6/11/2021

Knapsack Problem

-Genetic Algorithm Implementation

Castellanos, Alexis
UNIVERSITY OF MICHIGAN DEARBORN

1.Introduction

This project aims to produce a solution to the knapsack problem using the implementation of a genetic algorithm. The knapsack problem is a combinatorial optimization problem, where you are given a set of items each with a weight and value. The knapsack has a limit to how much weight it can carry, and you must maximize the value to gather equal to or under the weight threshold. A genetic algorithm, often abbreviated as GA, is a type of stochastic algorithm based on probability theory. Three main features of genetic algorithms are selection of fittest, crossover and mutation. Section 3 will outline the process on constructing the GA.

1.1 Environment and Files

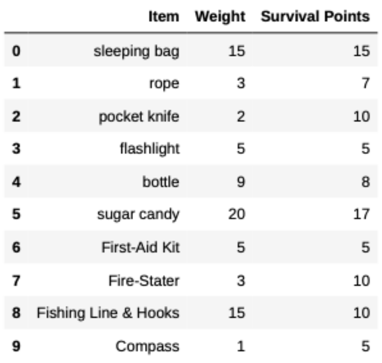
This solution will be developed is an Anaconda environment using Python 3 with a Jupyter Notebook. A PDF copy of the annotated source code will be submitted alongside this report. The submission will also include a folder including two files. ipynb (notebook environment) and .py (python compiler).

2.Data Pre-Processing

Data used for this can be represented by the data frame in *figure 2.0*. Items 0-5, weights, and values were provided to us. While items 6-9 were created by me and given corresponding weights and values.

Three python list will be created to hold number data from items (index), weights and value. These lists will then be passed into our genetic algorithm as parameters and set as attributes.

Figure 2.0

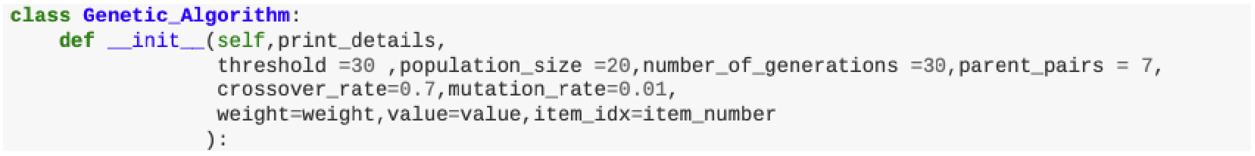


	Item	Weight	Survival Points
0	sleeping bag	15	15
1	rope	3	7
2	pocket knife	2	10
3	flashlight	5	5
4	bottle	9	8
5	sugar candy	20	17
6	First-Aid Kit	5	5
7	Fire-Stater	3	10
8	Fishing Line & Hooks	15	10
9	Compass	1	5

3.Solution Implementation

The given requirements for our GA were threshold of 30, initial population of 20, using 30 generations, with 7 parent pairs following a 70% crossover rate and 1% mutation rate. An object class was created to pass values as default parameters. The GA object will take the items, weights, and values as default parameters. The only parameter not set as default is **print_details,** a Boolean value when true will display generational information and results. Lastly all parameters will be set as attributes for our genetic algorithm class.

Figure 3.0



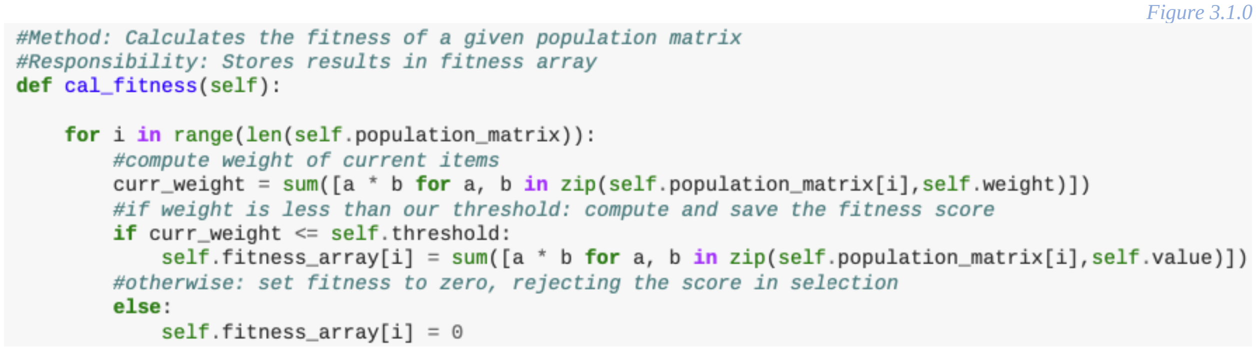
```
class Genetic_Algorithm:
    def __init__(self,print_details,
                 threshold =30 ,population_size =20,number_of_generations =30,parent_pairs = 7,
                 crossover_rate=0.7,mutation_rate=0.01,
                 weight=weight,value=value,item_idx=item_number
                ):
```

Initial population will be created by a class method named **create_population.** When invoked, it creates a random population (row) by item (column) matrix. These matrix value will be saved as an attribute named **population_matrix**.

3.1 Parent Selection Process

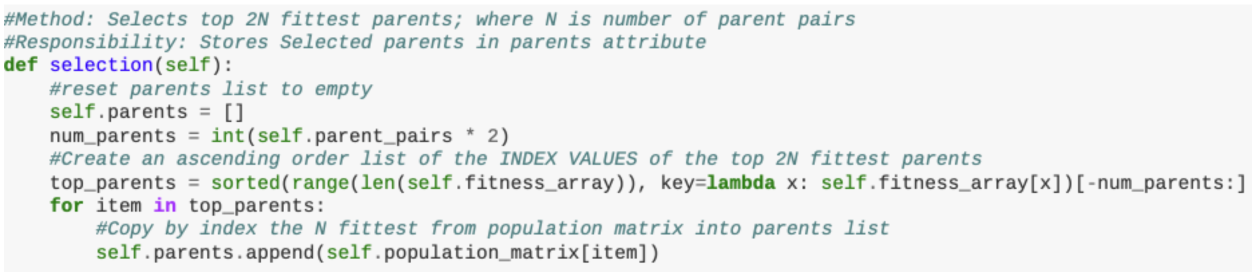
The fittest individuals of a population will be chosen as parents. Fitness can first be calculated when there exist a population. To first determine if a row in the population matrix is valid, we can multiply the current row by our weight. If the result is less than or equal to our threshold have possible solution. We many now calculate the fitness of this subject by multiplying it times the value list. Subjects that exceed our threshold will be set as zero to aid in the selection process. This process can be seen in *figure 3.1.0*. Selection is done by finding the top fittest (our case14 fittest) from the current population. These selected rows of our population matrix will now be saved as parents (dynamic attribute: type list). The selection method can be seen in *figure 3.1.1* below.

Figure 3.1.0



```
#Method: Calculates the fitness of a given population matrix
#Responsibility: Stores results in fitness array
def cal_fitness(self):

    for i in range(len(self.population_matrix)):
        #compute weight of current items
        curr_weight = sum([a * b for a, b in zip(self.population_matrix[i],self.weight)])
        #if weight is less than our threshold: compute and save the fitness score
        if curr_weight <= self.threshold:
            self.fitness_array[i] = sum([a * b for a, b in zip(self.population_matrix[i],self.value)])
        #otherwise: set fitness to zero, rejecting the score in selection
        else:
            self.fitness_array[i] = 0
```

Figure 3.1.1



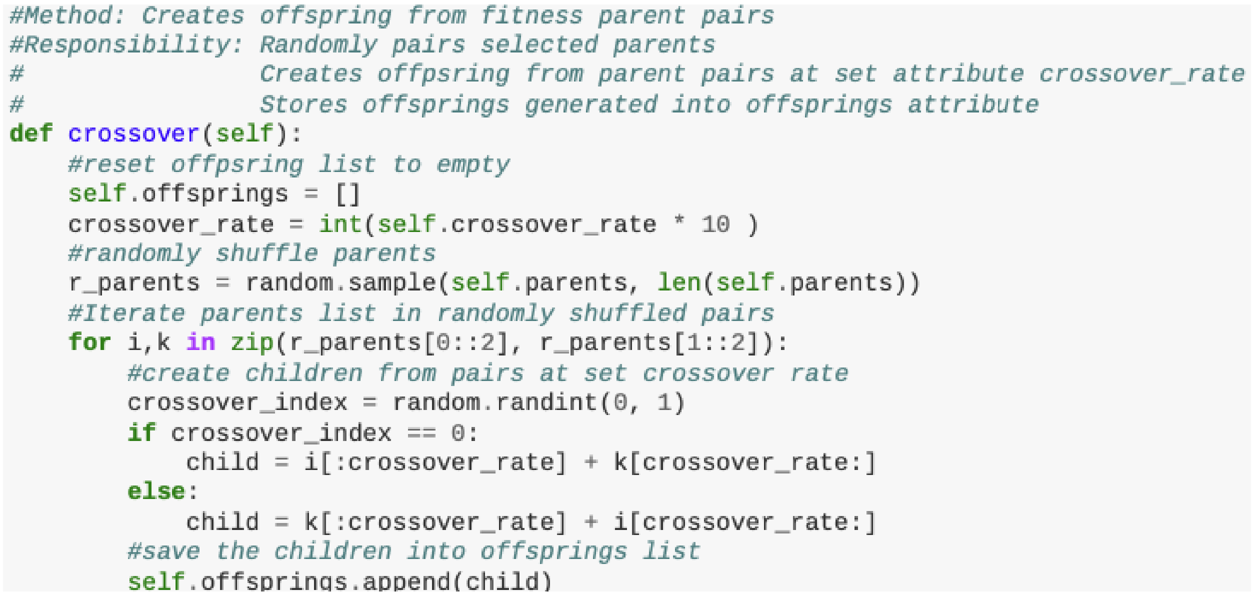
```
#Method: Selects top 2N fittest parents; where N is number of parent pairs
#Responsibility: Stores Selected parents in parents attribute
def selection(self):
    #reset parents list to empty
    self.parents = []
    num_parents = int(self.parent_pairs * 2)
    #Create an ascending order list of the INDEX VALUES of the top 2N fittest parents
    top_parents = sorted(range(len(self.fitness_array)), key=lambda x: self.fitness_array[x])[-num_parents:]
    for item in top_parents:
        #Copy by index the N fittest from population matrix into parents list
        self.parents.append(self.population_matrix[item])
```

3.2 Offspring and Next Population

The role of recombination plays an important role for robust adaptive systems. Recombination is implemented by crossover and is the process which operates on pairs of parents to produce offspring. For our implementation I have used a single crossover point at a rate of 70%. Crossover is implemented by selecting a random point in the parent's sequence and sequence values will be swapped from this point, creating an offspring. However, since we are using a rate of 70% at a length of 10, we will randomly choose an index 3 or 7 to swap values. This will assure offspring contains 70% of one of its parents. This process can be seen in *figure 3.2.0*.

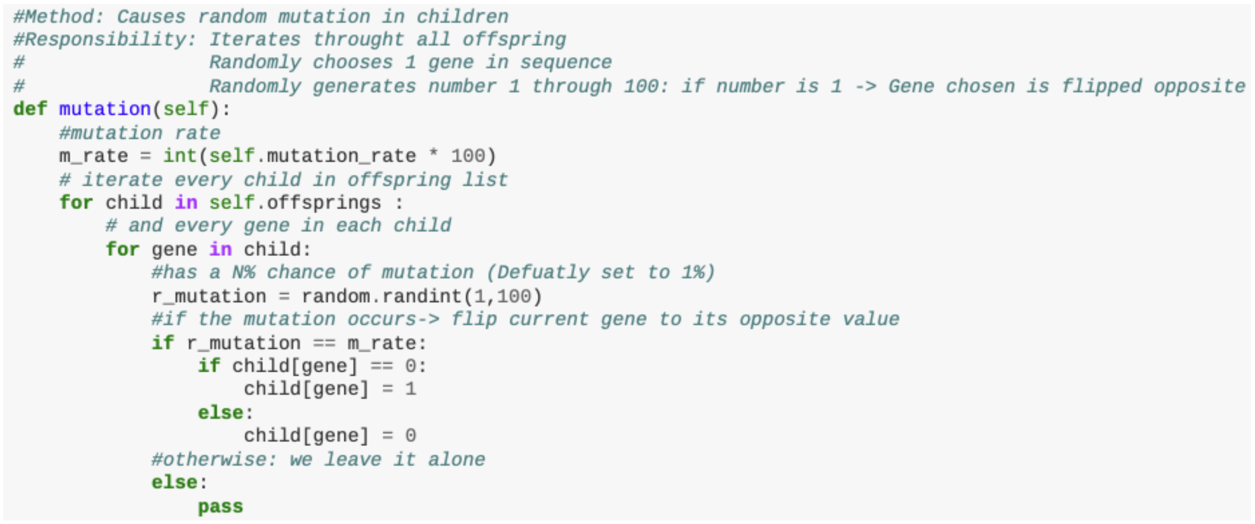
Before finalizing offspring, mutation can occur. Mutation is when a trait is randomly selected and flipped its current value. Mutation can help create diversity in solutions but may also hinder performance. In our case, each gene has a 1% chance of mutation. This process can be seen in *figure 3.2.2*. After mutation is complete, the offspring can be finalized.

Figure 3.2.0



```
#Method: Creates offspring from fitness parent pairs
#Responsibility: Randomly pairs selected parents
#                Creates offpsring from parent pairs at set attribute crossover_rate
#                Stores offsprings generated into offsprings attribute
def crossover(self):
    #reset offpsring list to empty
    self.offsprings = []
    crossover_rate = int(self.crossover_rate * 10 )
    #randomly shuffle parents
    r_parents = random.sample(self.parents, len(self.parents))
    #Iterate parents list in randomly shuffled pairs
    for i,k in zip(r_parents[0::2], r_parents[1::2]):
        #create children from pairs at set crossover rate
        crossover_index = random.randint(0, 1)
        if crossover_index == 0:
            child = i[:crossover_rate] + k[crossover_rate:]
        else:
            child = k[:crossover_rate] + i[crossover_rate:]
        #save the children into offsprings list
        self.offsprings.append(child)
```

Figure 3.2.1



```
#Method: Causes random mutation in children
#Responsibility: Iterates throught all offspring
#                Randomly chooses 1 gene in sequence
#                Randomly generates number 1 through 100: if number is 1 -> Gene chosen is flipped opposite
def mutation(self):
    #mutation rate
    m_rate = int(self.mutation_rate * 100)
    # iterate every child in offspring list
    for child in self.offsprings :
        # and every gene in each child
        for gene in child:
            #has a N% chance of mutation (Defuatly set to 1%)
            r_mutation = random.randint(1,100)
            #if the mutation occurs-> flip current gene to its opposite value
            if r_mutation == m_rate:
                if child[gene] == 0:
                    child[gene] = 1
                else:
                    child[gene] = 0
            #otherwise: we leave it alone
            else:
                pass
```

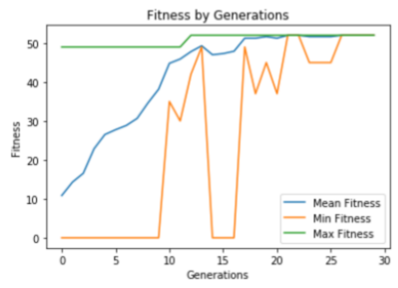
Once parents and offspring have been selected and created, we can reevaluate our current population matrix. The method **update_population** takes both parents offspring and re-evaluates the current population. After reevaluation we now replace current population entries with fitter ones from the parents & offspring list. This process will repeat up to the number of generations the user chooses.

4.Results

The figure below displays the fitness of generations throughout the iteration process, last generation matrix, fitness of last generation, fitness by generation graph and average fitness of last generation.

Figure 4.0

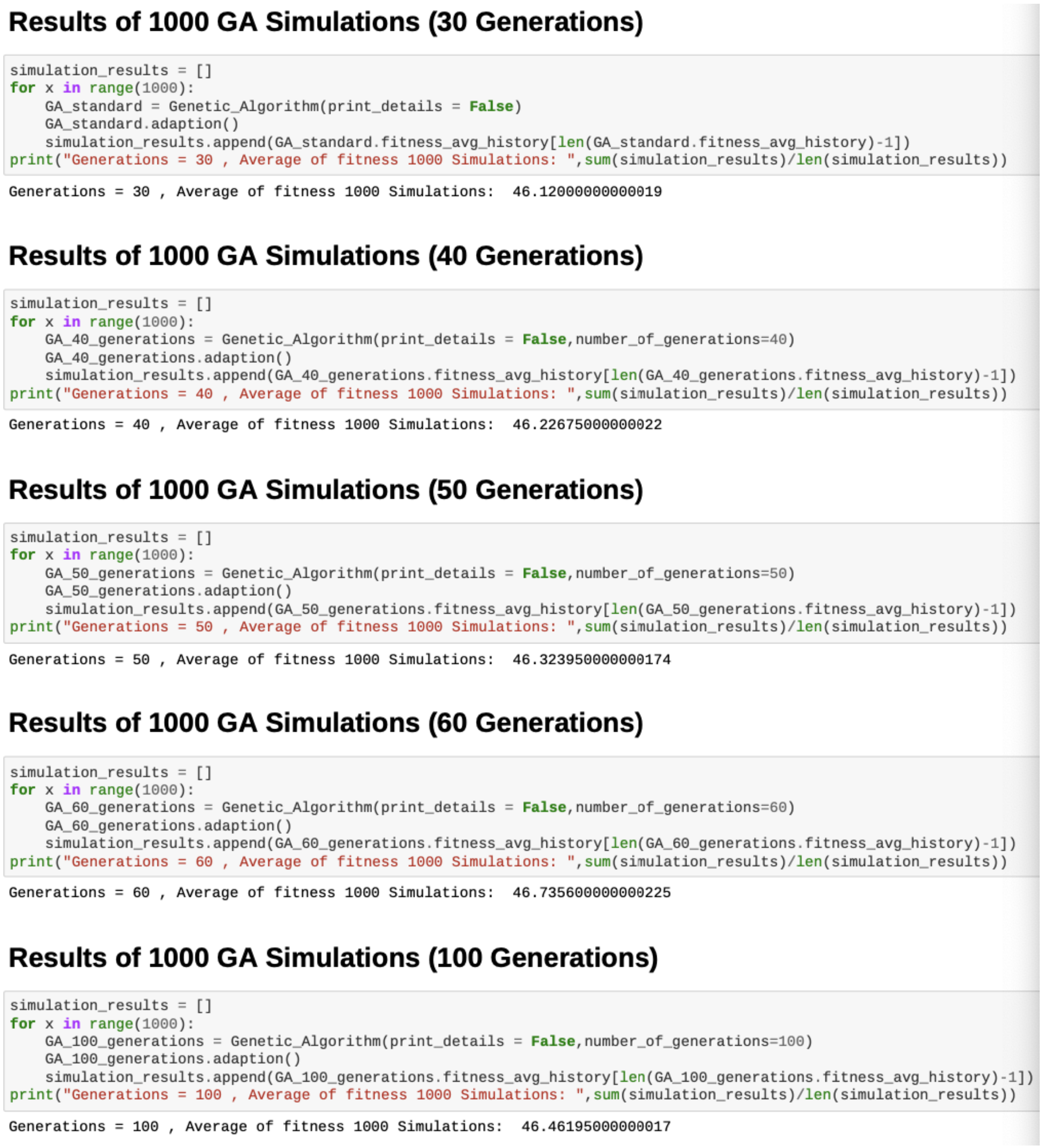
```
Fitness Score of Generation 1:[0, 0, 33, 49, 0, 0, 0, 0, 40, 0, 20, 25, 32, 18, 0, 0, 0, 0, 0, 0]
Fitness Score of Generation 2:[0, 0, 0, 0, 0, 0, 18, 20, 25, 32, 33, 40, 49, 0, 0, 0, 30, 39, 0, 0]
Fitness Score of Generation 3:[0, 0, 0, 0, 18, 20, 25, 30, 32, 33, 39, 40, 49, 20, 0, 0, 0, 0, 0, 25]
Fitness Score of Generation 4:[0, 0, 18, 20, 20, 25, 25, 30, 32, 33, 39, 40, 49, 25, 0, 23, 0, 25, 33, 20]
Fitness Score of Generation 5:[20, 23, 25, 25, 25, 25, 30, 32, 33, 33, 39, 40, 49, 15, 0, 22, 49, 25, 0, 20]
Fitness Score of Generation 6:[25, 25, 25, 25, 25, 30, 32, 33, 33, 39, 40, 49, 49, 25, 22, 0, 45, 0, 18, 15]
Fitness Score of Generation 7:[25, 25, 25, 25, 30, 32, 33, 33, 39, 40, 45, 49, 49, 30, 0, 0, 45, 25, 12, 15]
Fitness Score of Generation 8:[25, 25, 30, 30, 32, 33, 33, 39, 40, 45, 45, 49, 49, 0, 49, 49, 30, 0, 0, 10]
Fitness Score of Generation 9:[30, 30, 32, 33, 33, 39, 40, 45, 45, 49, 49, 49, 49, 0, 39, 30, 40, 45, 15, 0]
Fitness Score of Generation 10:[33, 33, 39, 39, 40, 40, 45, 45, 45, 49, 49, 49, 49, 0, 0, 45, 40, 39, 45, 39]
Fitness Score of Generation 11:[39, 40, 40, 40, 45, 45, 45, 45, 45, 49, 49, 49, 49, 44, 45, 49, 49, 45, 35, 49]
Fitness Score of Generation 12:[45, 45, 45, 45, 45, 45, 49, 49, 49, 49, 49, 49, 49, 49, 45, 49, 30, 45, 45, 42]
Fitness Score of Generation 13:[45, 45, 45, 45, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 52, 45, 49, 42]
Fitness Score of Generation 14:[49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 52, 49, 49, 52, 49, 49, 49, 49]
Fitness Score of Generation 15:[49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 52, 52, 49, 49, 49, 49, 52, 49, 0]
Fitness Score of Generation 16:[49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 52, 52, 52, 52, 52, 49, 49, 0, 49, 49]
Fitness Score of Generation 17:[49, 49, 49, 49, 49, 49, 49, 49, 52, 52, 52, 52, 52, 49, 0, 52, 49, 52, 52, 52]
Fitness Score of Generation 18:[49, 49, 49, 49, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 49, 52, 52, 52, 52]
Fitness Score of Generation 19:[52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 37, 52, 52, 52, 52, 52]
Fitness Score of Generation 20:[52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 45, 52, 52, 52, 52, 52, 52]
Fitness Score of Generation 21:[52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 37, 52, 52, 52]
Fitness Score of Generation 22:[52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52]
Fitness Score of Generation 23:[52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52]
Fitness Score of Generation 24:[52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 45, 52, 52, 52, 52]
Fitness Score of Generation 25:[52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 45, 52, 52, 52, 52, 52, 52]
Fitness Score of Generation 26:[52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 45, 52, 52]
Fitness Score of Generation 27:[52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52]
Fitness Score of Generation 28:[52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52]
Fitness Score of Generation 29:[52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52]
Fitness Score of Generation 30:[52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52]


Last Generation Matrix:
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1]


Fitness of the last generation: [52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52]
```



```
Average of fitness  of last gen: 52.0
```

The following figure are the results of 1000 simulations done per each generation values 30,40,50,60,100 displaying their average fitness scores.

Figure 4.1



Results of 1000 GA Simulations (30 Generations)

```
simulation_results = []
for x in range(1000):
    GA_standard = Genetic_Algorithm(print_details = False)
    GA_standard.adaption()
    simulation_results.append(GA_standard.fitness_avg_history[len(GA_standard.fitness_avg_history)-1])
print("Generations = 30 , Average of fitness 1000 Simulations: ",sum(simulation_results)/len(simulation_results))
```

```
Generations = 30 , Average of fitness 1000 Simulations:  46.12000000000019
```

Results of 1000 GA Simulations (40 Generations)

```
simulation_results = []
for x in range(1000):
    GA_40_generations = Genetic_Algorithm(print_details = False,number_of_generations=40)
    GA_40_generations.adaption()
    simulation_results.append(GA_40_generations.fitness_avg_history[len(GA_40_generations.fitness_avg_history)-1])
print("Generations = 40 , Average of fitness 1000 Simulations: ",sum(simulation_results)/len(simulation_results))
```

```
Generations = 40 , Average of fitness 1000 Simulations:  46.22675000000022
```

Results of 1000 GA Simulations (50 Generations)

```
simulation_results = []
for x in range(1000):
    GA_50_generations = Genetic_Algorithm(print_details = False,number_of_generations=50)
    GA_50_generations.adaption()
    simulation_results.append(GA_50_generations.fitness_avg_history[len(GA_50_generations.fitness_avg_history)-1])
print("Generations = 50 , Average of fitness 1000 Simulations: ",sum(simulation_results)/len(simulation_results))
```

```
Generations = 50 , Average of fitness 1000 Simulations:  46.323950000000174
```

Results of 1000 GA Simulations (60 Generations)

```
simulation_results = []
for x in range(1000):
    GA_60_generations = Genetic_Algorithm(print_details = False,number_of_generations=60)
    GA_60_generations.adaption()
    simulation_results.append(GA_60_generations.fitness_avg_history[len(GA_60_generations.fitness_avg_history)-1])
print("Generations = 60 , Average of fitness 1000 Simulations: ",sum(simulation_results)/len(simulation_results))
```

```
Generations = 60 , Average of fitness 1000 Simulations:  46.735600000000225
```

Results of 1000 GA Simulations (100 Generations)

```
simulation_results = []
for x in range(1000):
    GA_100_generations = Genetic_Algorithm(print_details = False,number_of_generations=100)
    GA_100_generations.adaption()
    simulation_results.append(GA_100_generations.fitness_avg_history[len(GA_100_generations.fitness_avg_history)-1])
print("Generations = 100 , Average of fitness 1000 Simulations: ",sum(simulation_results)/len(simulation_results))
```

```
Generations = 100 , Average of fitness 1000 Simulations:  46.46195000000017
```

5.Discussion

The results of our standard genetic algorithm are shown in *figure 4.0* and display the optimal solution. If we look at our <u>Fitness by Generations</u> plot, we can see Mean, Max and Min Fitness first all peek soon after generation 20. Soon after, Min and Mean fitness drop slightly and raise back after generation 25. I believe the drop we saw was due to random mutation affecting the population matrix. We also saw a pattern that our last generation often <u>only</u> contains one solution duplicated. This was not the case in earlier generations, at about 7-12 we can see the most diversity. Due to this behavior, I hypothesized increasing the generation values would not significantly improve the mean fitness.

Five genetic algorithms with different generation values were created (30,40,50,60,100). Each GA was then to be simulated 1000 times and averaged the fitness scores. As we can see from *figure 4.1,* the is slight improvements within generations. However, we can see a decrease from generation 60 to 100. Hence, the improvements <u>are not</u> significant nor stable enough to continue this process.

6.Conclusion

By implementing a genetic algorithm, we were able to provide solutions the knapsack optimization problem. The best solution can be accomplished by a GA given our standard parameters first at generation 22 and reached again in the last generation (30), yet not unique to 30 generations. Increasing the number of generations can generate higher fitness averages. However, these results are not significant nor consistent enough for continue increasing the number generations.