# Knapsack problem solving using Genetic Algorithm(GA):

The knapsack problem is an optimization problem. Given a set of elements, each with a weight and a profit, determine the number of each element to include in a bag(Knapsack) so that the total weight is less than or equal to a given capacity of bag and the total profit is as large as possible.

**Definition:**

The most common problem being solved is the 0-1 knapsack problem, which restricts the number xi of copies of each kind of item to zero or one. Given a set of n items numbered from 1 up to n each with a weight *wi* and a profit *vi* along with a maximum weight capacity W,

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0,1\}.$$

Where, xi represents the number of instances of elements i to include in the knapsack.

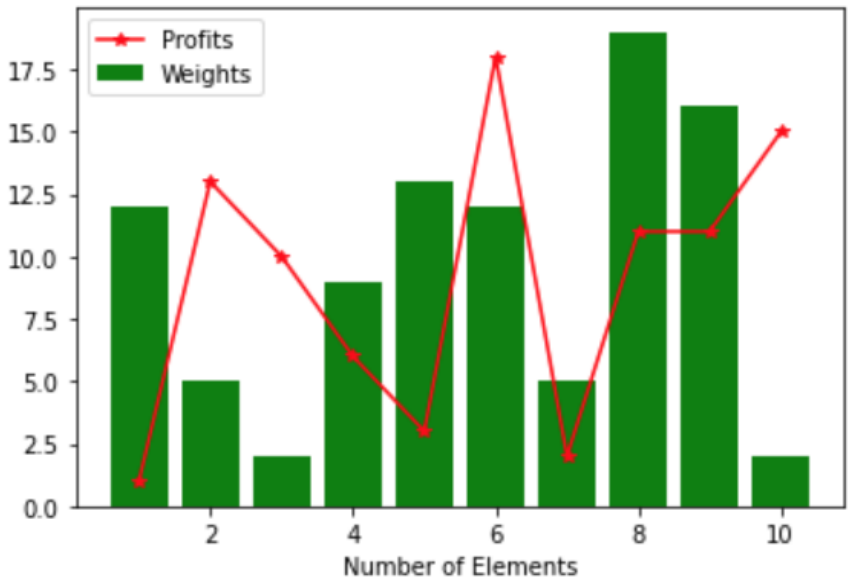
Consider the following instance of Knapsack Problem given 10 elements with associated weights and profits respectively. This chapter gives brief about importing respective libraries required for Knapsack Problem solving, weight and profit values has been initialized randomly, bag capacity is given 55 units and initial profit and weight graph is shown for respective elements.

```
import numpy as np
import pandas as pd
import random as rd
from random import randint
import matplotlib.pyplot as plt
element_number = np.arange(1,11)
weight = np.random.randint(1, 20, size = 10)
profit = np.random.randint(1, 20, size = 10)
bag_capacity = 55
print('The instance of a Knapsack Problem is given below:')
print('Elements No.   Weights   Profits')
for i in range(element_number.shape[0]):
```

```
    print('     {0}           {1}
{2}\n'.format(element_number[i], weight[i], profit[i]))
from matplotlib import pyplot as plt
plt.bar(element_number,weight,color='g',label='Weights')
plt.plot(element_number,profit,color='r',marker='*',label='Prof
its')
plt.legend()
plt.xlabel("Number of Elements")
plt.show()
```

**Output:**

```
The instance of a Knapsack Problem is given below:
Elements No.   Weights   Profits
     1           12        1

     2           5        13

     3           2        10

     4           9        6

     5           13        3

     6           12        18

     7           5        2

     8           19        11

     9           16        11

     10           2        15
```




## GA - Code Initialization

This chapter initialises the various variables like solution per population, population size, initial population, number of generations considered and initial population has been highlighted.

Providing the solution of a given bounded knapsack problem using genetic algorithms, first create a population, it has individuals and each individual has their own set of chromosomes. In this problem the genotype structure of chromosomes is 'binary' strings. If structure has '1' means that item is considered and if '0' means item is not considered or dropped.

```
solutions_per_pop = 4
pop_size = (solutions_per_pop, element_number.shape[0])
print('Population size = {}'.format(pop_size))
initial_population = np.random.randint(2, size = pop_size)
initial_population = initial_population.astype(int)
num_generations = 100
df=pd.DataFrame(initial_population)
df.style.highlight_max(color='green',axis=1)
```

**Output:**

```
Population size = (4, 10)
```

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| **1** | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| **2** | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| **3** | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |

## GA- Fitness Value Calculation

Fitness value is calculated using profit maximization formula for Knapsack Problem using constraint total weight of selected elements should not exceed bag capacity and maximum profit should be earned.

```
def fitness_value_calculation(weight, profit, population,
capacity):
    best_score_progress=[]
    fitness = np.empty(population.shape[0])
    for i in range(population.shape[0]):
        first_solution = np.sum(population[i] * profit)
        second_solution = np.sum(population[i] * weight)
        if second_solution <= capacity:
            fitness[i] = first_solution
        else :
            fitness[i] = 0
    return fitness.astype(int)
```

## GA - Selecting Parents and GA - Merging, Sorting and Selection

After calculating the fitness of each chromosome, the fittest are selected from the pool which come together to produce their offsprings. Generally, we should reproduce the chromosomes which have more fitness value. Although, it creates convergence of population as less diversity in solutions within a few generations.

```
def parent_selection(fitness, num_parents, population):
    fitness = list(fitness)
    parents = np.empty((num_parents, population.shape[1]))
```

```
    for i in range(num_parents):
        max_fitness_idx = np.where(fitness == np.max(fitness))
        parents[i,:] = population[max_fitness_idx[0][0], :]
        fitness[max_fitness_idx[0][0]] = -999999
    return parents
```

## GA - Performing Crossover

Now crossover operation is applied on chosen parent chromosomes to generate two new members for the next generation to produce off-springs.

```
def crossOver(selected_parents, num_childs):
    childs = np.empty((num_childs, selected_parents.shape[1]))
    crossOver_point = int(selected_parents.shape[1]/2)
    crossOver_rate = 0.5
    i=0
    while (selected_parents.shape[0] < num_childs):
        parent1_index = i%selected_parents.shape[0]
        parent2_index = (i+1)%selected_parents.shape[0]
        x = rd.random()
        if x > crossOver_rate:
            continue
        parent1_index = i%selected_parents.shape[0]
        parent2_index = (i+1)%selected_parents.shape[0]
        childs[i,0:crossOver_point] =
selected_parents[parent1_index,0:crossOver_point]
        childs[i,crossOver_point:] =
selected_parents[parent2_index,crossOver_point:]
        i=+1
    return childs
```

## GA - Performing Mutation

Change in chromosome structure or random alteration in genes is known as "Mutation" therefore a random swapping of genes on chromosome is done here, which promotes the idea of diversity in the population.

```
def mutation(childs):
    mutants = np.empty((childs.shape))
```

```
    mutation_rate = 0.1
    for i in range(mutants.shape[0]):
        random_profit = rd.random()
        mutants[i,:] = childs[i,:]
        if random_profit > mutation_rate:
            continue
        int_random_profit = randint(0,childs.shape[1]-1)
        if mutants[i,int_random_profit] == 0 :
            mutants[i,int_random_profit] = 1
        else :
            mutants[i,int_random_profit] = 0
    return mutants
```

## GA - The Main Function, GA - The main Loop and GA - Best Solution

```
def optimization_process(weight, profit, population,
initial_population_size, gen_number, capacity):
    operators, fitness_history,best_score_progress =[], [], []
    num_parents = int(initial_population_size[0]/2)
    num_childs = initial_population_size[0] - num_parents
    for i in range(gen_number):
        fitness = fitness_value_calculation(weight, profit,
population, capacity)
        fitness_history.append(fitness)

        parents = parent_selection(fitness, num_parents,
population)
        childs = crossOver(parents, num_childs)

        mutants = mutation(childs)

        population[0:parents.shape[0], :] = parents
        population[parents.shape[0]:, :] = mutants

    print('Best Generation Population as follows: \n
{}\n'.format(population))
    fitness_last_gen = fitness_value_calculation(weight,
profit, population, capacity)
    print('Value of Fitness for the Best Generation as follows:
\n {}\n'.format(fitness_last_gen))
    max_fitness = np.where(fitness_last_gen ==
```

```
np.max(fitness_last_gen))
    operators.append(population[max_fitness[0][0],:])
    return operators, fitness_history
```

## GA - Finalizing and Running GA

```
operators, best_score_progress= optimization_process(weight,
profit, initial_population, initial_population_size,
gen_number, bag_capacity)
print('Selected elements as below (1=> selected and 0=> Not
selected): \n{}'.format(operators))
selected_elements = element_number * operators
print('\nSelected elements that will earn maximum profit
without exceeding the bag capacity:')
for i in range(selected_elements.shape[1]):
  if selected_elements[0][i] != 0:
     print('{}\n'.format(selected_elements[0][i]))
```

**Output:**

```
Best Generation Pupulation as follows:
 [[1 1 0 1 1 0 1 1 1 1]
 [1 1 0 1 1 0 1 1 1 1]
 [1 0 0 1 1 0 1 1 1 1]
 [1 1 0 1 1 0 1 1 1 1]]

Value of Fitness for the Best Generation as follows:
 [5041 5041 4254 5041]

Selected elements as below (1=> selected and 0=> Not slected):
[array([1, 1, 0, 1, 1, 0, 1, 1, 1, 1])]

Selected elements that will earn maximum profit without exceeding the bag capacity:
1

2

4

5

7

8

9

10
```

## GA - Variable Ranges and GA - Decision Variable Bounds

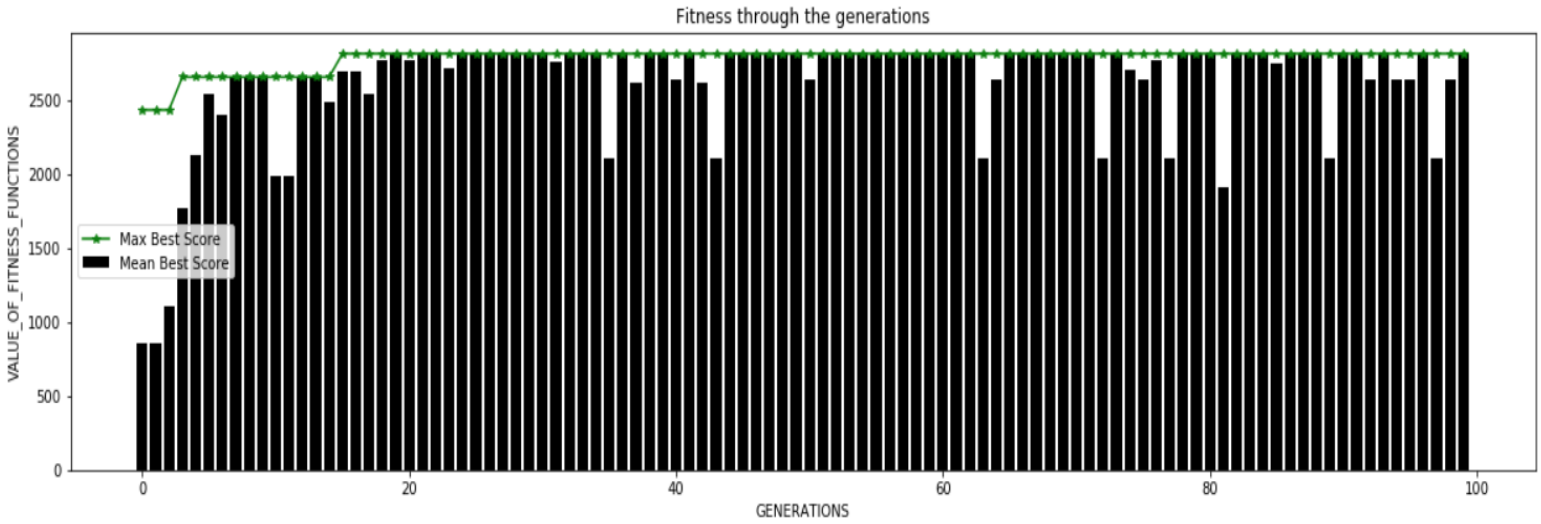
```
per_population_solutions =4 # Solutions per population
gen_number = 100 # Evolutionary Generations
initial_population_size[0] # Initial Population Size
initial_population # Length of Chromosome is 10
crossOver_rate = 0.5 # Crossover Probability
mutation_rate = 0.1 # Mutation Probability
```

## GA - Evaluation and Comparison

```
best_score_progress_mean = [np.mean(fitness) for fitness in
best_score_progress]
```

```
best_score_progress_max = [np.max(fitness) for fitness in
best_score_progress]
plt.bar(list(range(gen_number)), best_score_progress_mean,
color='k',label = 'Mean Best Score')
plt.plot(list(range(gen_number)), best_score_progress_max,
color='g', marker='*',label = 'Max Best Score')
plt.legend()
plt.title('Fitness through the generations')
plt.xlabel('GENERATIONS')
plt.ylabel('VALUE_OF_FITNESS_FUNCTIONS')
fig = plt.gcf()
fig.set_size_inches(20, 5)
plt.show()
print(np.asarray(best_score_progress).shape)
```

**Output:**




```
(100, 4)
```