DATA STRUCTURES & ALGORITHMS

GENETIC ALGORITHMS

Theoretical and Experimental Analysis with Python Implementation

Fizza Rubab

Yabudullah Ahmed

Mubaraka Shabbir

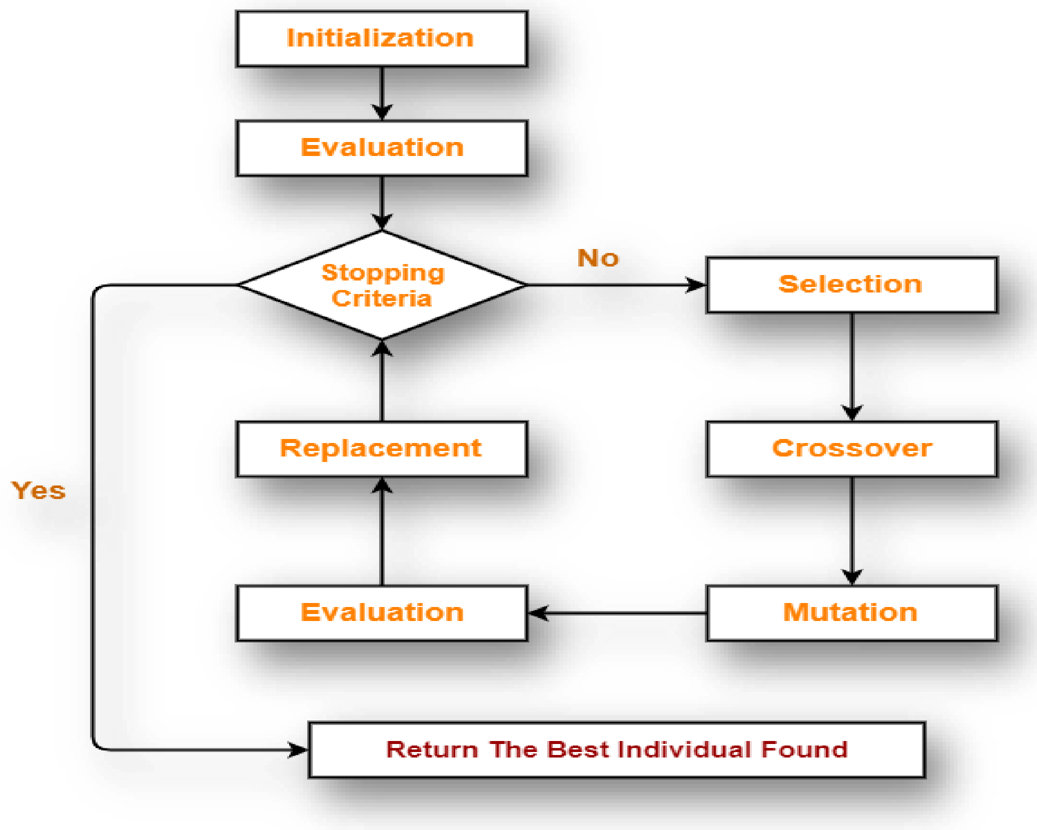
24 June 2020

CONTENTS

Executive Summary of Genetic Algorithms

A **genetic algorithm** is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

Program Flow



How Genetic Algorithm Works

PROJECT OUTCOMES

Our project GENETIC ALGORITHMS: Theoretical and Experimental Analysis with Python Implementation, will provide a deep insight into GAs and their implementation. Following outcomes will be met through the completion of our project:

- Implementation of Genetic Algorithms on real-world optimization problems to find the optimal solution.
- Evaluation of time complexity and order of growth of GAs that is largely dependent on fitness, crossover, selection and mutation operators.
- Comparison and Contrast of GAs performance on the same objective function but deploying different selection and crossover strategies.
- Experimental Analysis and Performance evaluation for multiple datasets by calculating the stable execution time of GA.
- Implementation and manipulation of list data structure (ADT) to represent populations and individuals in the Genetic Algorithm.
- Insight to modular programming used in Genetic Algorithms.

Our project implements the above genetic algorithm objectives on 3 important computing problems in python:

- Travelling Salesperson (TSP) problem to find the smallest route between cities.
- Find the global maxima of a mathematical function over a given domain and range.
- Solve the 0-1 Knapsack Problem to maximize the value of items within the given weight
- constraints.

DATA STRUCTURES INVOLVED

Population-

- List Data Structure

Individuals-

- TSP: Tuples of coordinates (x,y)
- Math Optimization: Dictionaries of coordinates
- Knapsack Problem: list of selected items

LIBRARIES USED

- Numpy Library
- Random Library
- Networx Library
- Matplotlib
- Pandas Library

MODULAR PROGRAMMING

The ability of Genetic Programming to scale to problems of increasing difficulty operates on the premise that it is possible to capture regularities that exist in a problem environment by decomposition of the problem into a hierarchy of modules
Generate Initial Population -> Evaluate fitness-> Select Parent Pool-> Perform Crossover -> Mutate Offspring ->Populate the new generation with offspring

THEORETICAL ANALYSIS

The code implemented uses two different strategies for the terminating of GA:

1. Fixed number of generations.
2. Until Optimal Solution is found

To conduct a theoretical analysis, we will use the number of generations as our termination criteria.

In the analysis the variable factors are

1. The No. of Generations: g
2. Number of Solutions or Population size: n

The fixed/constant elements of GA include:

1. Parent Pool size for Tournament: 4
2. Elite Size: 2

Following is a brief description of time complexity in the implemented **GA using Tournament Selection**.

1. **generate_population(population_size, x_domain, y_domain):**

   This functions generates a list of n dictionaries which contain random x,and y coordinates. Hence complexity is O(n)

```
#Generate random solutions {x,y} within the domain of the function
#and adds them to population
def generate_population(population_size, x_domain, y_domain): # O(n)
    lower_x_boundary, upper_x_boundary = x_domain    # O(1)
    lower_y_boundary, upper_y_boundary = y_domain    # O(1)

    population = []                                  # O(1)
    for i in range(population_size):           #Loop executes for n times. O(n)
        individual = {
            "x": round(random.uniform(lower_x_boundary, upper_x_boundary),5),# O(1)
            "y": round(random.uniform(lower_y_boundary, upper_y_boundary),5) # O(1)
        }
        population.append(individual)                # O(1)

    return population                                # O(1)
```

2. apply_fitness_function(individual):

This function simply accesses the x and y coordinate from the individual dictionary and substitutes it into the equation to return fitness value. O(1).

```
##### OBJECTIVE FUNCTION WE WANT TO MAXIMIZE #####
def apply_fitness_function(individual):     # O(1)
    x = individual["x"]     # O(1)
    y = individual["y"]     # O(1)
    return round((-((x) ** 2 + y ** 2)+5),5)  # O(1)
```

3. tournament_selection(k,sorted_population):

This function generate a parent pool of k constant individuals from the population hence it time complexity is k. Since k is a constant 4, its time complexity is O(1)

```
### SELECTION STRATEGY TO FORM A PINWHEEL AND SELECT A RANDOM NUMBER ###
def tournament_selection(k,sorted_population):      ##O(k) since k is a
constant- the no of elements in the tournamaent pool
    parent_pool=random.sample(sorted_population, k)  # O(k)
    return max(parent_pool, key=apply_fitness_function) # O(k)
```

4. **sort_population_by_fitness(population):**

This uses python built in method to sort population with respect to their fitness value and has time complexity of O(nlogn).

```
def sort_population_by_fitness(population):  # O(nlogn) time complexity of
python's built in sort method
    return sorted(population, key=apply_fitness_function)   # O(nlogn)
```

5. **crossover (individual_1, individual_2)**

This function accesses coordinates of two individual and returns their midpoint. This has a time complexity O(1).

```
### CROSSOVER BY TAKING A POINT IN BETWEEN THE TWO COORDINATES ###
def crossover(individual_1, individual_2):   # O(1)
    x1 = individual_1["x"]  # O(1)
    y1 = individual_1["y"]  # O(1)
    x2 = individual_2["x"]  # O(1)
    y2 = individual_2["y"]  # O(1)
    return {"x":round((x1 + x2)/2,5) , "y":round((y1 + y2) / 2,5)} # O(1)
```

6. **mutate (individual, x_domain, y_domain):**
This function takes a solution and add or subtract a very small value in its existing coordinates which is O(1).

```
########### MUTATE WITHIN THESE BOUNDS ##############
def mutate(individual, x_domain, y_domain): # O(1))
    next_x = individual["x"] + round(random.uniform(-0.15, 0.15),5) # O(1)
    next_y = individual["y"] + round(random.uniform(-0.15, 0.15),5) # O(1)
    # Guarantee we keep inside boundaries
    mutated_x = min(max(next_x, x_domain[0]), x_domain[1]) # O(1)
    mutated_y = min(max(next_y, y_domain[0]), y_domain[1]) # O(1)
    return {"x": mutated_x, "y": mutated_y} # O(1)
```

7. **make_next_generation (previous_population, x_domain, y_domain):**
This function sorts the population first and then combines all of the above functions to generate a population of n individuals (through a loop). The expensive function here is sorting hence its complexity is O(nlogn)

```
def make_next_generation(previous_population, x_domain, y_domain): # 0(nlogn)
    mutation_rate, elite_size=0.2,2 # O(1)
    next_generation = []  # O(1)
    sorted_by_fitness_population = sort_population_by_fitness(previous_population)
#O(nlogn)
    population_size = len(previous_population) # O(1)
    fitness_sum = sum(apply_fitness_function(individual) for individual in
previous_population) #O(n)
    for i in range(population_size-elite_size): #O(n)
        first_choice = tournament_selection(4,sorted_by_fitness_population) #O(k)
        second_choice = tournament_selection(4,sorted_by_fitness_population) #O(k)
        individual = crossover(first_choice, second_choice)  #O(1)
        if random.random()>mutation_rate:  # O(1)
            individual = mutate(individual,x_domain,y_domain)  #O(1)
        next_generation.append(individual)  #O(1)
    next_generation.extend(sorted_by_fitness_population[population_size-elite_size:])
#O(elitesize) -> O(1) since elite_Size is constant
    return next_generation   # O(1)
```

8. **evolve (generations, x_domain, y_domain):**

This final function combines all of the above modules and runs the algorithm for g generations so its complexity becomes O(g*nlogn).

Hence the time complexity of GA which has been implemented using Tournament Selection is O(g*nlogn)

```
def evolve(generations,x_domain, y_domain): #O(g*nlogn)
    i = 1        # O(1)
    progress=[] # O(1)
    population = generate_population(population_size, x_domain, y_domain) #O(n)
    while True:  #O(g)
        print(f" GENERATION {i}")
        flag=False
        for individual in population: #O(n)
            z=apply_fitness_function(individual) # O(1)
            print(individual,z ) # O(1)
        progress.append(apply_fitness_function(max(population,
key=apply_fitness_function))) #O(n))
        if i == generations: # O(1)
            return population, progress # O(1)
        i += 1  # O(1)
        population = make_next_generation(population, x_domain,y_domain) #O(nlogn)
```

Roulette Wheel Selection GA Time complexity varies slightly:

1. **Roulette_Wheel-Selection (sorted_population,fitness_sum):**

This functions calculates relative probability of each individual being selected and returns a random one. More chaces are of the fitter individuals being selected so its complexity is O(n)

```
### SELECTION STRATEGY TO FORM A PINWHEEL AND SELECT A RANDOM NUMBER ###
def choice_by_roulette(sorted_population, fitness_sum): #O(n)
    offset = 0 # O(1)
    normalized_fitness_sum = fitness_sum #total fitness # O(n)
    lowest_fitness = apply_fitness_function(sorted_population[0])# O(1)
    if lowest_fitness < 0:# O(1)
        offset = -lowest_fitness # O(1)
        normalized_fitness_sum += offset * len(sorted_population) # O(1)
    draw = random.uniform(0, 1) #random number# O(1)
    accumulated = 0 # O(1)
    for individual in sorted_population: # O(n)
        fitness = apply_fitness_function(individual) + offset # O(1)
        probability = fitness / normalized_fitness_sum  #fitness ratio # O(1)
        accumulated += probability # O(1)
        if draw <= accumulated:# O(1)
            return individual# O(1)
```

2. **make_next_generation (previous_population, x_domain, y_domain):**

This function sorts the population and then combines all of the above functions to generate a population f n individual. It iterates over to produce n individuals and in each iteration selects a parent son here is sorting hence its complexity is O(n^2).

```
def make_next_generation(previous_population, x_domain, y_domain):# O(n^2)
    mutation_rate, elite_size=0.2,2# O(1)
    next_generation = []# O(1)
    sorted_by_fitness_population = sort_population_by_fitness(previous_population)#
O(nlogn)
    population_size = len(previous_population)# O(1)
    fitness_sum = sum(apply_fitness_function(individual) for individual in
previous_population) # O(n)

    for i in range(population_size-elite_size):# O(n)
        first_choice = choice_by_roulette(sorted_by_fitness_population, fitness_sum)#
O(n)
        second_choice = choice_by_roulette(sorted_by_fitness_population,
fitness_sum)# O(n)
        individual = crossover(first_choice, second_choice)# O(1)
        if random.random()>mutation_rate:# O(1)
            individual = mutate(individual,x_domain,y_domain)# O(1)
        next_generation.append(individual)# O(1)
    next_generation.extend(sorted_by_fitness_population[population_size-
elite_size:])# O(1)
    return next_generation# O(1)
```

3. **evolve (generations,x_domain, y_domain):**

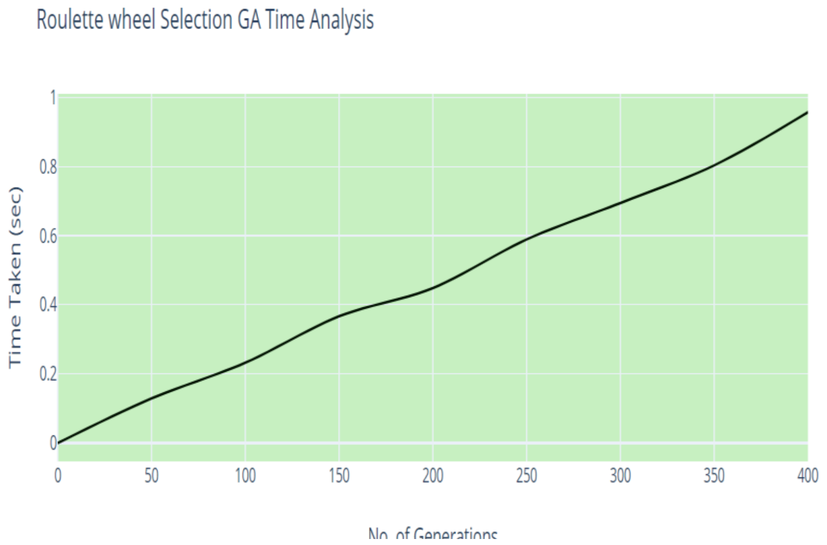
This final function combines all of the above modules and runs the algorithm for g generations so its complexity becomes O(g*n^2).

Hence the time complexity of GA-Roulette Wheel Selection is O(g*n^2)

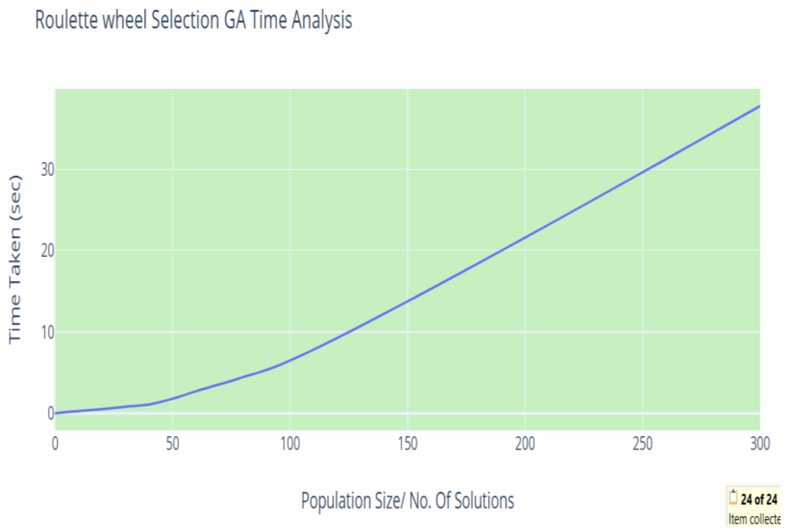
```
def evolve(generations,x_domain, y_domain):# O(g*n^2)
    i = 1# O(1)
    progress=[]# O(1)
    population = generate_population(population_size, x_domain, y_domain)# O(n)
    while True:# O(g)
        print(" GENERATION", i)# O(1)
        flag=False# O(1)
        for individual in population:
            z=apply_fitness_function(individual)# O(1)
            print(individual,z ) # O(1)

progress.append(apply_fitness_function(sort_population_by_fitness(population)[-1]))#
O(1)
        if i == generations: #Termination Criteria
            return population, progress
        i += 1
        population = make_next_generation(population, x_domain,y_domain)# O(n^2)
```

EXPERIMENTAL ANALYSIS

For a constant population size of 10, if we increase the number of generations g time increases linearly which validates our time complexity
Run-Time ∝ G



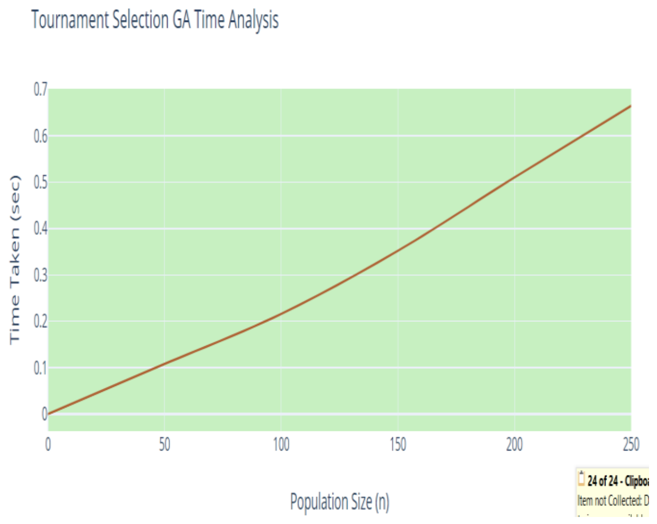
For a constant no. of generations 100, if we increase the population size n, time increases rapidly indicating a n^2 relation which validates our time complexity
Run-Time ∝ N^2



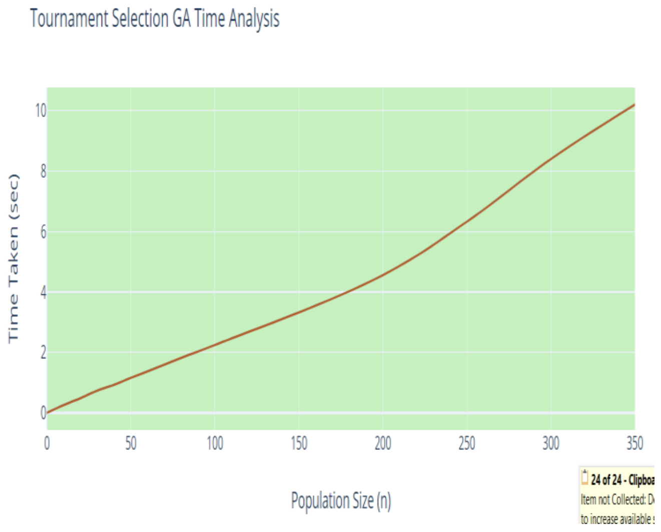
This hints that the Big O complexity of Roulette Wheel Selection has been calculated correctly O(g*n^2)

EXPERIMENTAL ANALYSIS

For a constant population size of 10, if we increase the number of generations g time increases linearly which validates our time complexity
Run-Time ∝ G

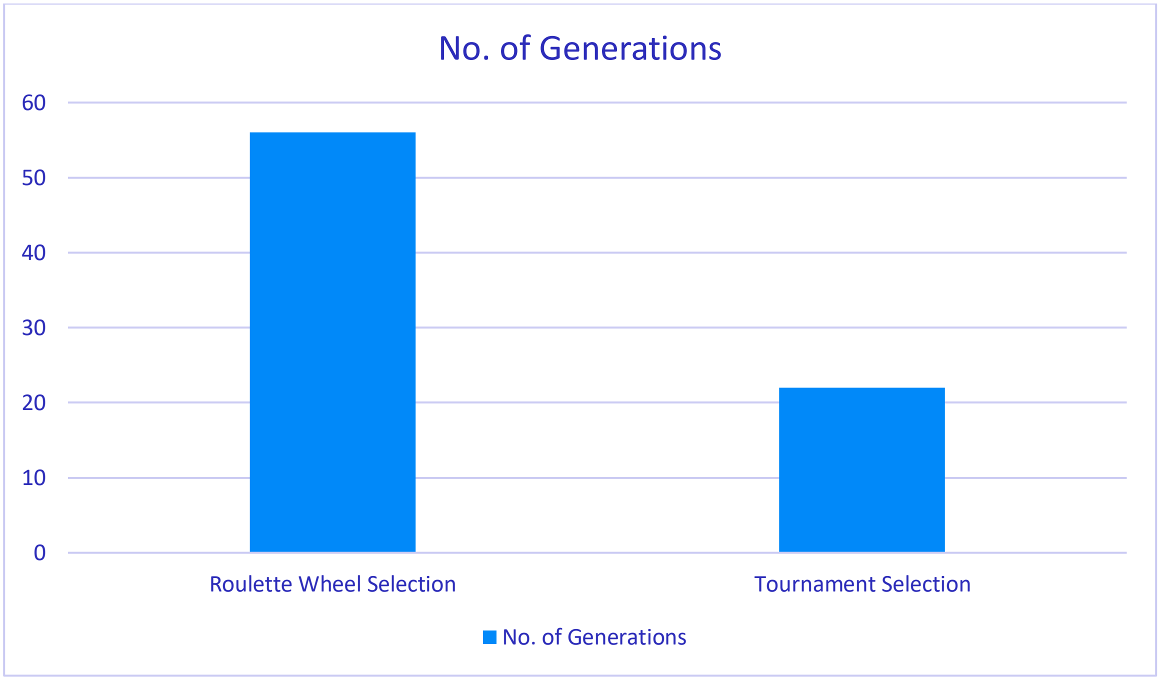


For a constant no. of generations 100, if we increase the population size n, time increases not linearly but with a small rate indicating a possible nlogn relation which validates our time complexity
Run-Time ∝ NlogN



This hints that the Big O complexity of Tournament Selection has been calculated correctly O(g*nlogn)
Thus we can conclude that on the longer run GA with Tournament selection strategy is more efficient.

GA CONVERGENCE WITH DIFFERENT SELECTION METHODS



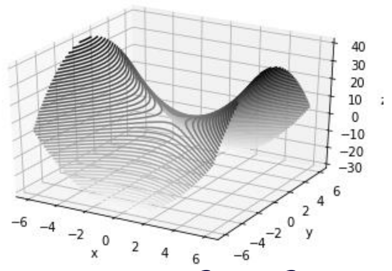
Tournament selection GA converges quickly in an average of 22 generations

While the Roulette Wheel Selection GA converges in 56 generation taking much more time.

ELITISM EFFECT ON GA

Elitism involves copying a small proportion of the fittest candidates, unchanged, into the next generation. This can sometimes have a dramatic impact on performance by ensuring that the EA does not waste time re-discovering previously discarded partial solutions.

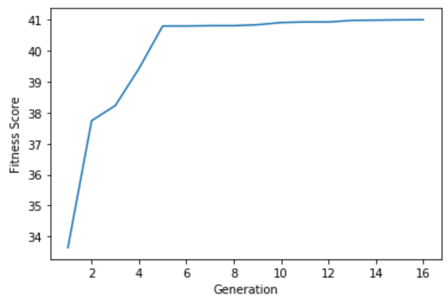
How does elitism affect the convergence of a Algorirthm:

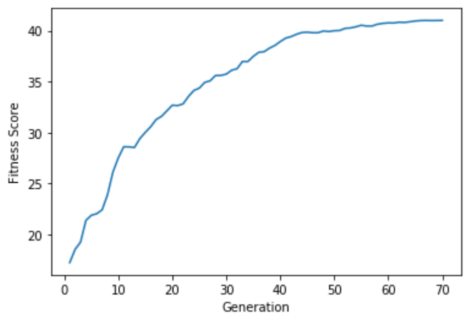


$$f(x, y) = x^2 - y^2 + 5$$

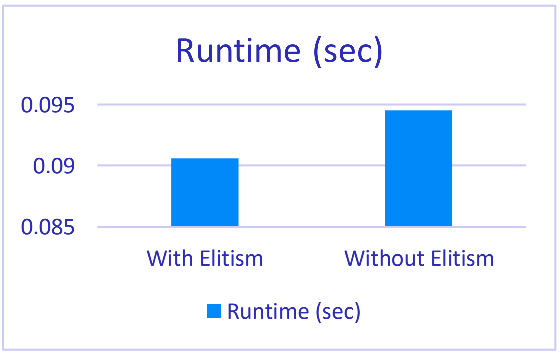
With Elitism, GA preserves fitter solutions and it converges in 16 generations. Time taken is 0.0008054

Without Elitism, we often discard high fitness solutions hence it takes greater time. GA converges in 70 generations. Time taken is 0.002271





Stable run time over 100 iterations 0.0945 without elitism and 0.0906for GA with elitism hence it is 1.04 times faster.



N=size of input

M=size of population

G=number of generations
CODE:

1. Input parameters:

```
  import random as random
from time import time
objects=[(random.randint(0,30),random.randint(0,15))for x in range(0,10)] --------
-O(1)
knapsack_threshold=35 ---------O(1)
no_of_generations=200---------O(1)
```

2. Initialization of population or possible solutions:

```
population=[] ---------O(1)
for i in range(100):--------0(1)
    individuals=[(random.randint(0,1))for x in range(0,len(objects))] -----------
O(N)
    population.append(individuals)  ---------O(1)
```

3. The fitness function:

```
def fitness(individuals,objects,threshold):----------O(N)
    total_weight=0 ---------O(1)
    total_value=0  ---------O(1)
     index=0 ---------O(1)
    for chromosomes in individuals:----------O(N)
        if index>=len(individuals): -----------O(1)
            break          ---------O(1)
       elif chromosomes==1:  ---------O(1)
            total_value+=objects[index][0] -----------O(1)
            total_weight+=objects[index][1] ------------O(1)
        index+=1---------O(1)
    if total_weight<=threshold: ---------O(1)
            return total_value ----------O(1)
    else:                          -----------O(1)
            return 0                 -------------O(1)
```

Explanation for complexity Analysis:

Only a single loop runs in the function for the number of chromosomes that are equal to the number of objects as each chromosome represents the presence/absence of an object. Hence the loop runs for the number of objects and the run-time-complexity of the fitness function is O(N) where N is the number of objects/length of the objects list.

4. The Selection Function:

```
def selection(population,objects,knapsack_threshold,num_of_parents):------------
O(N x M^2)
     lst_of_fitness=[]-----------O(1)
    parents=[]---------O(1)
    for individuals in population:----------O(M)
        fitness_values=fitness(individuals,objects,knapsack_threshold)----------O(N)
        lst_of_fitness.append(fitness_values)-----------O(1)
    for i in range(num_of_parents):-----------O(N)
        for j in range(len(lst_of_fitness)):---------O(M)
            if lst_of_fitness[j]==max(lst_of_fitness):--------O(M)
                max_fitness_idx=j--------O(1)
        parents.append(population[max_fitness_idx])-------O(1)
        lst_of_fitness[max_fitness_idx] = -100000----------O(1)
    return parents---------O(1)
```

Explanation for complexity Analysis:

The first for loop runs for individuals in the population hence it runs M times which is the length of the list of individuals. The fitness function with complexity O(N) is called M times as a result hence the run time is N x M.

The second for loop runs for the number of parents which has been set as (len of objects/2) hence it runs N/2 times. A nested for loop runs for the length of fitness list which is M times and a max function is called for the list of fitness inside the nested loop which has the complexity O(M) hence the total run time is N/2 x M^2

Adding both run times of the for both the for loops we get $(N/2)M^2 + NM$ hence through asymptotic analysis we get the Big O complexity of the function to be $O(N \times M^2)$

5. The Cross-over Function:

```
def crossover(parents): ------------O(N)
    offsprings=[]--------O(1)
    cross_over_rate=0.8---------O(1)
    crossover_point=len(parents[0])//2---------O(1)
    i=0------O(1)
```

```
        no_of_offsprings = len(objects)-len(parents)----------O(1)
        while len(offsprings) < no_of_offsprings :---------O(N)
            random_value=random.random()---------O(1)
            if random_value > cross_over_rate: -----------O(1)
                continue---------O(1)
            else:        ----------O(1)
                parent1_index = i%len(parents)--------O(1)
                parent2_index = (i+1)%len(parents)---------O(1)

    offspring=parents[parent1_index][0:crossover_point]+parents[parent2_index][cro
    ssover_point:]   -------O(1)
                offsprings.append(offspring)--------O(1)
            i+=1-------O(1)
        return offsprings--------O(1)
```

Explanation for complexity Analysis:

The function contains a single while loop which runs (number of offsprings-1) times. Since the number of offsprings is len(objects)/2=N/2, the loop runs (N/2)-1 times and the complexity of the function is O(N)

6. The Mutation Function:

```
    def mutate(offsprings):------------O(N)
        mutation_rate=0.4--------O(1)
        mutants =[]--------O(1)
        for i in range(len(offsprings)):--------O(N)
            random_value = random.random()----------O(1)
            mutants = offsprings[i]-----------O(1)
            if random_value > mutation_rate:--------O(1)
                continue-------O(1)
            index=random.randint(0,len(offsprings[i])-1)--------O(1)
            if mutants[index] == 0 :----------O(1)
                mutants[index] = 1----------O(1)
            else : ----------O(1)
                mutants[index] = 0 -------O(1)
        return offsprings ---------O(1)
```

Explanation for complexity Analysis:

The function has a single loop which runs (N/2)-1 times which is the length of the list of offspring hence the functions complexity is O(N).

7. The Optimization/Main Function:

```
def knap_sack_problem():---------O(G x N x M^2)
  for i in range(no_of_generations):-----------O(G)
        num_of_parents=len(objects)//2--------O(1)
        parents = selection(population,objects,knapsack_threshold,num_of_parents)-----
--O(N x M^2)
        offsprings = crossover(parents)---------O(N)
        mutants = mutate(offsprings)---------O(N)
        population[num_of_parents:]=mutants[0:]------O(1)
        population[0:num_of_parents]=parents[0:]--------O(1)

    return population--------O(1)
print(knap_sack_problem())-------O(1)
```

Explanation for complexity Analysis:
The optimization function or the main function call a series of functions, number of generations (G) times and adding up the complexities of all the functions called into the number of generations gives us the complexity of the Knap sack function to be O(G x N x M^2) which is:
Number of generations x number of objects/size of input x size of population/number of possible solutions

8. A function for decoding the solution obtained through the Main Function:

```
def decoder():------O(G x N x M^2)
  lst=[]------O(1)
  value=0------O(1)
  final_lst=knap_sack_problem()------O(G x N x M^2)
  for i in range(len(final_lst[0])):------O(N)
      if final_lst[0][i]==1:---------O(1)
         value+=objects[i][0]-------O(1)
         lst.append(objects[i])-------O(1)
  return value,lst--------O(1)
```

The complexity of the overall code remains to be O(G x N x M^2) hence the complexity of Genetic Algorithm depends on these three factors:

- Number of Generations ‘G’
- The size of the input ‘N’

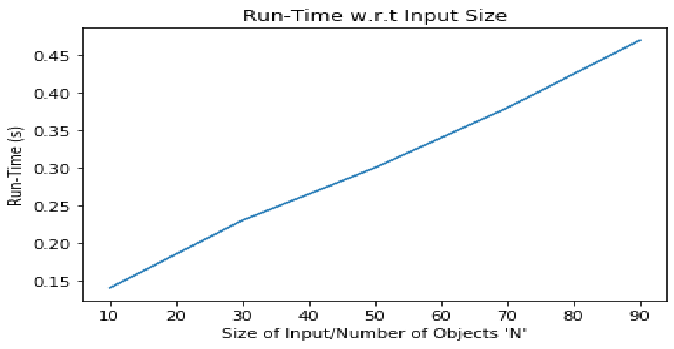
- The size of the population ‘M’

EXPERIMENTAL ANALYSIS

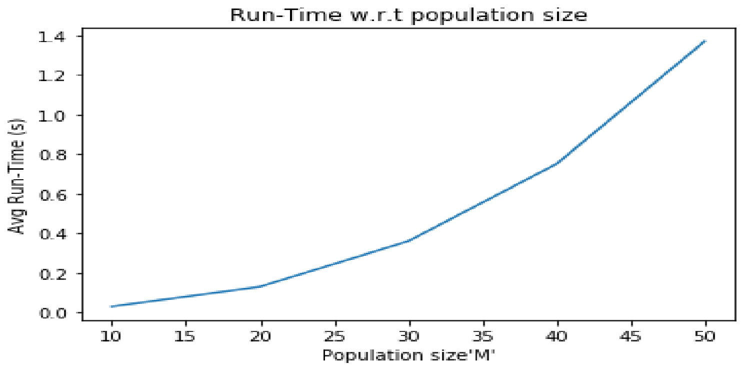
The obtained complexity shows that the Algorithm's run time depends on three factors:

N=size of input
M=size of population
G=number of generations

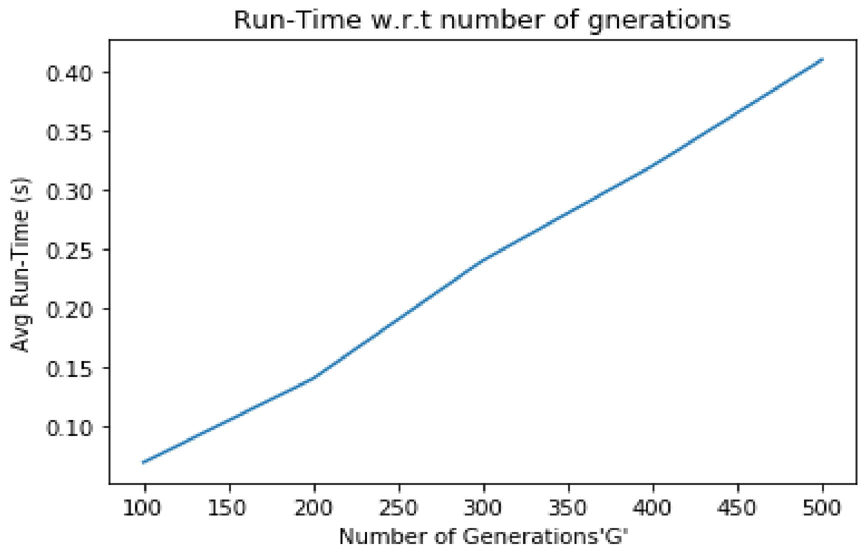
- If we keep the number of generations and the size of the population constant at G=100 and M=20:
  The graph of average run time w.r.t the size of input is almost a straight line which shows that run time is linearly dependent on the input size ‘N’ i.e Run-Time ∝ N:



- If we keep number of generations and Input size constant at G=100 and N=10:
  The graph of average run time w.r.t the population size appears to be parabolic indicating that Run-Time ∝ M^2:



- If we keep the Input size and the population size constant at N=10 and M=20:
  The graph of average run time w.r.t the number of generations appears to be linear which follows that the Run-Time $\propto$ G:



The Experimental Analysis also leads us to the verification of the obtained Run-Time complexity:

The complexity obtained is O(G x N x M^2):

For it to hold true the following conditions need to be met:

I. When G and N are constant, run time depends on the square of the size of population hence the graph of run time w.r.t M should be parabolic.

II. When N and M are constant, the run time depends on G linearly so the graph of runtime w.r.t G should be linear.

III. When G and M are constants, run time depends on N linearly so the graph of run time w.r.t N should be linear.

All three conditions are met when the algorithm goes through Experimental Analysis as shown above hence the Big O complexity O(G x N x M^2) is verified and is the complexity of the Genetic Algorithm for solving the knap sack problem.

THEORETICAL ANALYSIS

n = input size
m = population size
g = number of generations

Generating Input:

```
def generate_input(N): #---------------------- O(n)
    cityList=[]
    for i in range(0,N):
        x=int(random.random()*200)
        y=int(random.random()*200)
        cityList.append((x,y))
    return cityList
```

Initial Population:

```
def createRoute(cityList): #---------------------------------------O(n)
    route = random.sample(cityList, len(cityList)) #---- n
    return route


def initialPopulation(popSize, cityList): #------------------------O(n*m)
    population = []
    for i in range(0, popSize): #---- m
        population.append(createRoute(cityList)) #--- m*n
    return population
```

Explanation: These functions simply create the initial population, the initialPopulation functions runs a loop for m times and calls on the createRoute function which has complexity n thus having total complexity n*m

Fitness:

```
def distance(citylist,city1, city2): #--------------------- O(1)
        xDis = abs(city1[0] - city2[0])
        yDis = abs(city1[1] - city2[1])
        distance = math.sqrt((xDis ** 2) + (yDis ** 2))
        return distance


def routeDistance(route): #----------------------------------O(n)
    pathDistance=0
    for i in range(0,len(route)): #----n
        fromCity = route[i]
        if i+1 < len(route):
            toCity=route[i+1]
        else:
            toCity=route[i]
        pathDistance += distance(route,fromCity,toCity)#---1
    return pathDistance


def routeFitness(route): #-----------------------------------O(n)
    fitness= 1 / float(routeDistance(route))#---n
    return fitness


def rankRoutes(population): #-------------------------------------------O(mn)
    fitnessResults={}
    for i in range(0,len(population)): #----m
        fitnessResults[i]= routeFitness(population[i]) #---- m*n

    return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse = True)
```

Explanation: The first loop in the rankRoutes function runs for m times which calls routeFitness function which simply gives the inverse value that the routeDistance function returns that has time complexity of n, giving total complexity of m*n

Selection:

```
def selection(popRanked, eliteSize): #-------------------------------------O(nm+m^2)
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index","Fitness"])#----- 1
    df['cum_sum'] = df.Fitness.cumsum()
    df['cum_perc'] = 100*df.cum_sum/df.Fitness.sum()

    for i in range(0, eliteSize): #------------------------------ m for both loops
        selectionResults.append(popRanked[i][0])
    for i in range(0, len(popRanked) - eliteSize): #------mn
        pick = 100*random.random() #------m*n
        for i in range(0, len(popRanked)):#------m*m
            if pick <= df.iat[i,3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults
def matingPool(population, selectionResults): #--------- O(n)
    matingpool = []
    for i in range(0, len(selectionResults)):
        index = selectionResults[i]
        matingpool.append(population[index])
    return matingpool
```

Explanation: The matingPool function simply runs a loop for n times. The selection function first creates a dataframe, then runs 2 loops for m times, the second loop calls a random function which has complexity n and another loop inside it which also runs m times, giving total complexity nm+m^2, O(nm^2)

Breeding:

```
def breed(parent1, parent2): #---------------------------------- O(n)
    child = []
    childP1 = []
    childP2 = []
    geneA = int(random.random() * len(parent1))#---n
    geneB = int(random.random() * len(parent1))#---n
    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)
    for i in range(startGene, endGene): #-------------- n
        childP1.append(parent1[i])
    childP2 = [item for item in parent2 if item not in childP1] #---n
    child = childP1 + childP2
    return child

def breedPopulation(matingpool, eliteSize): #---------------------O(n)
    children = []
    length = len(matingpool) - eliteSize #**
    pool = random.sample(matingpool, len(matingpool))
    for i in range(0,eliteSize): #-------------n for both loops
        children.append(matingpool[i])
    for i in range(0, length):
        child = breed(pool[i], pool[len(matingpool)-i-1]) #---- n
        children.append(child)
    return children
```

Explanation: both functions, breed and Breeding simply run loops n times giving compleity O(n)

Mutation:

```
def mutate(individual, mutationRate): #--------------------------O(n)
    for swapped in range(len(individual)): #-------n
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))
            city1 = individual[swapped]
            city2 = individual[swapWith]
            individual[swapped] = city2
            individual[swapWith] = city1
    return individual
```

```
def mutatePopulation(population, mutationRate): #------------------O(n)
    mutatedPop = []

    for ind in range(0, len(population)): #---- n
        mutatedInd = mutate(population[ind], mutationRate)
        mutatedPop.append(mutatedInd)
    return mutatedPop
```

Explanation: both functions, mutate and mutatePopulation simply run loops n times giving compleity O(n)

Main Functions:

```
def nextGeneration(currentGen, eliteSize, mutationRate): #-----------(nm+m^2)
    popRanked = rankRoutes(currentGen)#-------------------mn
    selectionResults = selection(popRanked, eliteSize) #------------(nm+m^2)
    matingpool = matingPool(currentGen, selectionResults) #-----------n
    children = breedPopulation(matingpool, eliteSize) #--------------n
    nextGeneration = mutatePopulation(children, mutationRate)#-----n
    return nextGeneration
```

```
def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):#-------O(g*n*m^2)
    pop = initialPopulation(popSize, population) #-----------------n
    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1])) #-----------mn

    progress = []
    progress.append(1 / rankRoutes(pop)[0][1]) #------------------mn

    for i in range(0, generations): #------------------------g
        pop = nextGeneration(pop, eliteSize, mutationRate) #--------n*m^2
        progress.append(1 / rankRoutes(pop)[0][1])

    print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))#--mn
    bestRouteIndex = rankRoutes(pop)[0][0]#---mn
    bestRoute = pop[bestRouteIndex]
```

Explanation: The nextGeneration simply calls all other helping functions and has complexity (nm+m^2) due to calling on selection function. The geneticAlgorithm functions calls on the nextGeneration function g times, which is the number of generations we give the function for stopping critera, thus the total complexity of this function is O(g(nm+m^2))

EXPERIMENTAL ANALYSIS:

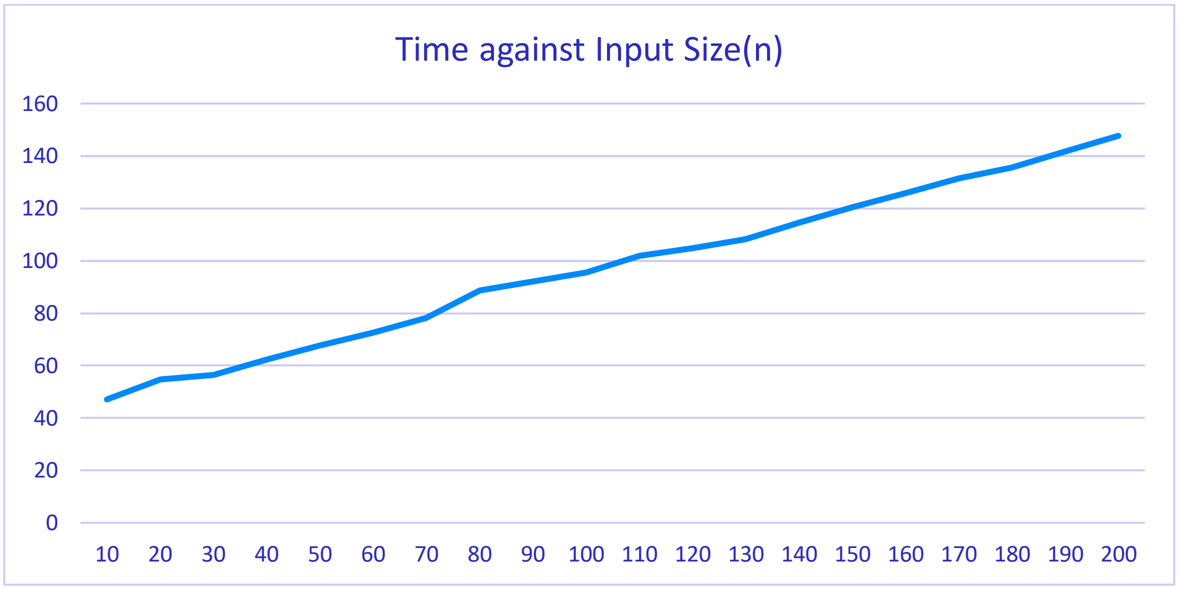
Recall that:

n = size of input

m = population size

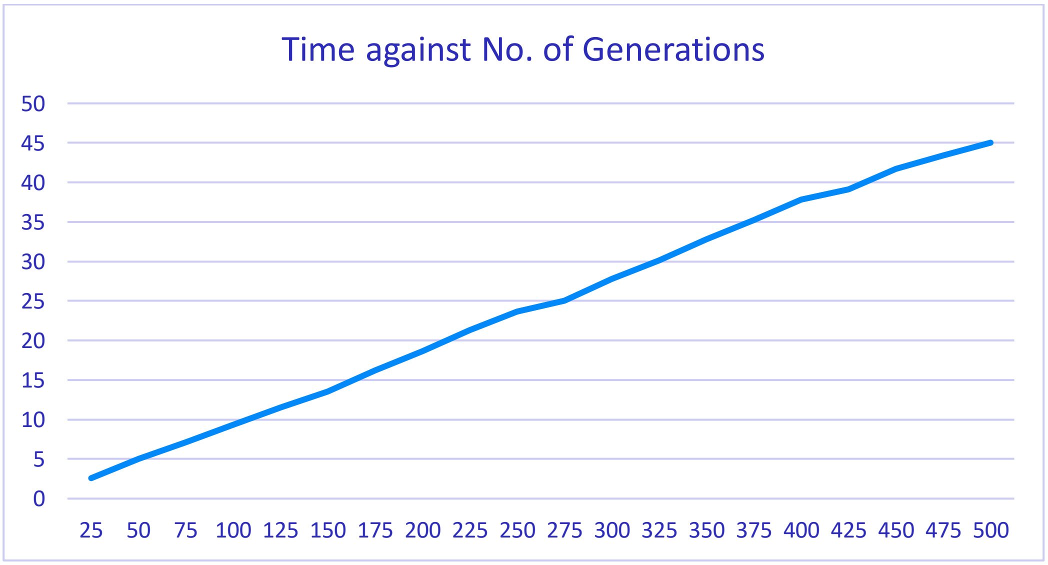
g= no. of generations

- Keeping population size constant at 10 and no. of generations 500, but varying n we get the following graph:



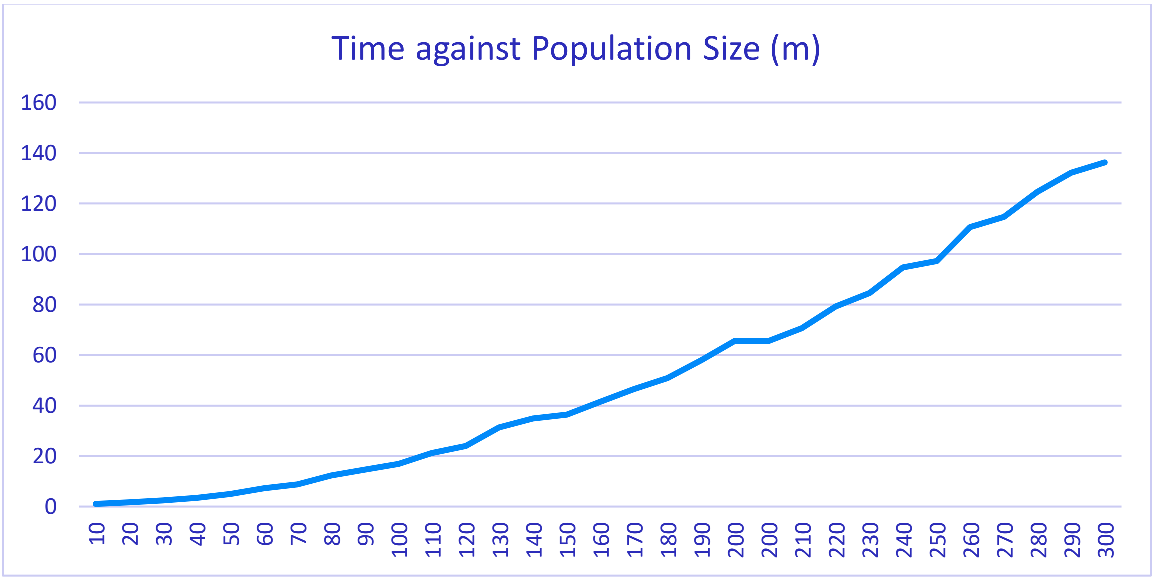
.
The graph shows that the time complexity of the function increases with n in a linear fashion.

- Keeping population size constant at 10, and input size at 10, but varying g we get the following graph:



The graph shows that the time complexity of the function increases with g in a linear fashion.

- Keeping input size constant at 10 and no. of generations at 150, but varying m we get the following graph



The graph shows that the time complexity of the function increases with m in a parabolic fashion.

CONCLUSION:

The Experimental Analysis varifies our theoretical calculation of the time complexity as

$$\boldsymbol{O}(\boldsymbol{g}(\boldsymbol{nm} + \boldsymbol{m}^2))$$

REFERENCES

https://medium.com/koderunners/genetic-algorithm-part-3-knapsack-problem-b59035ddd1d6

https://github.com/edmilsonrobson/0-1-Knapsack-Problem-with-Genetic-Algorithms/blob/master/main.py

https://hackernoon.com/genetic-algorithms-explained-a-python-implementation-sd4w374i

https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35