The Knapsack Problem – Genetic Algorithm

Nayan Arora
Faculty of Science and Technology
University of Canberra
Canberra, Australia
u3249907@uni.canberra.edu.au

***Abstract*—The following report will explain the methods used to develop and implement the Genetic Algorithm to solve the Knapsack Problem with a maximum threshold weight of 35kgs. The steps used to develop the solution along with a brief literature review will be used to summarize how the optimal results were achieved. Further discussion on results will reflect the overall inference and understanding of Genetic Algorithm (GA). The goal of this project is to create an in-depth understanding of Genetic Algorithms and its connection to the real-world as well as its applications in various scenarios.**

Keywords—genetic algorithm, knapsack, fitness evaluation, maximized selection, mutation, crossover.

I. Introduction

This report explains the methodology and implementation of the Genetic Algorithm to solve the Knapsack Problem. The problem statement is that we have a Knapsack (bag) that has a maximum threshold for the weight that it can carry. Now we assume there are a number of items with variable weight and corresponding price values. The overall goal is to select items such that we maximize the total price/value, while ensuring that we do not exceed the maximum weight that the knapsack can hold.

We use Genetic Algorithm for solving this problem by repeatedly modifying a population of individual solutions. We perform multiple steps where at each step the algorithm selects individuals (items) from the current population to be parents and uses them to produce children for the next generation [2]. Over multiple iterations the population evolves to produce an optimal solution.

In summary genetic algorithm mimics the working of biological natural selection process because it operates with a population of candidate solutions, each represented as a set of chromosomes that carry genetic information. In context of the knapsack problem a set of chromosomes would represent a set of items to be included in the knapsack. Furthermore, Genetic Algorithm is applied as a search and optimization technique for various types of problems including the ones in which the objective functions are discontinuous, nondifferentiable, stochastic and highly nonlinear.

II. Literature Review

A. *Genetic Algorithm (GA)*

GA is a part of evolutionary algorithms that use adaptive heuristic search algorithms [2]. As mentioned earlier, GA is based on the idea of natural selection, that is to be able to adapt to the changes in the environment and survive (survival of the fittest). The species that survives these changes can reproduce and go to the next generation [3]. Following these biological ideologies, the genetic algorithm is able to generate high quality solutions for optimization problems. The steps used in GA are as below [3]:

- Individuals in a population compete for resources to adapt to the environment.
- The fittest individuals survive to then mate and create more offspring.
- The fittest genes propagate throughout the generation. This means that sometimes parents can create an offspring that is better than either parent.
- Thus, each successive generation is better able to adapt to the changing environment.

In this process a fitness score is given to everyone in a population. The fitness score represents the ability of an individual to 'compete'. Individuals with the optimal fitness scores are then chosen. The operational steps followed by GA are as follows [2][3]:

- Initialization: We first begin by initializing a population of chromosomes or candidate solutions. In context, these chromosomes represent different combinations of items to be put in the knapsack.
- Fitness Evaluation: Next, we calculate the fitness for each chromosome on two metrics. First is if it satisfies the weight limit and second is if it maximizes the total price/value.
- Selection: We then select the chromosomes or candidate solutions from the population based on the previously calculated fitness scores. Highest fitting chromosomes are preferred for selection thus mimicking the natural process of 'survival' of the fittest.

- Crossover: In this step, we pair the selected chromosomes and perform a crossover to create a new offspring. In context of the knapsack problem, the one-point crossover is done based on the position of the selected items (1's and not 0's) in the parent chromosomes. This ensures that the total weight does not exceed the threshold capacity.
- Mutation: In this final step, we introduce random changes to some of the offspring chromosomes. This helps promote variability and diversity in the population.

Genetic Algorithm using the above steps has proven to be exponentially more efficient than standard optimization algorithms. It can find near-optimal solutions by evolving item combinations over multiple generations. Thus, using the heuristic approach offered by GA we can tackle more complex optimization problems easily and effectively.

B. *Genetic Programming*

We use the flowchart attached below to generate inference for the methodology we will use for writing python code.

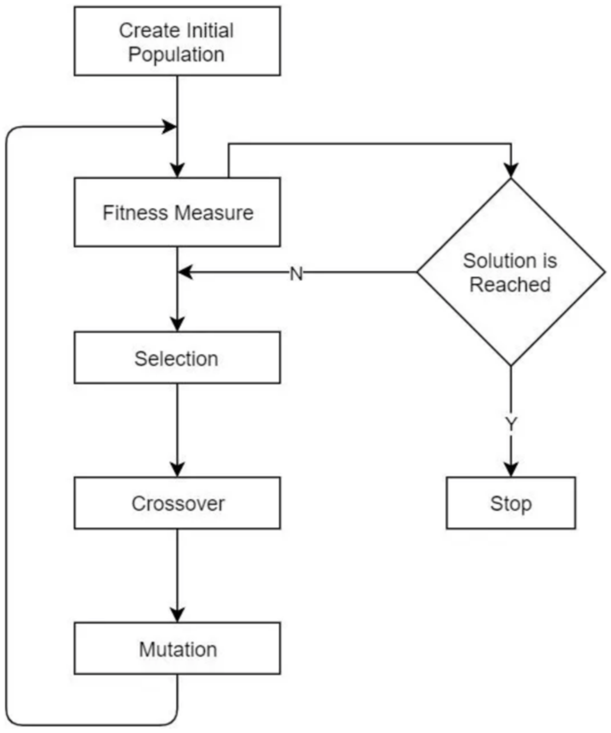


Figure 1: Steps used in Genetic Algorithm

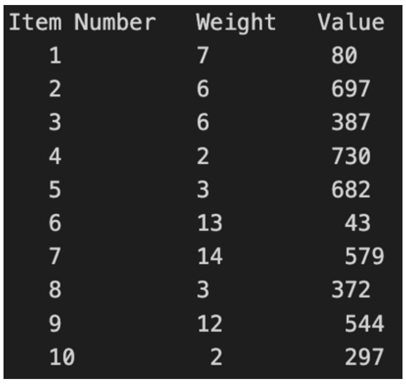
We can summarize the genetic Algorithm as below [2]:

- Randomly initialize a population.
- Evaluate the fitness of the population.
- Repeat the below until convergence:
    - Selected parents
    - Crossover to generate offspring.
    - Mutate the new population.
    - Evaluate the fitness of the new population.

Using the above steps, we define the functions that will be called in order for the defined initial values to optimize and generate a maximum value for the knapsack problem.

III. METHODOLOGY

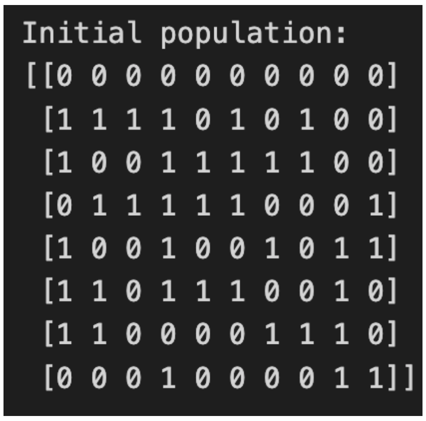
This section of the report will cover the methodology and the overall structure of the algorithm as defined in the python environment to achieve optimal results for the knapsack problem. We first initialize a random list of items with weight and price values. The list produced is attached below:



```
Item Number   Weight   Value
  1           7         80
  2           6        697
  3           6        387
  4           2        730
  5           3        682
  6           13        43
  7           14       579
  8           3        372
  9           12       544
  10           2       297
```

Figure 2: List of items

Then, we get the Initial population matrix as below. Here, each gene has a value of 1 or 0 that represents if an item is present or not.



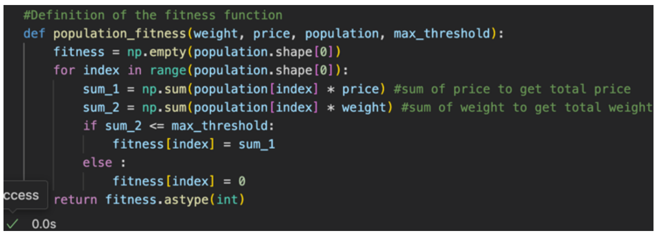
```
Initial population:
[[0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 0 1 0 1 0 0]
 [1 0 0 1 1 1 1 1 0 0]
 [0 1 1 1 1 1 0 0 0 1]
 [1 0 0 1 0 0 1 0 1 1]
 [1 1 0 1 1 1 0 0 1 0]
 [1 1 0 0 0 0 1 1 1 0]
 [0 0 0 1 0 0 0 0 1 1]]
```

Figure 3: Initial Population

Once we have initialized the population, we write the fitness function for further evaluations. The fitness functions used to evaluate is as follows:

$$fitness = \sum_{i=1}^{n} c_i v_i ; if \sum_{i=1}^{n} c_i w_i \leq kw$$

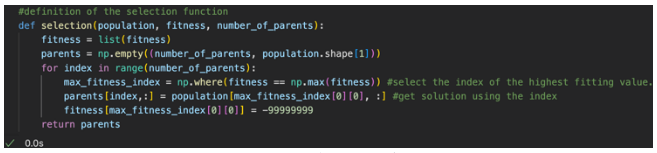
- n = chromosome length
- c_i = ith gene
- v_i = ith value
- w_i = ith weight
- kw = knapsack weight



```
#Definition of the fitness function
def population_fitness(weight, price, population, max_threshold):
    fitness = np.empty(population.shape[0])
    for index in range(population.shape[0]):
        sum_1 = np.sum(population[index] * price) #sum of price to get total price
        sum_2 = np.sum(population[index] * weight) #sum of weight to get total weight
        if sum_2 <= max_threshold:
            fitness[index] = sum_1
        else :
            fitness[index] = 0
    return fitness.astype(int)
✓ 0.0s
```

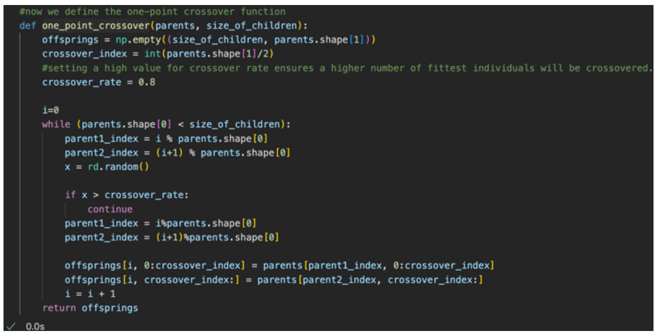
Figure 4: Fitness function

Next following the flowchart, we have the selection, crossover, and mutation function definitions as below:



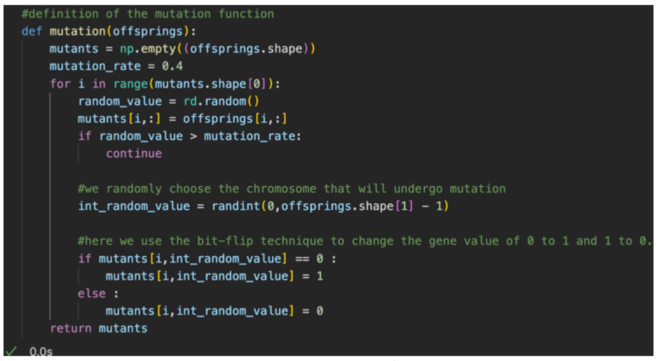
```
#definition of the selection function
def selection(population, fitness, number_of_parents):
    fitness = list(fitness)
    parents = np.empty((number_of_parents, population.shape[1]))
    for index in range(number_of_parents):
        max_fitness_index = np.where(fitness == np.max(fitness)) #select the index of the highest fitting value.
        parents[index,:] = population[max_fitness_index[0][0], :] #get solution using the index
        fitness[max_fitness_index[0][0]] = -99999999
    return parents
✓ 0.0s
```

Figure 5: Selection function



```
#now we define the one-point crossover function
def one_point_crossover(parents, size_of_children):
    offsprings = np.empty((size_of_children, parents.shape[1]))
    crossover_index = int(parents.shape[1]/2)
    #setting a high value for crossover rate ensures a higher number of fittest individuals will be crossovered.
    crossover_rate = 0.8

    i=0
    while (parents.shape[0] < size_of_children):
        parent1_index = i % parents.shape[0]
        parent2_index = (i+1) % parents.shape[0]
        x = rd.random()

        if x > crossover_rate:
            continue
        parent1_index = i%parents.shape[0]
        parent2_index = (i+1)%parents.shape[0]

        offsprings[i, 0:crossover_index] = parents[parent1_index, 0:crossover_index]
        offsprings[i, crossover_index:] = parents[parent2_index, crossover_index:]
        i = i + 1
    return offsprings
✓ 0.0s
```

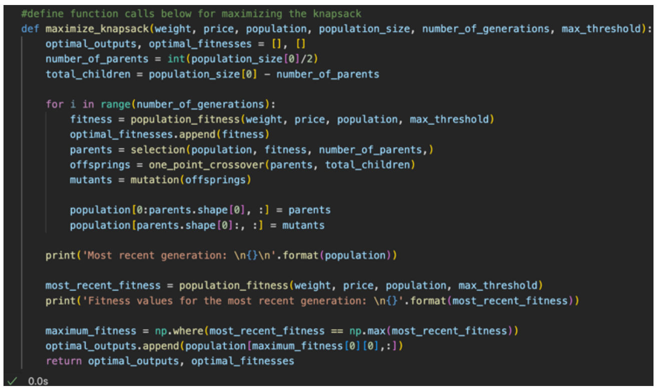
Figure 6: One-Point Crossover



```
#definition of the mutation function
def mutation(offsprings):
    mutants = np.empty((offsprings.shape))
    mutation_rate = 0.4
    for i in range(mutants.shape[0]):
        random_value = rd.random()
        mutants[i,:] = offsprings[i,:]
        if random_value > mutation_rate:
            continue

        #we randomly choose the chromosome that will undergo mutation
        int_random_value = randint(0,offsprings.shape[1] - 1)

        #here we use the bit-flip technique to change the gene value of 0 to 1 and 1 to 0.
        if mutants[i,int_random_value] == 0 :
            mutants[i,int_random_value] = 1
        else :
            mutants[i,int_random_value] = 0
    return mutants
✓ 0.0s
```

Figure 7: Mutation function

Once we define these functions that are used by the genetic algorithm, we define our main method that is used for calling the GA functions in order using all the initializations attached below:



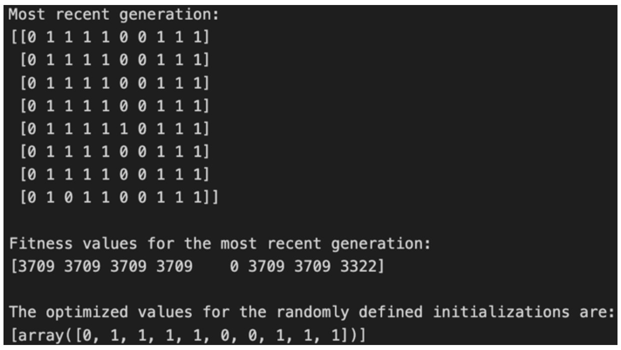
```
#define function calls below for maximizing the knapsack
def maximize_knapsack(weight, price, population, population_size, number_of_generations, max_threshold):
    optimal_outputs, optimal_fitnesses = [], []
    number_of_parents = int(population_size[0]/2)
    total_children = population_size[0] - number_of_parents

    for i in range(number_of_generations):
        fitness = population_fitness(weight, price, population, max_threshold)
        optimal_fitnesses.append(fitness)
        parents = selection(population, fitness, number_of_parents,)
        offsprings = one_point_crossover(parents, total_children)
        mutants = mutation(offsprings)

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

    print('Most recent generation: \n{}\n'.format(population))

    most_recent_fitness = population_fitness(weight, price, population, max_threshold)
    print('Fitness values for the most recent generation: \n{}'.format(most_recent_fitness))

    maximum_fitness = np.where(most_recent_fitness == np.max(most_recent_fitness))
    optimal_outputs.append(population[maximum_fitness[0][0],:])
    return optimal_outputs, optimal_fitnesses
✓ 0.0s
```

Figure 8: Function calls

It is understandable that the above figures may be too small to be able to read and interpret which is why a separate code file is attached to this report. Nevertheless, the code snippets explain the overall structure of the Genetic algorithm used to solve this problem.

IV. Results

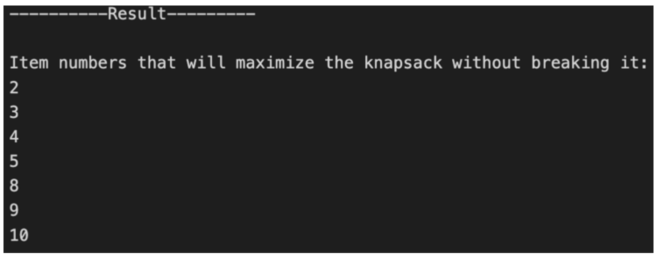
By following the above explained steps in Genetic Algorithm, we achieve the following results:



```
Most recent generation:
[[0 1 1 1 1 0 0 1 1 1]
 [0 1 1 1 1 0 0 1 1 1]
 [0 1 1 1 1 0 0 1 1 1]
 [0 1 1 1 1 0 0 1 1 1]
 [0 1 1 1 1 1 0 1 1 1]
 [0 1 1 1 1 0 0 1 1 1]
 [0 1 1 1 1 0 0 1 1 1]
 [0 1 0 1 1 0 0 1 1 1]]

Fitness values for the most recent generation:
[3709 3709 3709 3709    0 3709 3709 3322]

The optimized values for the randomly defined initializations are:
[array([0, 1, 1, 1, 1, 0, 0, 1, 1, 1])]
```

Figure 9: Fitness History

Using the fitness history, we get the last generation from the population of candidate solutions. This is assumed to be the convergence point. We then use the fitness values for this generation to get the result list of items that maximizes the knapsack value, ensuring it does not exceed the maximum threshold weight.



```
----------Result---------

Item numbers that will maximize the knapsack without breaking it:
2
3
4
5
8
9
10
```

Figure 10: Resultant Item numbers

We now ensure that the results achieved area actually optimal. This step helps us perform a manual crosscheck to ensure the algorithm definition is correct and the results achieved make sense. For the above

achieved optimal item numbers, we have the sum of weights below:
6 + 6 + 2 + 3 + 3 + 12 + 2 = 34,

Which is below the threshold weight of 35 kgs.

We get the 'fittest' most optimal price value for these item numbers as follows:

697 + 387 + 730 + 682 + 372 + 544 + 297 = 3709

We further visualize the fitness values over generations to infer how the optimal value (price) 'evolved' over generations.

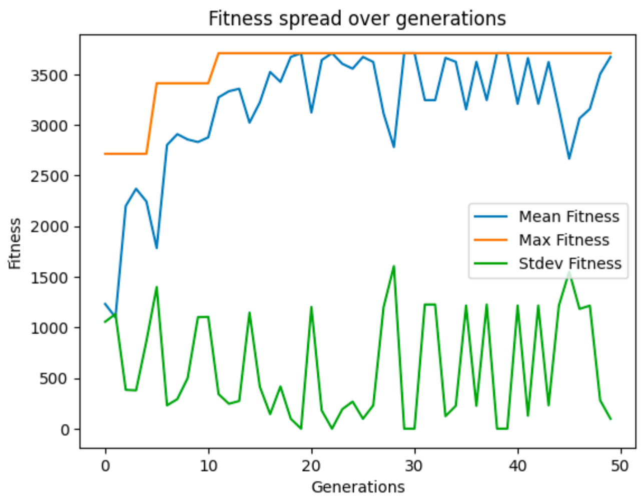


Figure 11: Fitness over Generations

Using the above inferences, we can easily conclude that the selected item numbers produce by the Genetic Algorithm maximize the total price without exceeding the threshold weight of 35kgs.

V. Challenges and Interesting Inferences

One of the most challenging steps in building the solution was the initialization step. Since defining and producing the functions needed for genetic algorithm follow similar ideologies for most problems, they are relatively generic and somewhat easier to infer. Because GA uses an underlying real-world biological structure of natural selection the functions definitions for fitness, selection, crossover, and mutation can be developed in a similar manner for most problem statements. But the initialization step for the knapsack problem was a challenge to understand and implement. The chromosomes or the candidate solutions in the definition of Initial population were represented using a binary vector where 1 represented a selected item and 0 represented an item that was not selected.

The second most challenging step was to understand and define an appropriate fitness function for the binary vector. For the knapsack problem we had two aspects to take care of:

- Ensure the sum of all price values is maximum.
- While the sum of all weight values is below the threshold of 35.

So, we calculated two sums, while ensuring the above definitions were satisfied.

While building the solution for this problem, various interesting inferences were made.

- The real-world biology of 'survival of the fittest' functions in the same manner as an optimization algorithm converging to an optimal solution. Because of this real-world connection this has been one of the most interpretable algorithms for me.
- Another interesting inference that could well be a basis for further future work is how well GA handles Real-world constraints. For example, handling non-linear costs, time-dependent values, and interdependence in items. These are areas that I would like to explore further in my future work.

VI. Conclusions

By way of this report, a crystal-clear understanding of Genetic Algorithm is produced. The initial approach used to solve the knapsack problem by randomly initializing a population using a randomly generated list of items is an important inference. The ideology and overall working behind the core functionality of GA is based on the idea of natural selection. This is implemented using the Fitness function to produce a list of selected (fittest) parents that crossover to produce an offspring. These children are then mutated to randomly add or remove genes, thus in real world most children have some traits of their parents but are not the exact copy of them. Children usually take the best traits of both the parents which is exactly the idea followed to produce the optimal results for the knapsack problem. As stated earlier I aim to further explore GA for time-dependent and more interdependent item values. This will help me further understand how GA can be used with real-world constraints.

References

[1] S. Tiwari, "Genetic Algorithm: Part 3 — Knapsack Problem,", *Medium,* Apr. 28, 2019. https://medium.com/koderunners/genetic-algorithm-part-3-knapsack-problem-b59035ddd1d6

[2] GeeksforGeeks, "Genetic Algorithms," *GeekForGeeks,* Jun. 29, 2017. https://www.geeksforgeeks.org/genetic-algorithms/

[3] Mathworks, "What Is the Genetic Algorithm? - MATLAB & Simulink - MathWorks Australia," https://au.mathworks.com/help/gads/what-is-the-genetic-algorithm.html