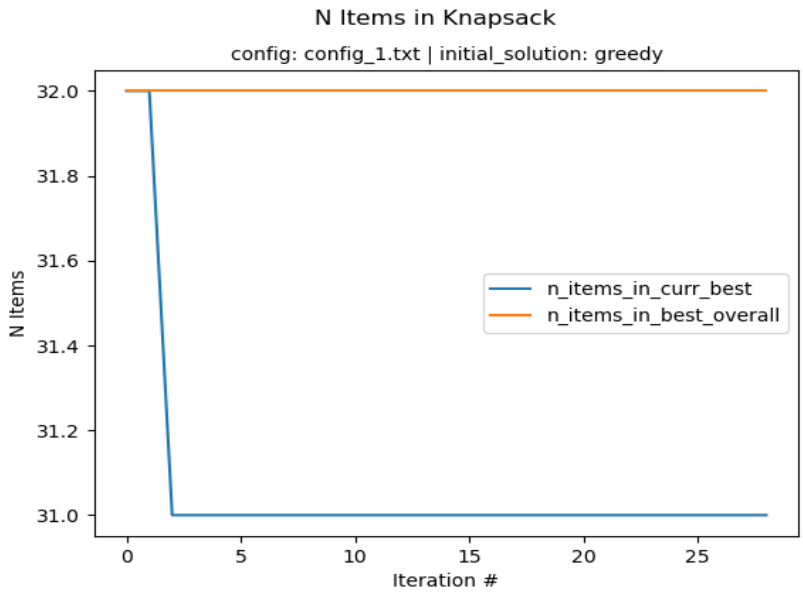
But let's see what happens if we choose a greedy initial solution instead of a random.

Run the following command to reproduce results:

```
python ./knapsack_01_tabu.py --verbose --visualize --initial-solution greedy
```



Tabu search now finds a much better solution, 7534 compared to 6719, that too at just second iteration compared to 14 by the genetic approach, and then it plateaus. We could've used early stopping here to save computation as well. Anyway, it is really impressive that with the help of a greedy initialization we can arrive at a much better solution so quickly.



And now it also includes more items than the genetic counterpart, 32 vs 28.

The story is same for config 2, therefore for the sake of saving space i won't include the results here but those can be found by the running the above commands.

Problem 3: Genetic Algorithm Formulation

Q6: Problem Description and GA Design

In machine learning, model performance heavily relies on the choice of hyperparameters. Choosing the correct mix of hyperparameters can greatly affect a model's precision, capacity to generalize, and performance. Conventional methods to tune hyperparameters, like GridSearch and RandomSearch, usually consume a lot of computational resouces, but genetic algorithms promise a much better alternative.

Chromosome Representation

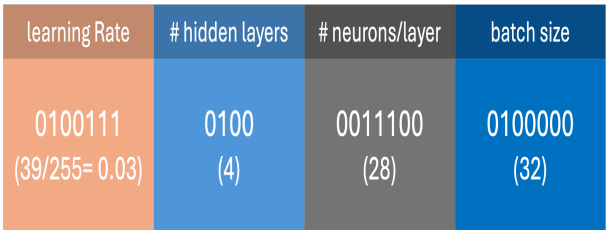
In GA, a chromosome is typically a candidate solution, and in hyperparameter tuning, each chromosome would represent a set of hyperparameters for the neural network. For the sake this question, let's consider following 4 hyperamaeters.

- Gene 1: Learning rate (Encode using 8 bits)
- Gene 2: Batch size (Encode using 7 bits)
- Gene 3: Number of hidden layers (Encode using 4 bits)

- Gene 4: Number of neurons per layer (Encode using 7 bits)

The range is determined by the number of bits, as well as any normalization post decimal conversion. Since learning rate is usually a small float value, so in practice we'll convert it to decimal and then normalize. For instance, if we want it in [0, 1] interval, we can divide it by 255.

Here's a visual example.



Population Initialization

The population represents a group of chromosomes (candidate solutions). We could initialize the population in the following ways:

- Random Initialization: Each chromosome is generated randomly by sampling from the possible range of hyperparameter values.
- Heuristic-based Initialization: Some prior knowledge of good hyperparameter values (based on experience or other methods) could be used to initialize part of the population closer to a good solution, while maintaining randomness for exploration. For instance, if I know that a learning rate around 0.01 works well for a given task, then some chromosomes could be initialized with learning rates close to this value.

Stopping Criteria

As stopping criteria determines when the algorithm should stop evolving the population. For our scenario, we can use any of the following depending on our goals and computataion resources:

- Max iterations: Stop after a fixed number of iterations.
- Early Stopping: Stop when the fitness function does not improve over a certain number of iterations.
- Threshold: Stop if we achieve a fitness above a certain threshold (e.g., accuracy of 95

Fitness Function

Since a fitness function evaluates how well a particular chromosome (i.e., set of hyperparameters) performs. In our case of neural networks, the fitness function would involve training the model with the hyperparameters encoded by the chromosome and evaluating its performance. One common fitness function could be the validation accuracy or the validation loss of the network after training.

For a simple example let's say we use validation accuracy, then,
fitness = validation accuracy = $\frac{\text{correct predictions}}{\text{total predictions}}$

Problem 4: Exploring Another Technique

Q7 Exploring Another Nature-inspired Search/Optimization Technique

For this section, I've chosen the Artificial Fish Swarm Algorithm from Dan Simon's Evolutionary Optimization Algorithms book's Chapter 17, section 2 [1].

Artificial Fish Swarm Algorithm

This algorithm is loosely based on the swarming behavior of fish. We keep track of each fish's location, and we also define a tunable visual range for every fish, beyond which they cannot see. For each fish we also keep track of all the fishes within its visual range based on the locations previously mentioned. In this algorithm, fish can behave either randomly, when there's no other fish within their range or when the best individual stops improving, the fish can also exhibit leaping behavior in this case, or they can chase other fish in their range which are closer to the highest food concentration. Fish can also have swarming behavior where it moves to the center of a fish group in its range. Under search behavior, a fish moves towards a fish with more food. Since all these behaviors are greedy, so AFSA is a greedy algorithm which then has the benefits and disadvantages of a greedy algorithm.

References

[1] Dan Simon (2013) *Evolutionary Optimization Algorithms, Page 423-426*, Wiley.