CS551.V

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

ÖZYEĞİN UNIVERSITY

ASSIGNMENT-2

SOLVING THE KNAPSACK PROBLEM USING GENETIC ALGORITHM

Atalay PABUŞÇU

026217

1. Introduction

In this study, we worked on finding the best solution for a Knapsack problem using genetic algorithm. The Knapsack Problem is an optimization problem, that aims to maximize the benefit of objects in a knapsack without exceeding its capacity [11]. We exploited the genetic algorithm in order to extract the best fitted solution for the given dataset. Our program takes the weight limit value from the user and solves the Knapsack problem for this limit. It generates population over again until the generation loop reaches the total generation number. Finally, the program plots maximum and average fitness scores according to generations and gives a solution output that includes "population size", "generation number", "Maximum Fitness score", "Maximum Average Fitness score" and, "Best fitting individual (solution)".

1.1. Knapsack Problem

The Knapsack problem is the problem that items with different values and weights are stuffed into a backpack, in a way that provides the highest possible value within the maximum weight limit. A lot of algorithms have been developed to solve this problem. These algorithms have been the solution to many problems such as capital budgeting allocation decisions, project selection, finding the least wasteful way to cut raw materials…

The objective function is to maximize the total value of items selected while the constraint is to ensure the total weight is no more than the weight limitation. We can represent the values and weights with two integer arrays consisting of N elements. Let's assume W is the knapsack capacity, N is the total item number and "val[]" is the subset which includes items. None of items can be broken, just can be picked completely.

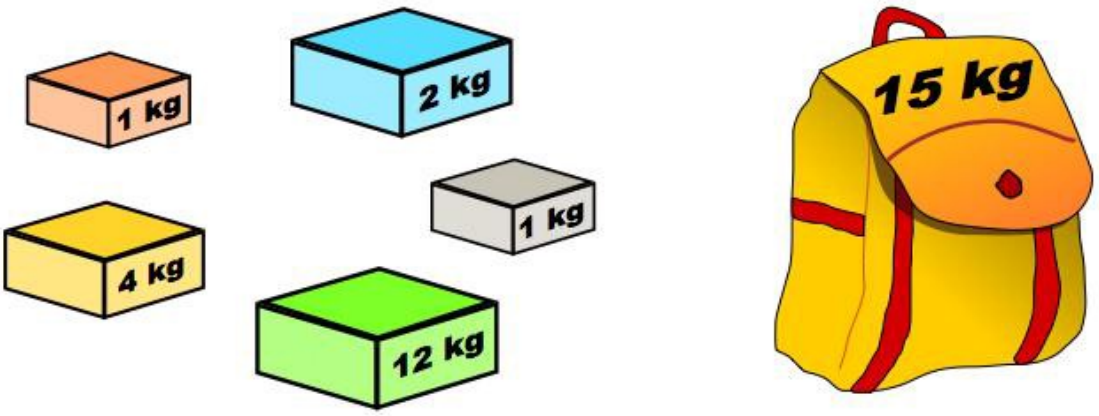

Fig.1. Knapsack problem demonstration. [7]

We will find out the maximum value subset of "val[]" such that the sum of the weights of this subset is smaller than or equal to W. We will consider all subsets of items and calculate the total weight and value of all subsets.

1.2. Genetic Algorithms

Genetic algorithms are adaptive heuristic search algorithms that are inspired by Charles Darwin's theory of natural evolution (natural selection and genetics). It reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. They belong to the larger part of evolutionary algorithms.

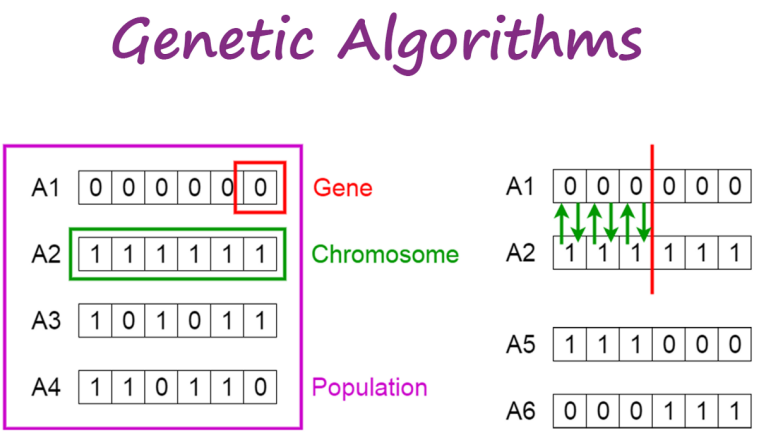

Fig.2. Genetic Algorithms demonstration. [5]

2. Methodology

In order to solve the Knapsack problem, there are a lot of methods in the literature. In this work, we exploited genetic algorithms to solve this problem. Genetic algorithms can be used to generate solutions for problems which we can not calculate the possible solutions. The Knapsack data example is shown below.

	Values	Weights
0	60	10
1	100	20
2	120	30

"Knapsack.xlsx" file data

2.1. Natural Selection

The natural selection process starts with selection of fittest individuals from a population. These selection will produce new children which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents

and have a better chance at surviving. This process provides to create new generations with iterations until the fittest individuals are found.

2.2. Genetic Algorithm Phases

Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. Each generation consist of a population of individuals and each individual represents a point in search space and possible solution. Genetic algorithms consist of 5 different phases as “Initial population”, “Fitness function”, “Selection”, “Crossover”, and “Mutation”.

2.2.1. Initial Population

In this phase, we will create set of individuals which is called a population. Each individual provides a way to solve the given (Knapsack) problem. An individual is characterized by a set of parameters, known as ***genes***. Genes are combined as string to form a ***chromosome***. The chromosome is the representation of the desired solution. In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Then we encode the genes in chromosome.

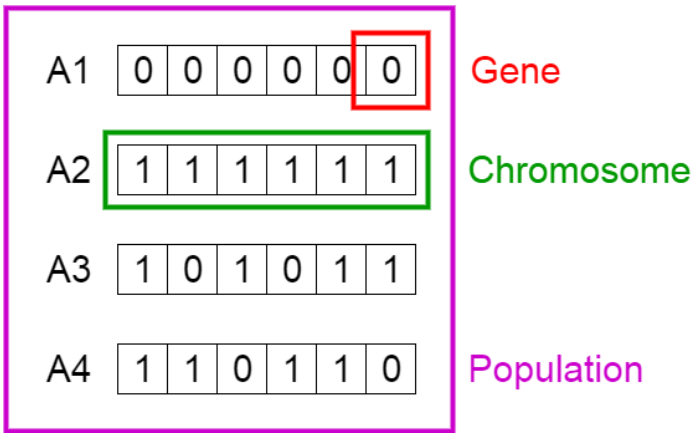

Fig.3. Create initial population [5]

2.2.2. Fitness function

We exploit the fitness function to determine how fit an individual is. It indicates the ability of an individual to compete with other individuals. This function produce an output as the fitness score for each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

2.2.3. Selection

In this phase, algorithm selects fittest individuals and lets them pass their genes to the next generation. Individuals with high fitness scores have more chance to be selected for reproduction. At the end of this phase two pairs of individuals are selected based on their fitness scores and forwarded to the next phase. The population size is not dynamic. So the space has to be created for new arrivals.

Some individuals should remove and some new individuals should be added to areas of these removed ones until all the mating opportunity of the old population is exhausted.

2.2.4. Crossover

In crossover phase, a crossover point is chosen at random from within the genes for each pair of parents to be mated. The new offsprings are created by exchanging their parents genes among themselves until the crossover point is reached. Then the new offspring are added to the population.

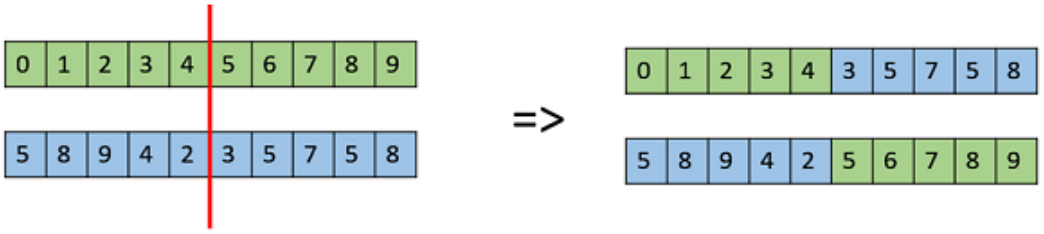

Fig.4. One-point crossover for genetic algorithm (crossover point is 5) [9]

2.2.5. Mutation

In this phase, some of new offsprings' genes which created in crossover phase can be changed randomly. Hereby, some of bits in the offsprings' bit spring will be flipped. The main idea of the mutation is maintaining the diversity in the population to avoid premature convergence.

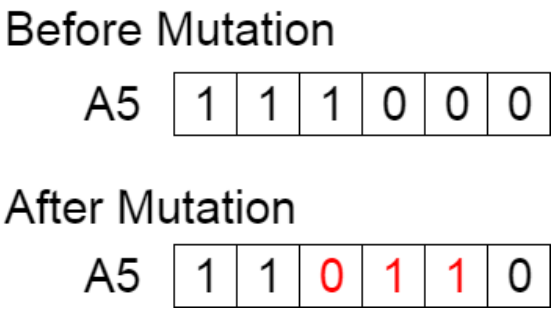

Fig.5. Mutation phase of the genetic algorithm [5]

At the end of all these phases algorithm begins to apply same processes to new individuals until the population is converged, it means the algorithm ends if any other different offspring cannot be produced, i.e. the control loop reaches the total generation number.

The pseudocode of the Genetic Algorithm that was developed for solving the Knapsack problem is given below [8].

Create randomly Population P with respect to population size.

while (generation number is reached):

P* = ∅

while (sizeof (P*) ≠ Population size):

Select p1 and p2 from P according to fitness scores

Crossover p1 and p2 to obtain h1 and h2

Mutate h1 and h2 to obtain c1 and c2

IF [Dis(p1,c1) + Dis(p2, c2)] ≤ [Dis(p1,c2) + Dis(p2, c1)]:

IF: c1 < p1 then P* = P* U {c1} ***ELSE***: P* = P* U {p1}

IF: c2 < p2 then P* = P* U {c2} ***ELSE***: P* = P* U {p2}

ELSE:

IF: c1 < p2 then P* = P* U {c1} ***ELSE***: P* = P* U {p2}

IF: c2 < p1 then P* = P* U {c2} ***ELSE***: P* = P* U {p1}

ENDWHILE

P = P*

ENDWHILE

3. Implementation

3.1. Used Knapsack Problem Datasets

In order to solve Knapsack Problem, we need to know the weight and value information for each given item. We gather weight and value information for items from a given excel file. This excel file name should be "**Knapscak.xlsx**". Our program searches the project directory to get "Knapsack.xlsx" file. If any Knapsack file is found, the program takes the weights and values as a list. In this work, we tried 2 different Knapsack problem datasets as below.

Values	Weights
60	10
100	80
120	95
90	55
80	40
95	65
70	20
110	92
40	25

100	75
58	24
75	42
14	8
82	24

Knapsack Problem Data (File-1)

Knapsack Problem Data Excel File-1 has 14 items. The maximum value of the Values list is **120** and this item's weight value is **95**. The Maximum value of the weight list is **95**. The total value of the list is **1094** and the total weight of the list is **655**.

Values	Weights
60	10
100	20
120	30
90	15
80	40
95	55
70	20
110	35
40	25
100	10

Knapsack Problem Data (File-2)

Knapsack Problem Data Excel File-2 has 10 items. The maximum value of the Values list is 120 and this item's weight value is 30. The Maximum value of the weight list is **55**. The total value of the list is **865** and the total weight of the list is **260**.

We used these datasets to find the best solution. We determined the weight limit as **550** for Knapsack File-1 and **200** for Knapsack File-2. We tried our algorithm with different population sizes as **10**, **20**, **30**, **50,** and **100** for these datasets.

3.2. Genetic Algorithm Implementation

3.2.1. Initial Population

We create the initial population using the ***number of individuals*** (chromosomes) and the ***number of genes*** (items) in an individual parameters. Initially, all genes of the individual are set

randomly as **0** or **1**. All created individuals are appended to the population list in order to be used later. The part of the code for creating the initial population is shown below.

```
def __create_population(self, num_ind, num_genes):
        ind_list = []
        ind_x = 0
        while ind_x < num_ind:
            ind = []
            for ix in range(num_genes):
                ind.append(random.randint(0, 1))
            if not ind in ind_list:
                ind_x += 1
                ind_list.append(ind)

        return ind_list
```

3.2.2. Fitness Function

We calculate the fitness function again for each individual in each generation phase. Fitness scores are stored in a list for each population element (individual) in order to be used in the selection phase. We calculate fitness scores by multiplying the item's values by genes (0, 1) and we take the sum of these values. We apply the same process to weight values. If the sum of the weight values of an individual exceeds the weight limit we set it's fitness score as 0. The part of the code for calculating the fitness scores is shown below.

```
  def __fitness_function(self):
        self.fitness_list = []    # Clear fitness_list list for new
fitness function calculation
        self.weight_list = []     # Clear weight_list list for new
fitness function calculation

        for ind_x in range(len(self.ind_list)):
            ind = self.ind_list[ind_x]
            self.fitness_list.append(0)
            self.weight_list.append(0)

            for i in range(self.items):
                self.fitness_list[ind_x] += self.values[i]*ind[i]
                self.weight_list[ind_x] += self.weights[i]*ind[i]
            # Return fitness score 0 if the weight limit is exceeded
```

```
            if self.weight_list[ind_x] > weigh_limit:
                self.fitness_list[ind_x] = 0
```

3.2.3. Selection

We select two different individuals according to fitness scores. We use two different individual pair selection methods; “best” and “tournament”. In the “best” selection method, we find the best fitting individual and the second one and return them to the algorithm in order to apply some operations such as crossover and mutation. In the “tournament” selection method, we first shuffle all individuals, then we select two individuals randomly and compare themes fitness scores and keep the best one, we apply the same process to different randomly selected individual pairs and get the best one. Finally, we return these two individuals to the algorithm.

```
    def selection(self, method='best'):
        if method == 'best':
            return self.__get_fittest_indexes()
        elif method == 'tournament':
            return self.__get_tournament_indexes()
        else:
            raise ValueError("Invalid selection method type, use
'best' or 'tournament'")
```

3.2.4. Crossover

Crossover is an evolutionary operation between two individuals, and it generates children having some parts from each parent. It takes two different individual indexes as a parameter. These parameters come from the selection phase according to the selection method. We randomly determine a crossover point. Then we cross parent individuals' bits (genes) one-by-one until the crossover point is reached. At the end of this process, we got two different children. We transfer them into the next phase of the algorithm: mutation.

```
    def crossover(self, ix1, ix2):
        cross_point = random.randint(0, (len(self.ind_list[ix1])-1))
# Radnom one-point crossover
        child1 = self.ind_list[ix1].copy()
        child2 = self.ind_list[ix2].copy()

        for ix in range(cross_point):
            child1[ix], child2[ix] = child2[ix], child1[ix]
        print("\nCross-point: ", cross_point)

        return (child1, child2)
```

3.2.5. Mutation

Mutation is an evolutionary operation that randomly mutates an individual. It takes two different individual indexes as parameters that come from the crossover result. In this phase, we randomly pick a mutation point and change (invert) the individual's bit (gene) which is at this point. We apply the same process to the second child individual. Then we give these two children to the offspring phase which generates new offspring from these chromosomes.

```
    def mutation(self, child1, child2):
        mut_point = random.randint(0, (len(child1)-1))  # Random
mutation point
        print("\nFirst mutation-point: ", mut_point)

        if child1[mut_point] == 0: child1[mut_point] = 1
        else: child1[mut_point] = 0

        mut_point = random.randint(0, (len(child1)-1))
        print("\nSecond mutation-point: ", mut_point)

        if child2[mut_point] == 0: child2[mut_point] = 1
        else: child2[mut_point] = 0

        return (child1, child2)
```

3.2.6. Offspring

We use this phase in order to contribute to the evolution of our population. Our offspring methodology relies on the finding least fitted two individuals, removing them from our population, and adding the new mutated children. Newly added children are more productive due to being produced by best fitted individuals.

```
    def offspring(self, child1, child2):
        min_ix1, min_ix2 = self.__get_least_fittest_indexes()
        # Replace least fittest individual with the fittest
offspring
        self.ind_list[min_ix1] = child1
        self.ind_list[min_ix2] = child2
```

3.2.7. Generations

This part of the code is not a phase of the genetic algorithm, but rather itself. We create new generations here, calculate fitness scores, select the best fitting individuals, apply crossover and

mutation processes to them, produce new individuals and add them to the population of the next generation. This process continues until the loop is reach to the given generation number.

```
def run_generations(self, generation, method='best', verbose=0):
        next_ind = []
        avg_fitness_scores = []
        self.history_avg_fitness = []
        self.history_max_fitness = []
        self.gen_count = 0
        max_fit = 0

        for _ in range(generation):
            self.__fitness_function()
            select_ix1, select_ix2 = self.selection(method)    #
Select best fitting individuals


self.history_avg_fitness.append(int(sum(self.fitness_list) /
len(self.fitness_list)))
            if self.fitness_list[select_ix1] > max_fit:
                max_fit = self.fitness_list[select_ix1]
            self.history_max_fitness.append(max_fit)

            if verbose > 0:
                self.__show_result_steps(select_ix1)

            child1, child2 = self.crossover(select_ix1, select_ix2)
            child1, child2 = self.mutation(child1, child2)
            # Calculate fitness function again for each individual
after crossover and mutation phases
            # self.__fitness_function()
            self.offspring(child1, child2)
            self.gen_count += 1
```

4. Working Results

We tried the genetic algorithm we developed to solve the Knapsack problem with different problem datasets which are Knapsack File-1 and Knapsack File-2. In addition, we changed the population sizes, maximum generation numbers, and individual selection methods, then we observed the changes in average fitness scores with respect to generations.

4.1. Genetic Algorithm Performance Comparison for Different Datasets

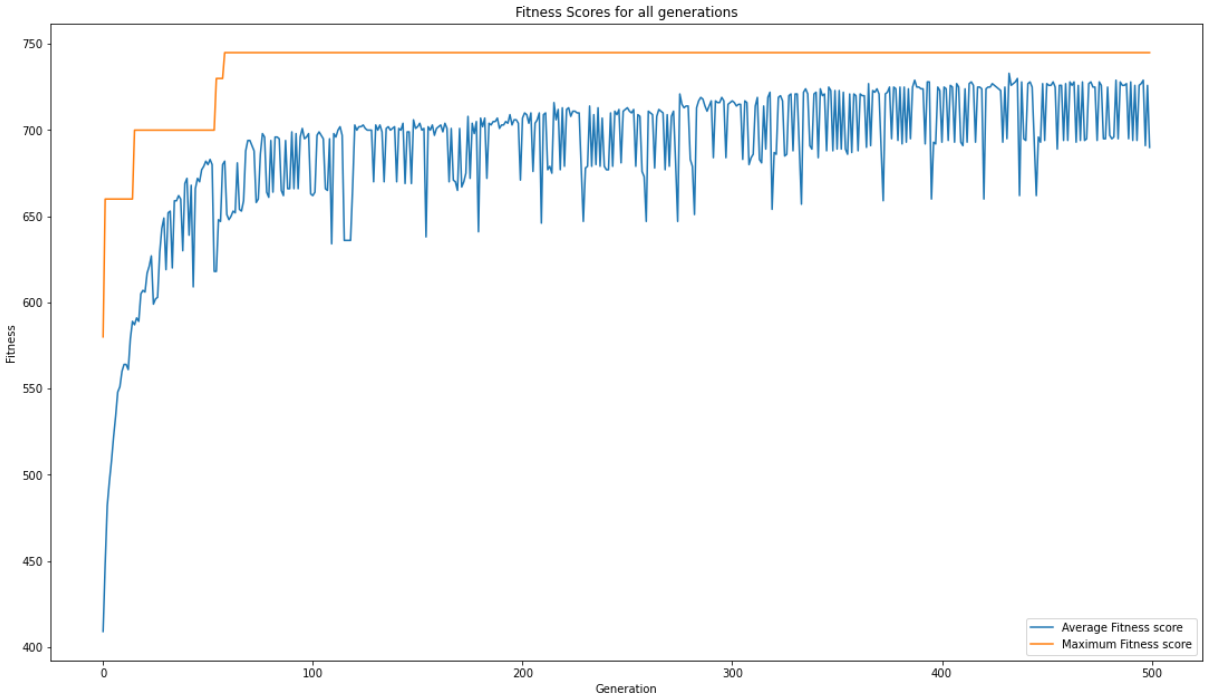

Fig.6. Average Fitness Score - Number of generations Graph (Knapsack Data-1, population size 20, maximum generations 500, weight limit 200, fitness criteria 'tournament')

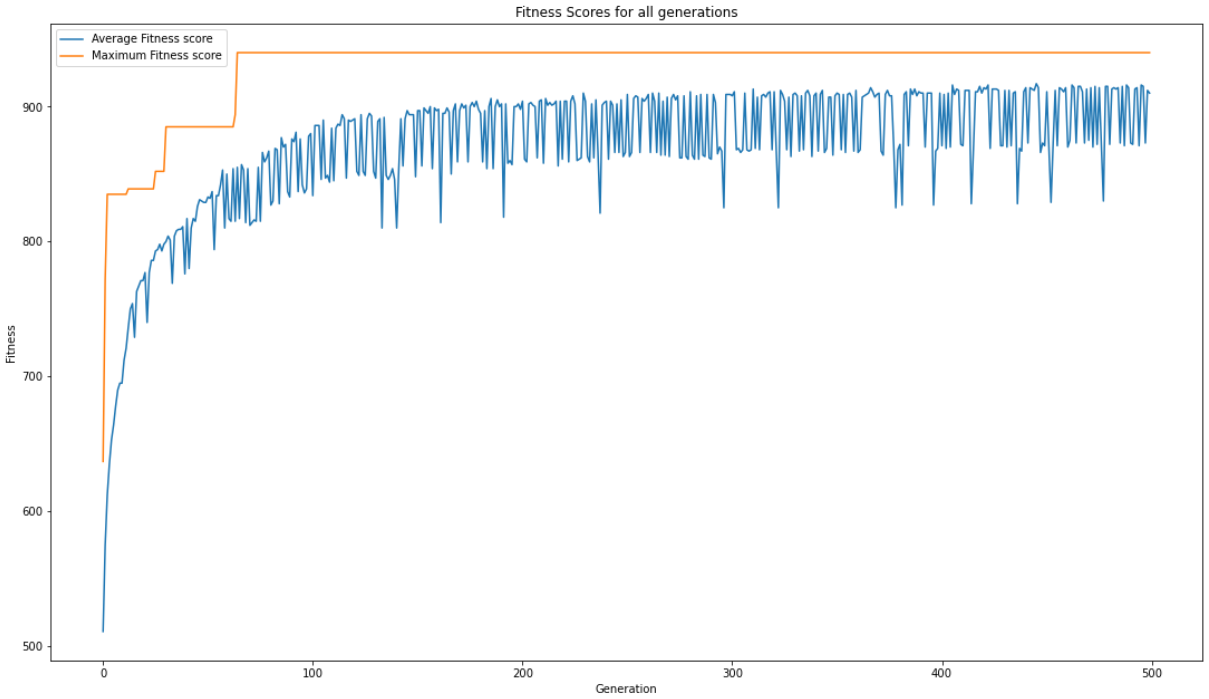

Fig.7. Average Fitness Score - Number of generations Graph (Knapsack Data-2, population size 20, maximum generations 500, weight limit 550, fitness criteria 'tournament')

Genetic algorithm shows similar performances for both datasets when the number of generations is 500. We reduced the total number of generations of the algorithm from **500** to **100** and found that the average fitness scores of Knapsack Data-1 approached the maximum fitness score earlier than the average fitness scores of Knapsack Data-2.

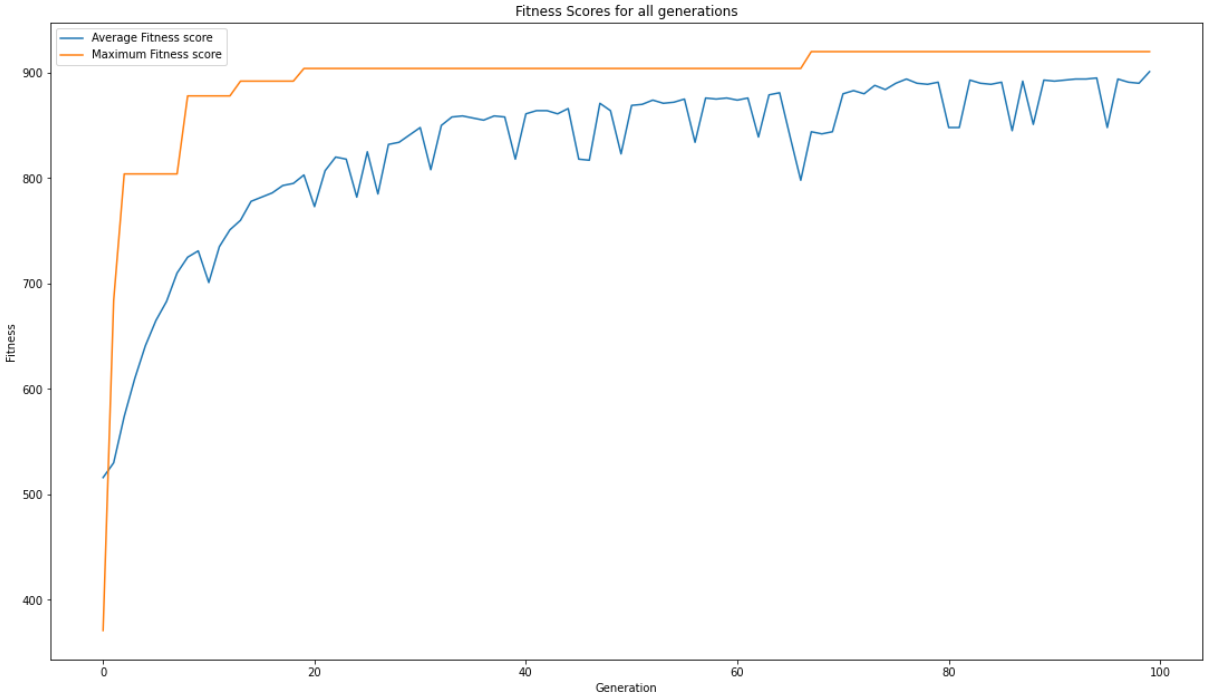

Fig.8. Average Fitness Score - Number of generations Graph (Knapsack Data-1, population size 20, maximum generations 100, weight limit 200, fitness criteria 'tournament')

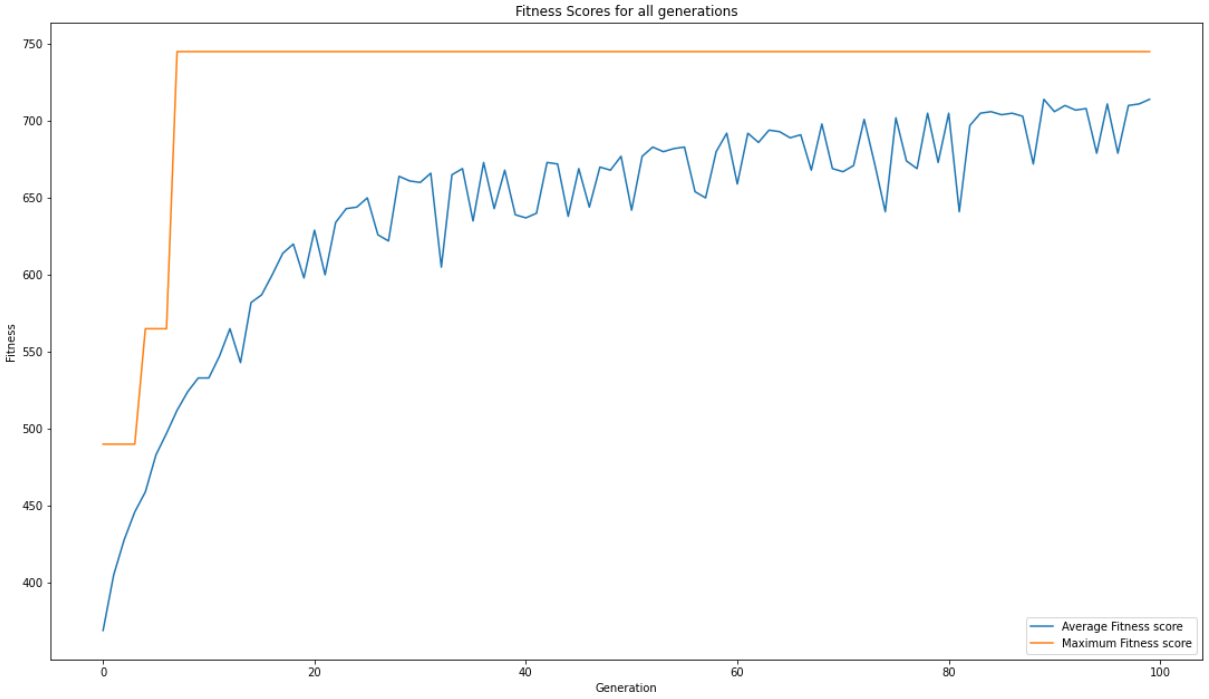

Fig.9. Average Fitness Score - Number of generations Graph (Knapsack Data-2, population size 20, maximum generations 100, weight limit 550, fitness criteria 'tournament')

4.2. Genetic Algorithm Performance Comparison for Individual Selection Methods

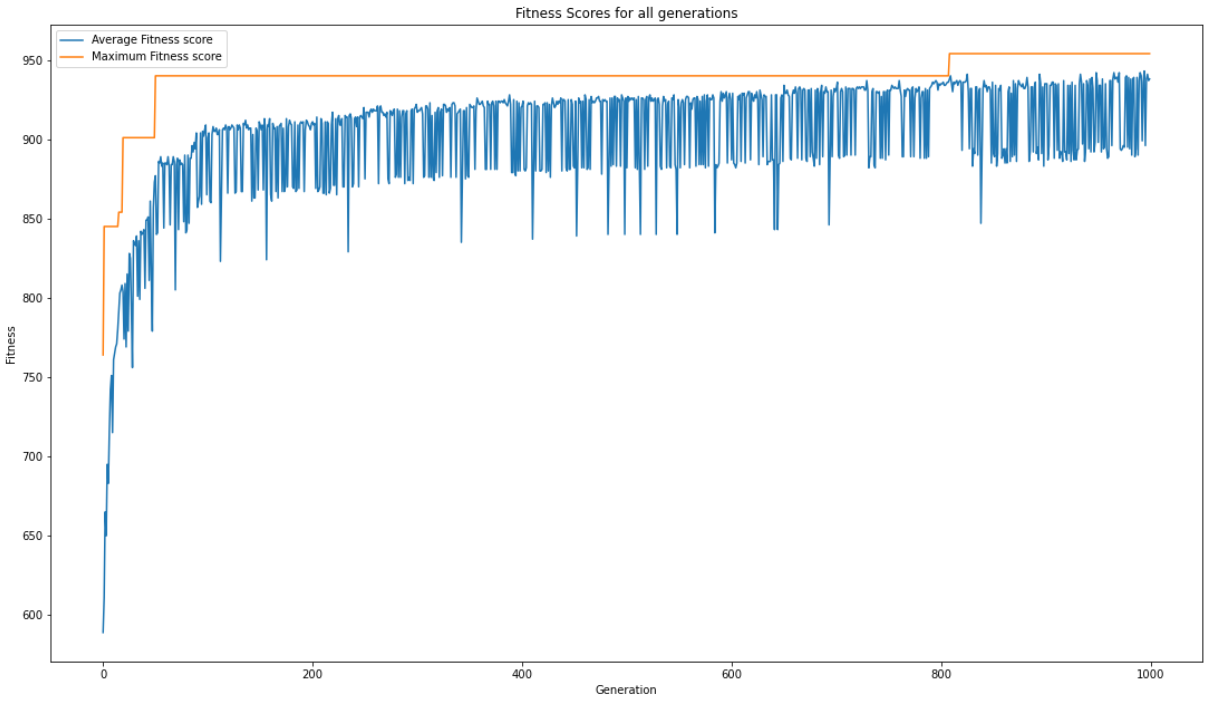

Fig.10. Average Fitness Score - Number of generations Graph (Knapsack Data-2, population size 20, maximum generations 1000, weight limit 550, fitness criteria 'tournament')

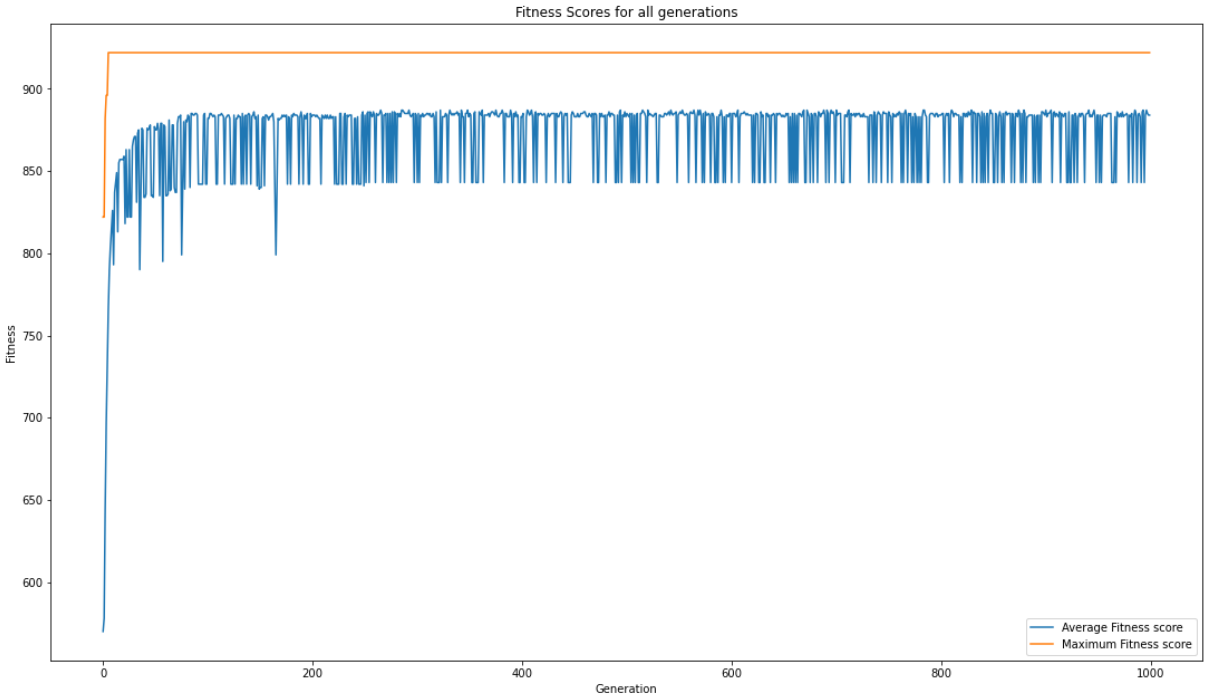

Fig.11. Average Fitness Score - Number of generations Graph (Knapsack Data-2, population size 20, maximum generations 1000, weight limit 550, fitness criteria 'best')

We use two different individual pair selection methods in our genetic algorithm; "best" and "tournament". In the "best" selection method, we find the best fitting individual and the second one and return them to the algorithm in order to apply some operations such as crossover and mutation. In the "tournament" selection method, we first shuffle all individuals, then we select two individuals randomly and compare themes fitness scores and keep the best one, we apply the same process to different randomly selected individual pairs and get the best one. Then, we return these two individuals to the algorithm.

We implemented these two selection algorithms on different datasets. We also tried these algorithms with different maximum generation numbers and population sizes, and as a result, we obtained the same result for all trials, 'tournement' individual selection method shows better performance on datasets. The '**best**' individual selection method decides the fittest individuals **faster**, but the average fitness score remains below the maximum fit score than when the '**tournament**' individual selection method is applied.

4.3. Genetic Algorithm Performance Comparison for Maximum Generation Numbers

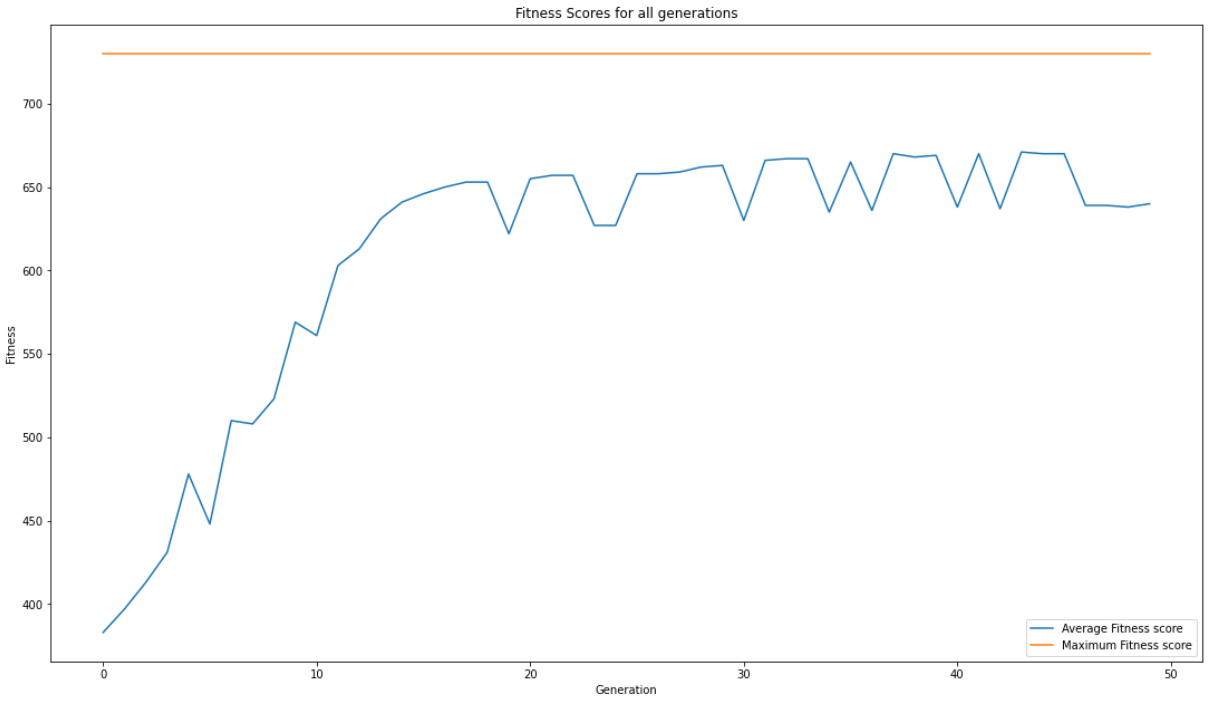

Fig.12. Average Fitness Score - Number of generations Graph (Knapsack Data-1, population size 20, maximum generations 50, weight limit is 200, fitness criteria is 'best')

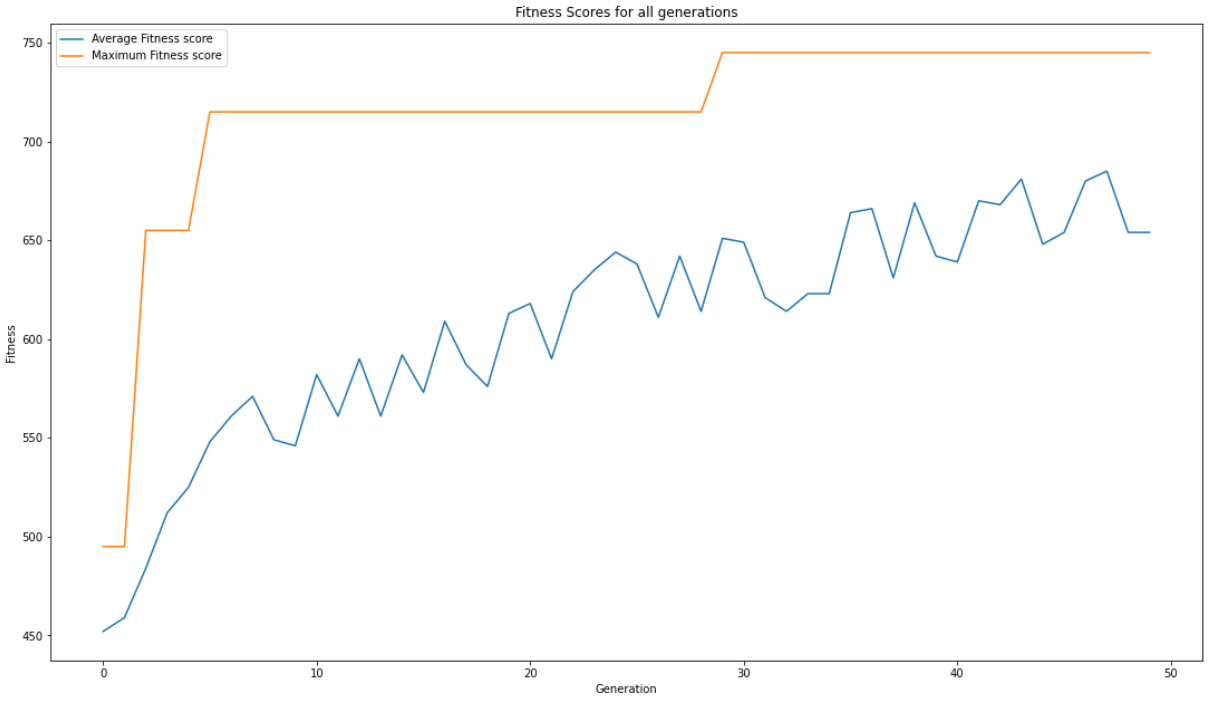

Fig.13. Average Fitness Score - Number of generations Graph (Knapsack Data-1, population size 20, maximum generations 50, weight limit is 200, fitness criteria is 'tournament')

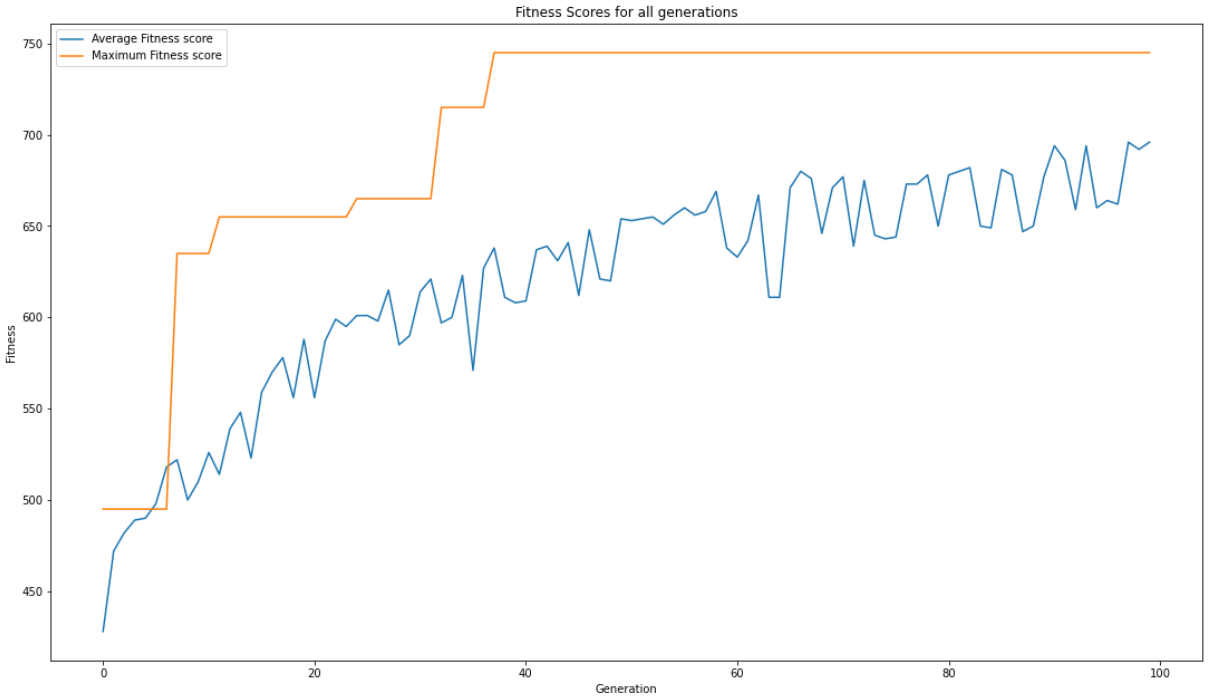

Fig.14. Average Fitness Score - Number of generations Graph (Knapsack Data-1, population size 20, maximum generations 100, weight limit is 200, fitness criteria is 'tournament')

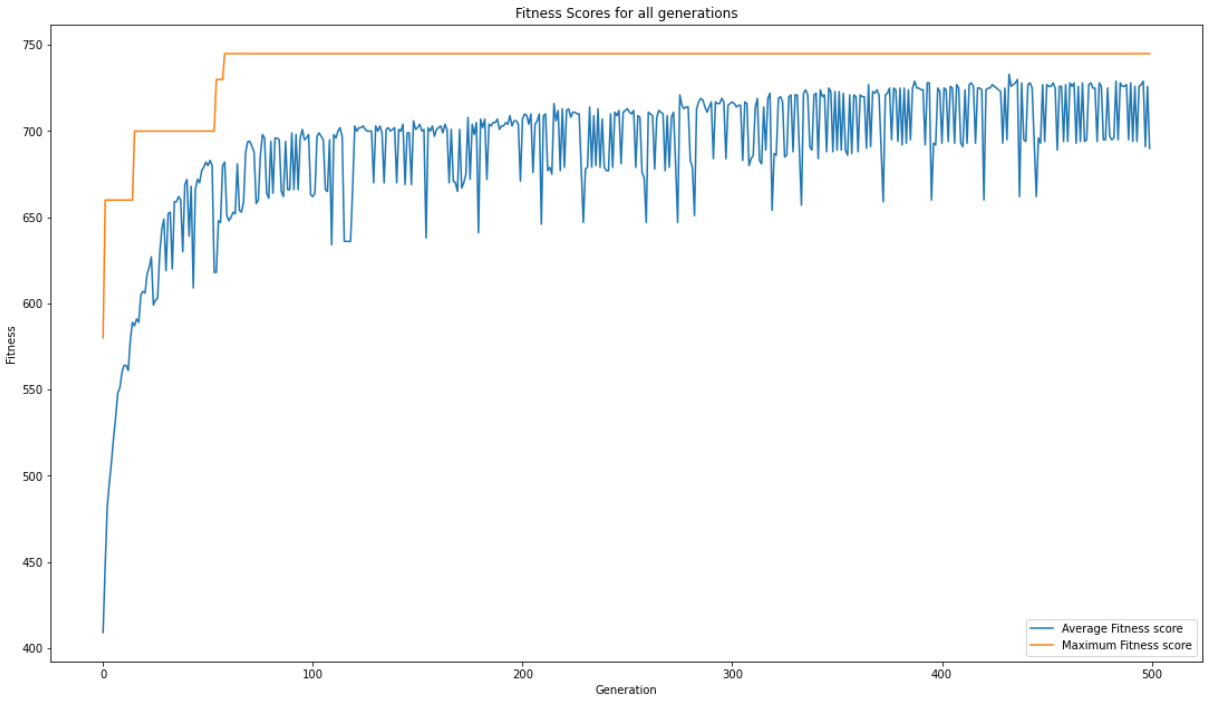

Fig.15. Average Fitness Score - Number of generations Graph (Knapsack Data-1, population size 20, maximum generations 500, weight limit is 200, fitness criteria is 'tournament')

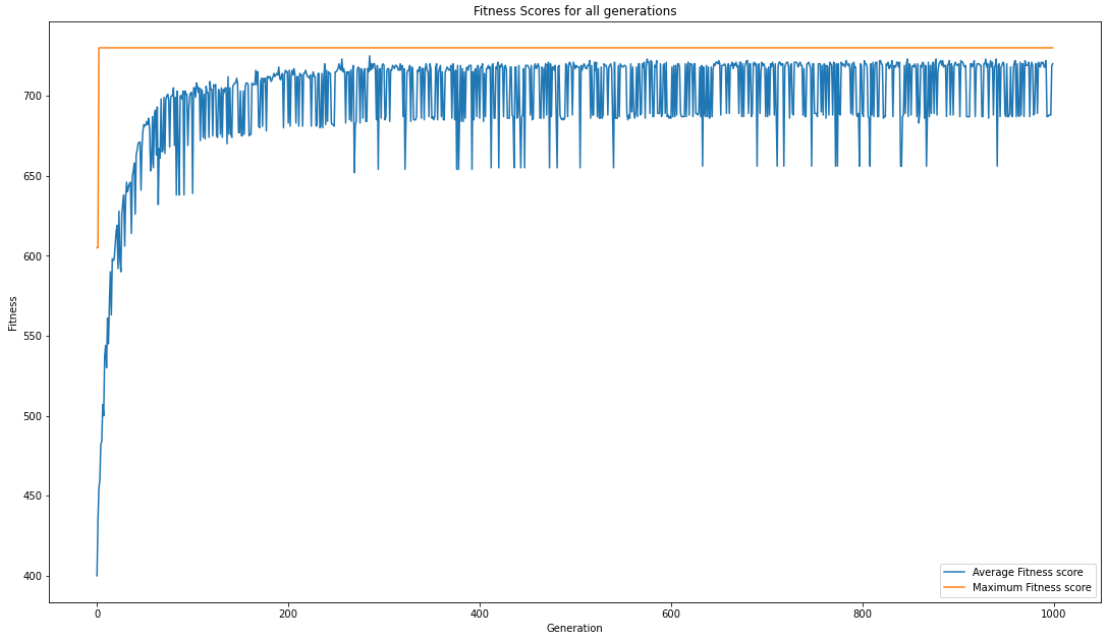

Fig.16. Average Fitness Score - Number of generations Graph (Knapsack Data-1, population size 20, maximum generations 1000, weight limit is 200, fitness criteria is 'tournament')

As can be seen in the graphs above, when the maximum number of generations is increased, the average fitness line becomes more stable, and the variation range becomes narrower. The maximum fitness score calculated from the populations produced to solve the problem is beginning to be found more precisely.

4.4. Genetic Algorithm Performance Comparison for Population Sizes

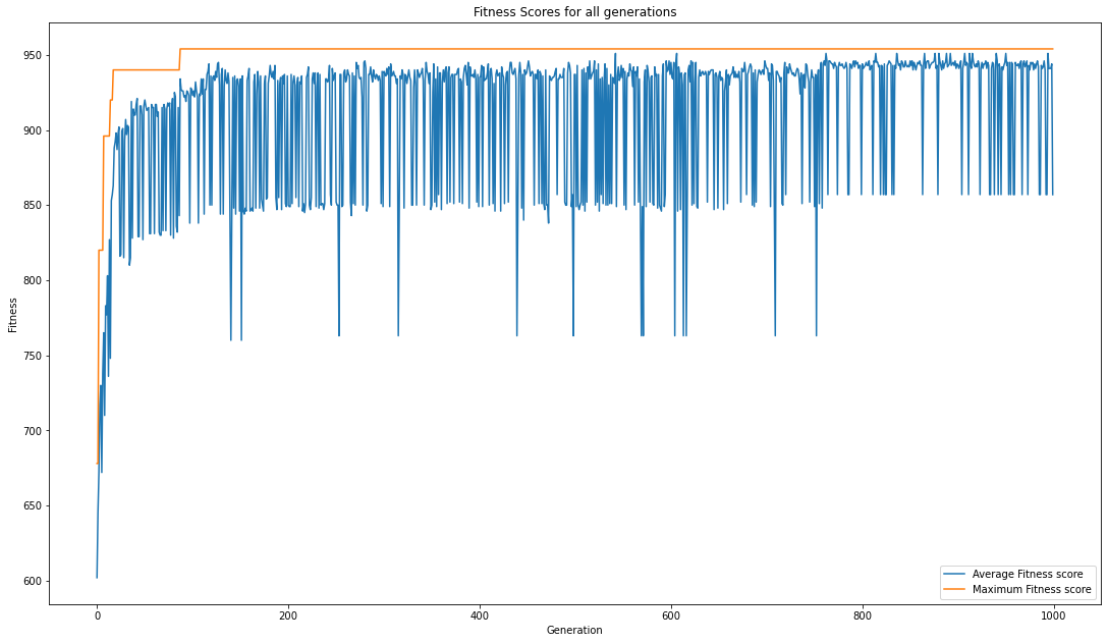

Fig.17. Average Fitness Score - Number of generations Graph (Knapsack Data-2, population size 10, maximum generations 1000, weight limit is 550, fitness criteria is 'tournament')

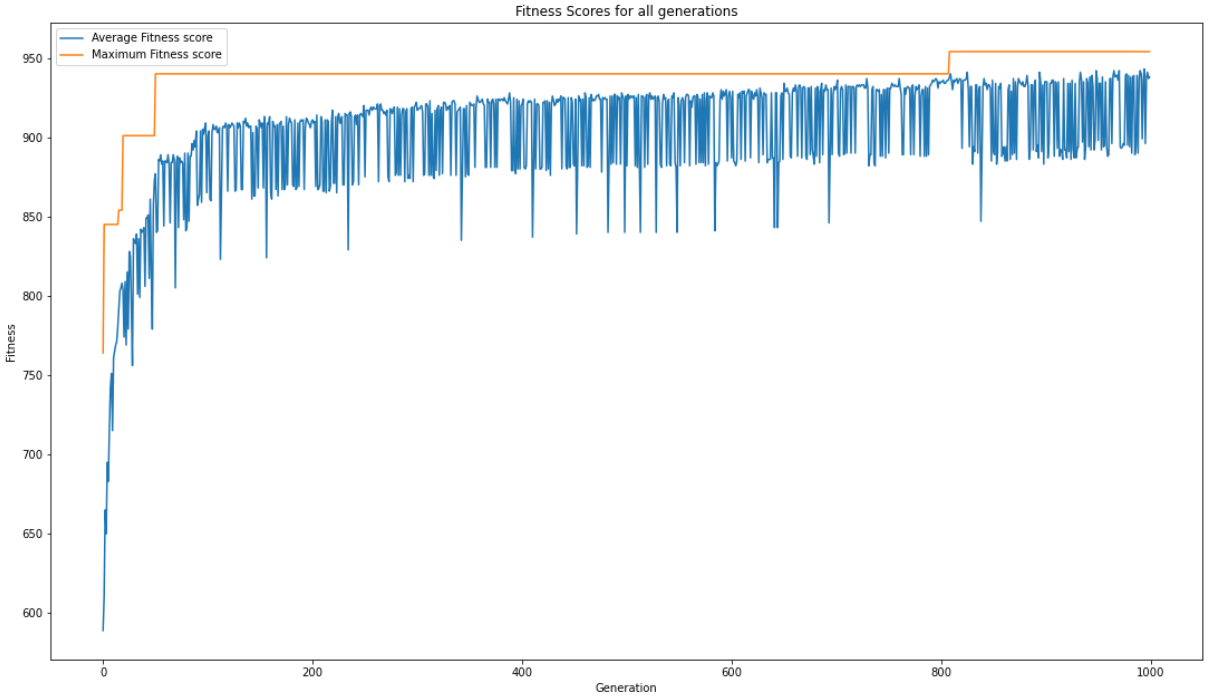

Fig.18. Average Fitness Score - Number of generations Graph (Knapsack Data-2, population size 20, maximum generations 1000, weight limit is 550, fitness criteria is 'tournament')

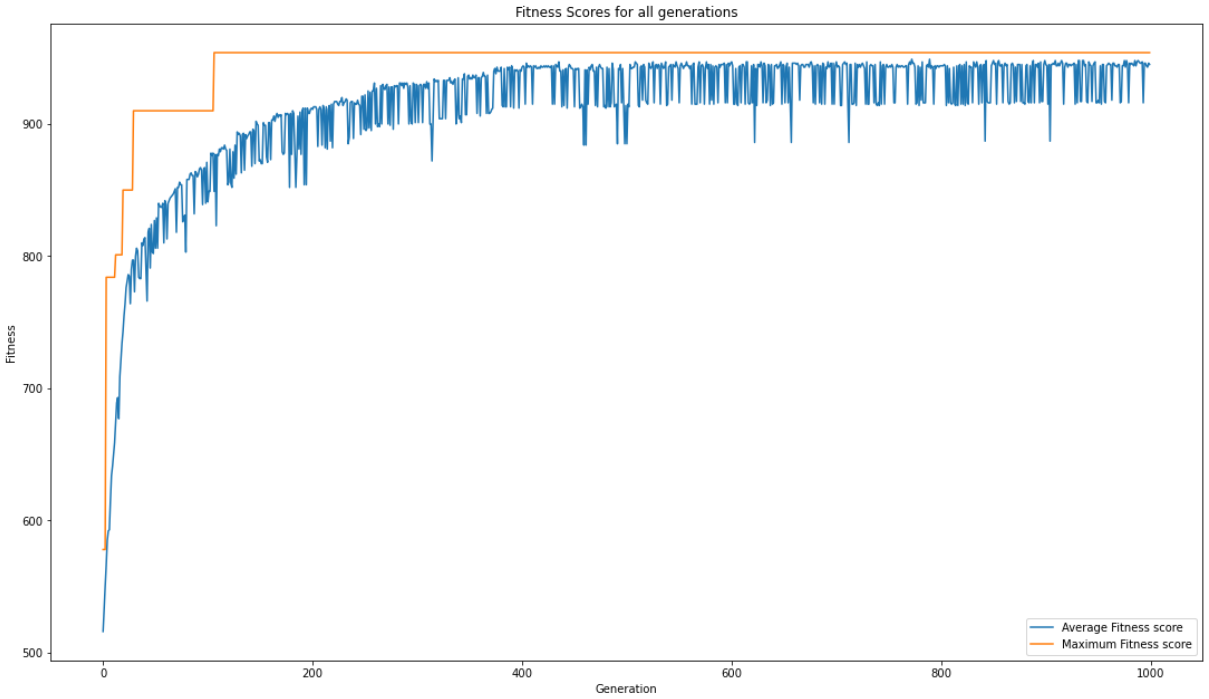

Fig.19. Average Fitness Score - Number of generations Graph (Knapsack Data-2, population size 30, maximum generations 1000, weight limit is 550, fitness criteria is 'tournament')

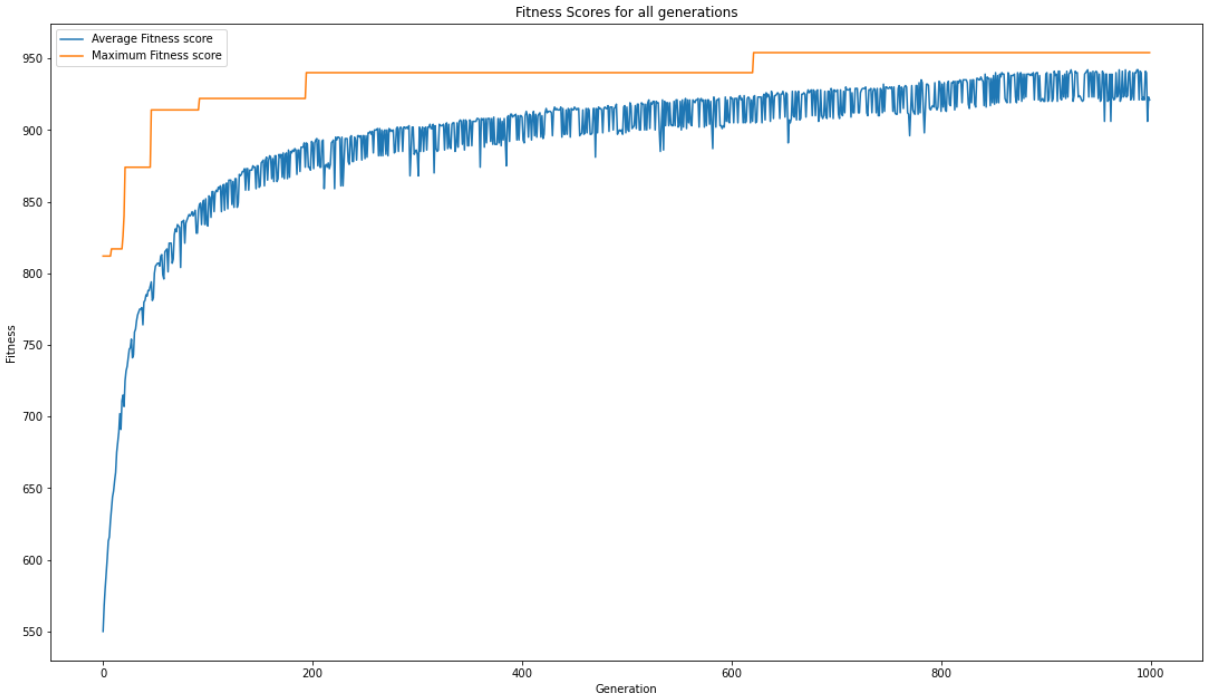

Fig.20. Average Fitness Score - Number of generations Graph (Knapsack Data-2, population size 30, maximum generations 1000, weight limit is 550, fitness criteria is 'tournament')

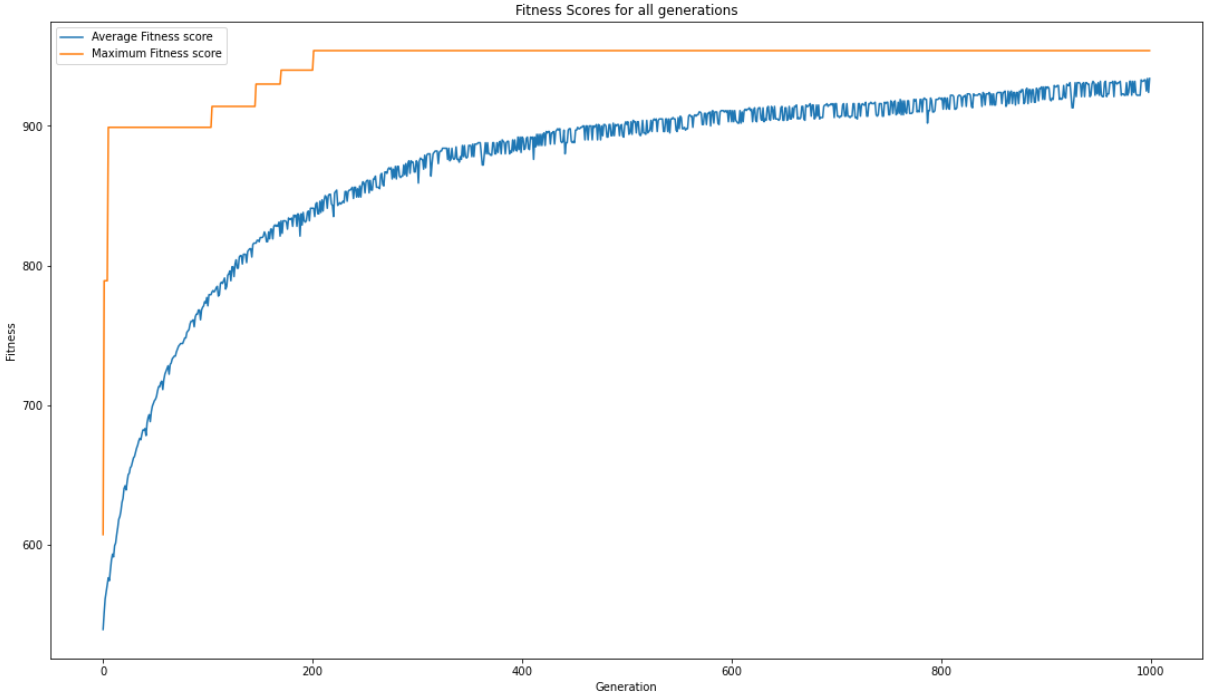

Fig.21. Average Fitness Score - Number of generations Graph (Knapsack Data-2, population size 100, maximum generations 1000, weight limit is 550, fitness criteria is 'tournament')

As can be seen from the graphs above, increasing the population size provided to get a better and more precise solution for the Knapsack problem which has the Knapsack Data-2 dataset. From this, it is understood that an optimum value should be found for the population size based on the size of the data set we have. When the population size starts to grow, as can be seen from the last graph, the program needs more generations for the solution. Therefore, the optimum solution is not to increase the population size as much as possible, but to determine this value optimally.

5. Conclusion

In this study, we solved the Knapsack problem using the genetic algorithm. We observed the variation of the mean fitness score with respect to the number of generations. Choosing the generation number constant low undermines the deterministic approach required to find the result due to randomness. As the number of produced generations increases, the population's average fitness scores become more stabilized. The reason for this is the logic behind evolution. Fertile offspring are inherited to the next generation, while unproductive offspring perish over time. As a result of these processes, populations begin to be produced, the majority of which are fertile offspring. We tried two different Knapsack problem data with different algorithm parameters such as the number of generations, number of population sizes, and individual selection methods. We used two different approaches to the selection process of the population for each individual: the '***best-fitting***' approach and the '***tournament***' approach. Then we used these methods in the individual selection phase. The 'Best-fitting' approach reached the average fitness score stability at lower generation numbers but the 'tournament' approach gave better results (solutions) for the given Knapsack problems. It means the Average fitness scores are more close to the Maximum fitness score if the fitting approach is the 'tournament' approach. Besides, we observed that if the number of populations is increased for datasets containing large numbers of data, the ***average fitness score*** calculated by the algorithm for the generated populations is much more ***stable*** and fits a line that is close to the maximum fitness score.

6. References

[1] https://www.excel-easy.com/vba/examples/knapsack-problem.html [accessed 6 Dec, 2022]

[2] https://www.theoptimizationexpert.com/case-studies/knapsack/ [accessed 6 Dec, 2022]

[3] https://en.wikipedia.org/wiki/Knapsack_problem [accessed 6 Dec, 2022]

[4] https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/ [accessed 6 Dec, 2022]

[5] https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3 [accessed 7 Dec, 2022]

[6] https://www.geeksforgeeks.org/genetic-algorithms/ [accessed 7 Dec, 2022]

[7] https://medium.com/@fabianterh/how-to-solve-the-knapsack-problem-with-dynamic-programming-eb88c706d3cf [accessed 8 Dec, 2022]

[8] Evolutionary Algorithms for Multiobjective and Multimodal Optimization of Diagnostic Schemes - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Pseudocode-of-the-genetic-algorithm-based-on-deterministic-crowding-used-for-PAF_fig5_7290874 [accessed 10 Dec, 2022]

[9] https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm [accessed 10 Dec, 2022]

[10] https://arpitbhayani.me/blogs/genetic-knapsack [accessed 18 Dec, 2022]

[11] Maya Hristakeva, Dipti Shrestha. Solving the 0-1 Knapsack Problem with Genetic Algorithms. Simpson College, Computer Science Department. USA.

Atalay PABUŞÇU

026217