AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING

Senior 2 - Mechatronics Engineering Dep.
Specialized Programs – Spring 2023

Engineering Optimization

MCT434

Project

"Service Towers Distribution"

TEAM (12)

Name	ID
Mostafa Ahmed Mahmoud Mohamed Qusit	*1803215*
Mostafa Ashraf Elsayed Elsayed Attia	*1806727*
Samy Gamal Mahmoud Moustafa	*1801300*
Samy Ayman Mosaud Elshourbagy	*1803380*
Abdullah Eid Abd-Elmenam Balek	*1902339*
Mohammed Atef Ramadan Abdelfattah Elfeky	*1808649*

Table of Contents

Table of Figures

1 Introduction

With the spread of networks and means of communication around the world, the demand for providing these services has increased on a large scale, which raises a problem on the surface, the methods of distributing the sources (towers) in the regions to reach the greatest benefit without prejudice to the limited resources, as well as sensitive areas where it is forbidden to connect these networks for military reasons .

2 Problem Formulation

Given **a limited map** get the best distribution for certain **number of service towers** with certain **covering range** (circle shape) to minimize the **uncovered area** with get to account the **Forbidden zones** that must reach no service to them.

Figure 1 - Problem Visualization

2.1 Objective Function

Maximize the function is the Total Area that covered from the towers except the forbidden zone.

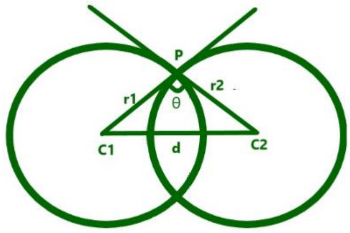
$$\boldsymbol{Obj.function} = \boldsymbol{A_{towers}} + \boldsymbol{A_{forbidden\ zones}}$$

$$= \left[\sum_{t=1}^{T} A_t - \sum_{t_1=1}^{T-1}\left(\sum_{t_2=t_1+1}^{T} IA(t_1, t_2)\right)\right] + \left[\sum_{f=1}^{F} A_f - \sum_{f_1=1}^{F-1}\left(\sum_{f_2=f_1+1}^{F} IA(f_1, f_2)\right)\right]$$

$where: \ T \equiv No.of\ towers.$

$F \equiv No.of\ forbidden\ zones.$

$IA \equiv Intersection\ area\ between\ 2\ circles:$



$$\alpha = 2\cos^{-1}\left(\frac{{r_1}^2 + d^2 - {r_2}^2}{2\,r_1 d}\right), \quad \beta = 2\cos^{-1}\left(\frac{{r_2}^2 + d^2 - {r_1}^2}{2\,r_2 d}\right)$$

$$a_1 = \frac{1}{2}{r_2}^2\,\beta - \frac{1}{2}{r_2}^2\sin(\beta), \qquad a_2 = \frac{1}{2}{r_1}^2\,\alpha - \frac{1}{2}{r_1}^2\sin(\alpha) \quad \rightarrow \quad IA = floor(a_1 + a_2)$$

2.2 Decision Variables

Our variables are the position of all service towers in x and y.

x_1	y_1	x_2	y_2	x_3	y_3	$\cdots$	$\cdots$	x_T	y_T

2.3 Constraints

The constraints that if solution step over them, it will be re-randomized again and again until be within those constraints.

- <u>Constraint on tower position:</u>

$$r_t < x_t < (L_{map} - r_t) \qquad , \qquad r_t < y_t < (W_{map} - r_t)$$

$$where: \ r \equiv radius\ of\ the\ circular\ tower\ range.$$

- <u>Constraint for the forbidden zones:</u>

If the any tower range intersects with forbidden zone

3 Algorithms

3.1 Simulated Annealing (SA)

an optimization method which mimics the slow cooling of metals, which is characterized by a progressive reduction in the atomic movements that reduce the density of lattice defects until a lowest-energy state is reached.

3.1.1 Codes

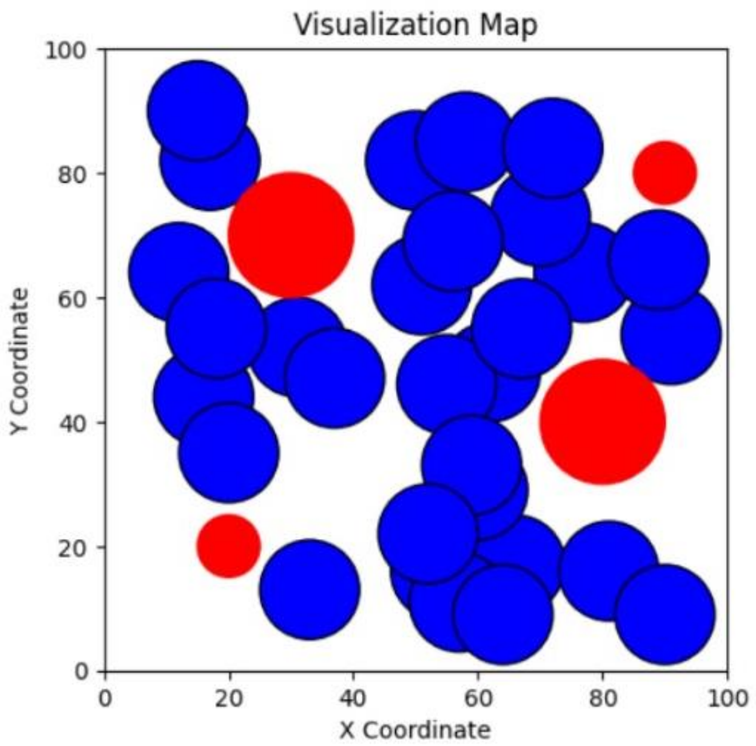
```
while (T > Tf and i <= i_max):
    for n in range(N):
        # Generate new solution:
        new_towers = towers.copy()
        for t in range(len(towers)):
            new_towers[t].add_randomize(scaler)

            # Feasibility Check:
            while new_towers[t].feasibility_check() != True:
                new_towers[t].randomize()

        # Compute change in energy:
        Delta_E = objective_function(towers) - objective_function(new_towers)
        if Delta_E < 0:
            towers = new_towers.copy()
        else:
            acceptance_probability = np.exp(-1.0*Delta_E/T)
            if acceptance_probability > 0.75:   #random.random()
                towers = new_towers.copy()
            else:
                pass # do nothing

        # Store the Best Solution in scope of ALL Solutions
        if objective_function(towers) > objective_function(Best_towers):
            Best_towers = towers.copy()

    # Update the Temperature and iteration number
    if   cooling_rate == 'linear'    : T = Ti - Beta * i
    elif cooling_rate == 'Geometric': T = Ti * pow(alpha, i)
    i += 1
```

3.1.2 Results

@ no. of towers = 30 , tower radius = 8

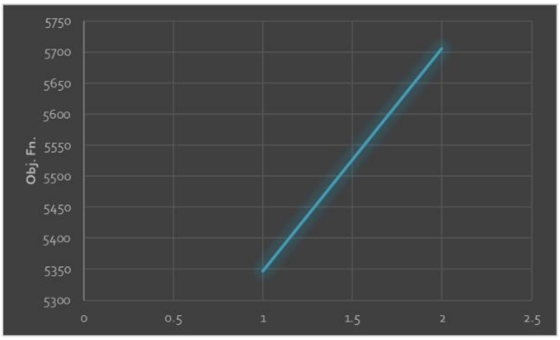
No.	$T_{initial}$	T_{final}	Iterations	Acceptance Prob.	Cooling Rate	Cost
1	100	0.001	100	0.9	Linear	**5346.65**
2	100	0.001	1000	0.9	Linear	**5526.65**
3	100	0.001	500	0.7	Linear	**5956.65**
4	**1000**	**0.001**	**500**	**0.7**	**Linear**	**6055.65**
5	100	0.001	100	0.9	Geometric	**5705.65**
6	100	0.001	500	0.9	Geometric	**5930.65**

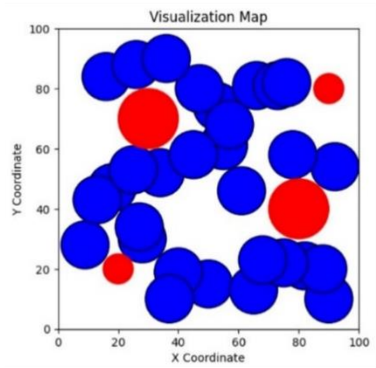
- **Best Result:**



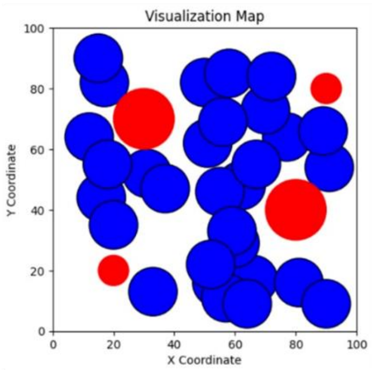
❖ Parameters Effect:

- Initial Temperature



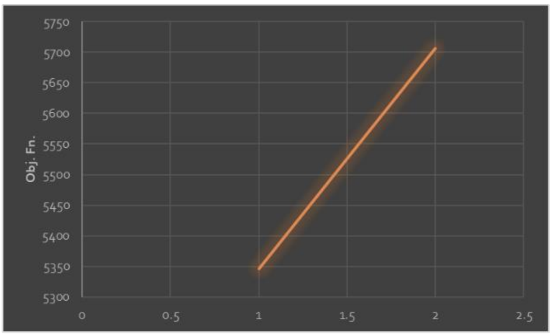


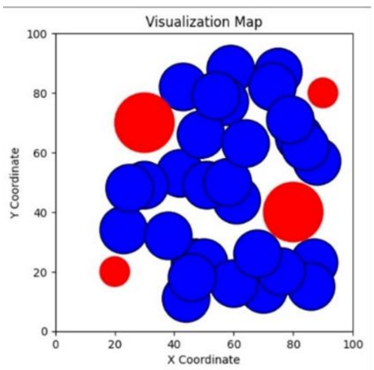
Ti = 100



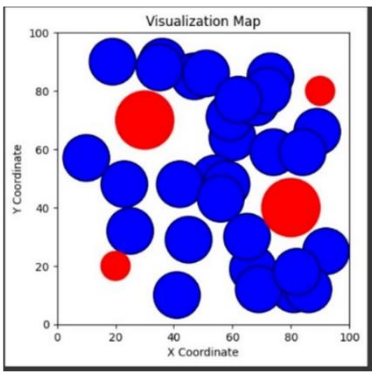
Ti = 1000

- No. of Iterations



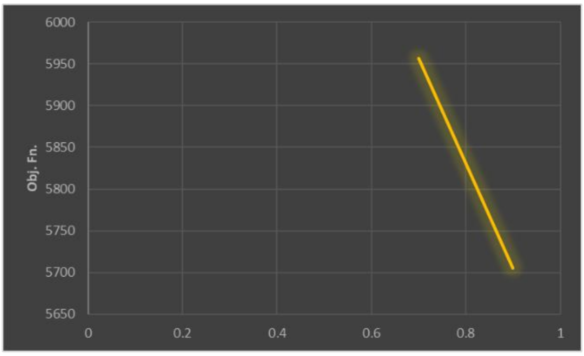


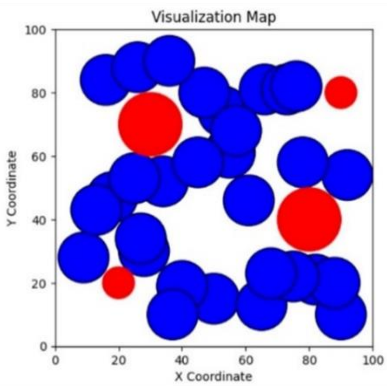
N = 100

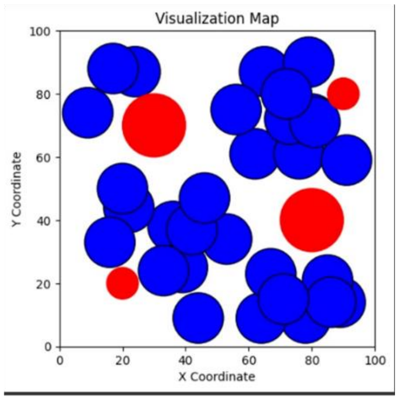


N = 1000

- Acceptance Probability

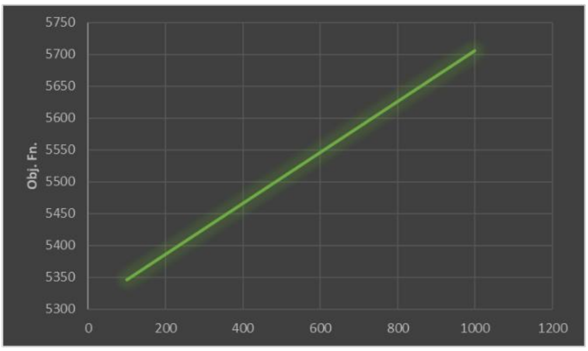


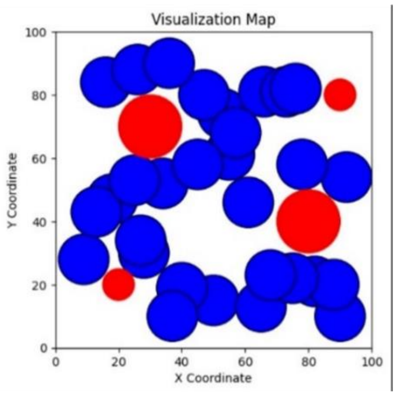


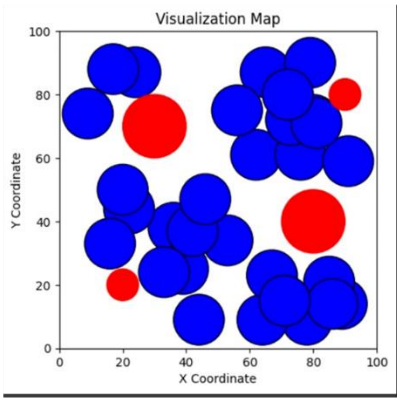


AP = 0.7 AP=0.9

- Cooling Rate







Linear Geometric

3.2 Genetic Algorithm (GA)

is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions.

3.2.1 Codes

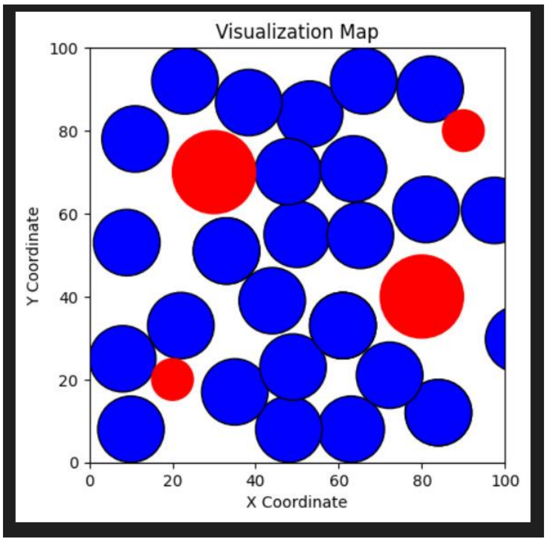
```
for generation in range(generation_numbers):
    if generation == 0:
        io = list(np.argsort(old_fitness_values))   # Sort the members

    # Elitisim Stage:
    for e in range(0, elite_size):
        new_towers_population[io[e]] = old_towers_population[io[e]].copy()

    # Cross-Overing Stage:
    for co in range(elite_size+1, elite_size+CrossOver_size+1, 2):
        parent1 = old_towers_population[io[random.randrange(0, elite_size    )]].copy()
        parent2 = old_towers_population[io[random.randrange(0, towers_PopSize)]].copy()

        child1 = parent1.copy()
        child2 = parent2.copy()
        for t in range(towers_number):
            child1[t].x = (alpha)*parent1[t].x + (1-alpha)*parent2[t].x
            child1[t].y = (alpha)*parent1[t].y + (1-alpha)*parent2[t].y

            child2[t].x = (1-alpha)*parent1[t].x + (alpha)*parent2[t].x
            child2[t].y = (1-alpha)*parent1[t].y + (alpha)*parent2[t].y
        # Feasibility Check:
        while feasibility_check(child1) != True:    randomize(child1)
        while feasibility_check(child2) != True:    randomize(child2)

        new_towers_population[io[co  ]] = child1.copy()
        new_towers_population[io[co+1]] = child2.copy()

    # Mutation Stage:
    for m in range(elite_size+CrossOver_size+1, members_number):
        add_randomize(new_towers_population[io[m]], scaler)
        # Feasibility Check:
        while feasibility_check(new_towers_population[io[m]]) != True:
            randomize(new_towers_population[io[m]])

    # get the fitness for the new generation:
    new_fitness_values = [objective_function(new_towers_member) for new_towers_member in new_towers_population]

    # Update the Population and their fitness values:
    old_towers_population = new_towers_population.copy()
    old_fitness_values = new_fitness_values.copy()

    io = list(np.argsort(old_fitness_values))   # Sort the members

    # Store the Best Solution in scope of ALL Generations:
    if objective_function(old_towers_population[io[0]]) < objective_function(Best_towers):
        Best_towers = old_towers_population[io[0]].copy()
```

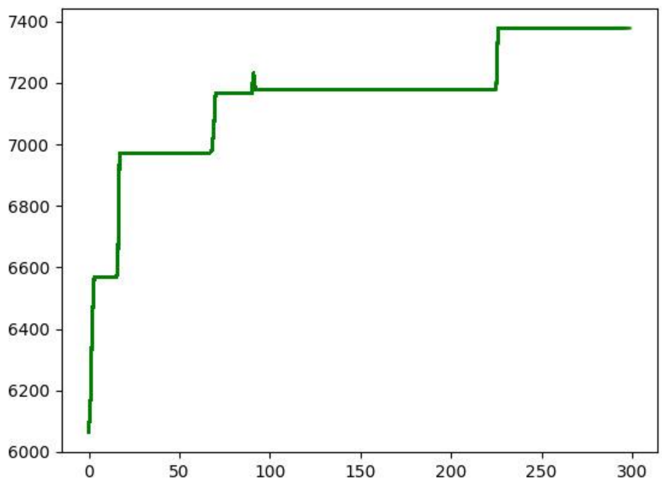
3.2.2 Results

@ no. of towers = 30 , tower radius = 8

No.	Pop Size	Gene. Size	Elite Ratio	Mut. Ratio	Cost
1	20	200	0.33	0.33	**7350.00**
2	20	300	0.40	0.20	**7155.00**
3	3	300	0.33	0.33	7390.00

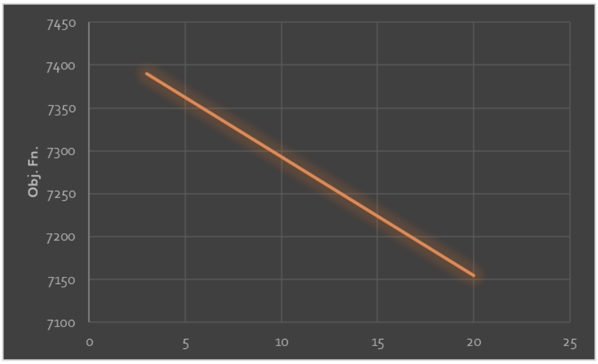
❖ **Best Result:**

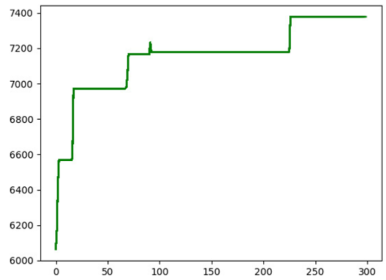




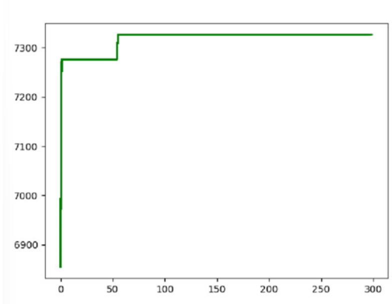
❖ Parameters Effect:

- **Population Size**



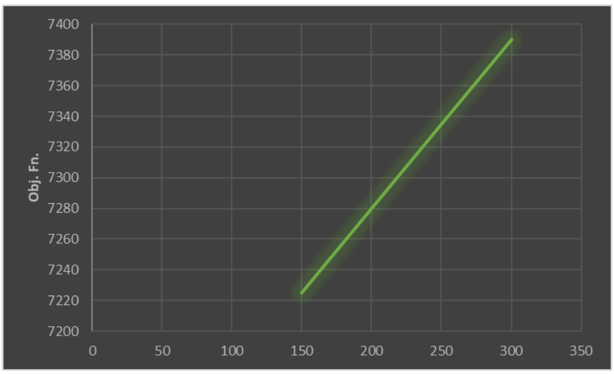


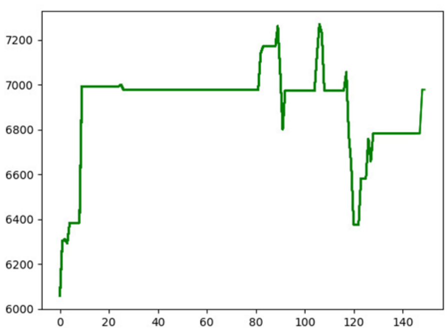
PS = 3



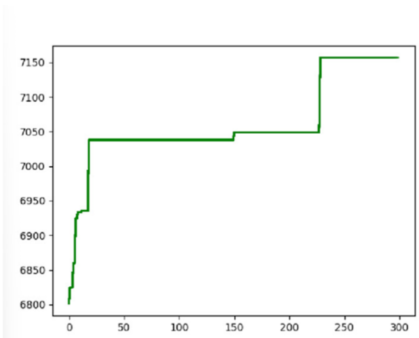
PS = 20

- **Generation Size**



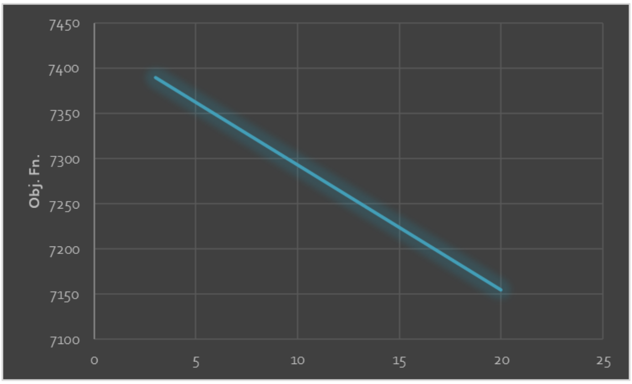


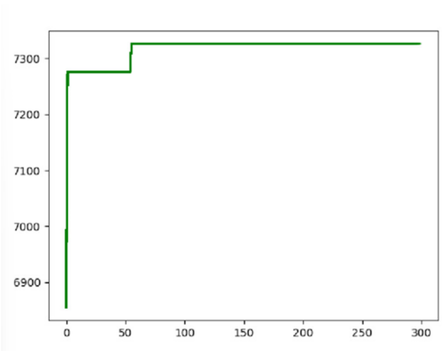
N = 150



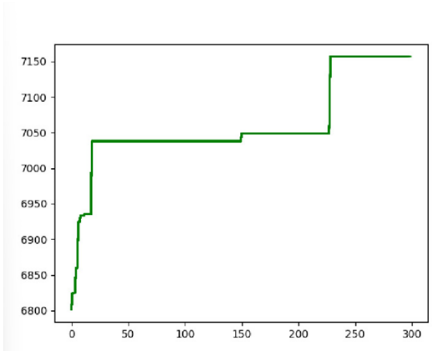
N = 300

- **Elite & Mutation Ratios**





E=0.33, M=0.33



E=0.4, M=0.20

3.3 Particle Swarm Optimization (PSO)

is a stochastic optimization technique based on the movement and intelligence of swarms. In PSO, the concept of social interaction is used for solving a problem. It uses several particles (agents) that constitute a swarm moving around in the search space, looking for the best solution.

3.3.1 Codes

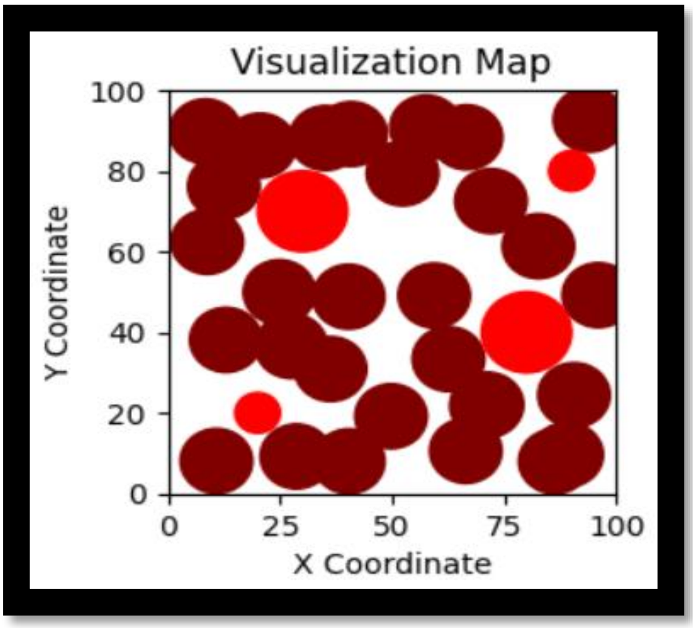
```
#Step(1): Generate First Generation :

Particles_Coordinates = np.random.randint(0+TowerR, 100-TowerR, [Swarm,No_of_Towers,2])
Particles_Velocities = np.zeros([Swarm,No_of_Towers,2])

Personal_Best_fit = np.zeros([Swarm]) #Personal Values for each Particle
Personal_Best_Particles = np.zeros([Swarm,No_of_Towers,2]) #Personal Coordinates of Towers for each Particle

Obj_Fn_Records = np.zeros([Max_Num_Generation])

#Feasibility Check for each Particle in the Swarm
for mem_No in range(0,Swarm):
  Feasiblity_Correction(mem_No)

for Generation_Counter in range(0,Max_Num_Generation):

  #Step(2): Calculate Fitness Function of All Members Of the Generation:
  fitnesses= [0]*Swarm
  for mem_No in range(0,Swarm):
    #X_Coordinates = Particles[mem_No,:,0]
    #Y_Coordinates = Particles[mem_No,:,1]
    fitnesses[mem_No] = Particle_obj_fn(mem_No)
#------------------------------------------------------------------------------#

  #Step(3): Update the Personal Best of all members of this Generation
  for mem_No in range(0,Swarm):
    if Personal_Best_fit[mem_No] < fitnesses[mem_No]:

      Personal_Best_fit[mem_No] = fitnesses[mem_No]
      Personal_Best_Particles[mem_No] = Particles_Coordinates[mem_No]
#------------------------------------------------------------------------------#

  #Step(4): Find Global Best:
  Global_Best_Fitness = max(fitnesses)
  Global_Best_index = fitnesses. index(Global_Best_Fitness)
  Global_Best = Particles_Coordinates[Global_Best_index]

  Visualize_Solution(Global_Best)
  print(Global_Best_Fitness)
  Obj_Fn_Records[Generation_Counter] = Global_Best_Fitness
#------------------------------------------------------------------------------#

  #Step(5): Update Velocities:
  for mem_No in range(0,Swarm):
    Global_Factor   = C1*random.random() * (Global_Best - Particles_Coordinates[mem_No])
    Personal_Factor = C2*random.random() * (Personal_Best_Particles[mem_No] - Particles_Coordinates[mem_No])

    Particles_Velocities[mem_No] = Particles_Velocities[mem_No]*IW + Global_Factor + Personal_Factor
#------------------------------------------------------------------------------#

  #Step(6): Update Positions:
    Particles_Coordinates = Particles_Coordinates + Particles_Velocities
#------------------------------------------------------------------------------#

  #Step(7): Validate Constraints:
    for mem_No in range(0,Swarm):
      for T in range(0,No_of_Towers):
        for C in range(0,2):
          if Particles_Coordinates[mem_No][T][C] <0+TowerR :   Particles_Coordinates[mem_No][T][C] = TowerR
          if Particles_Coordinates[mem_No][T][C] >100-TowerR : Particles_Coordinates[mem_No][T][C] = 100-TowerR
      Feasiblity_Correction(mem_No)
```

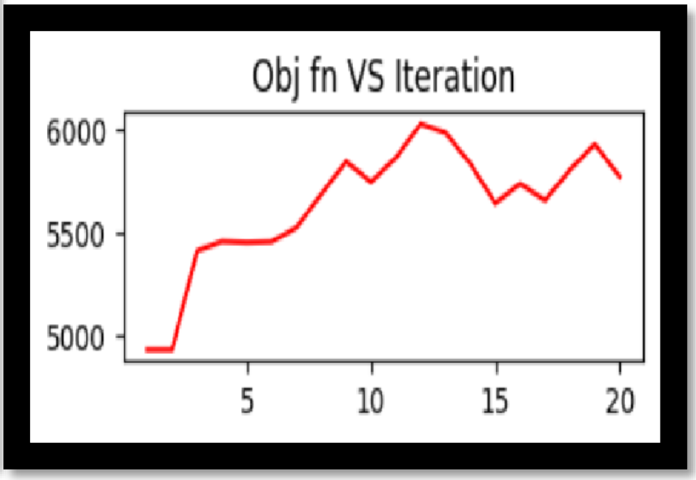
3.3.2 Results

@ no. of towers = 30 , tower radius = 8

No.	Pop Size	Iterations	Inertia	gBest Coeff	pBest Coeff	Cost
1	**20**	**20**	**0.005**	**0.2**	**0.03**	**5763**
2	20	20	0.005	0.2	0.05	**5879**
3	**20**	**20**	**0.005**	**0.2**	**0.07**	**6030**
4	20	30	0.005	0.2	0.1	**5656**
5	20	20	0.01	0.2	0.07	**5465**
6	20	20	0.003	0.2	0.07	**5881**
7	20	20	0.005	0.1	0.07	**5497**
8	20	20	0.005	0.3	0.07	**5512**
9	20	40	0.005	0.2	0.07	**5880**
10	40	20	0.005	0.2	0.07	**5586**
11	10	20	0.005	0.2	0.07	**5662**

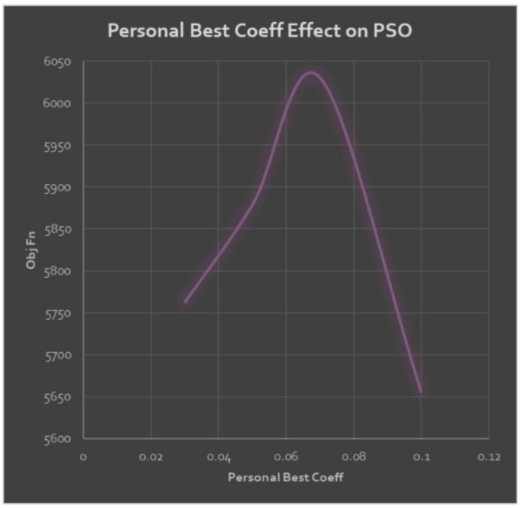
- **Best Solution:**



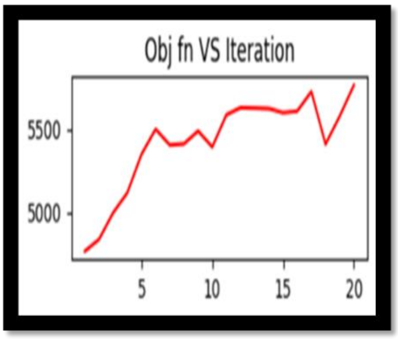


❖ Parameters Effect:

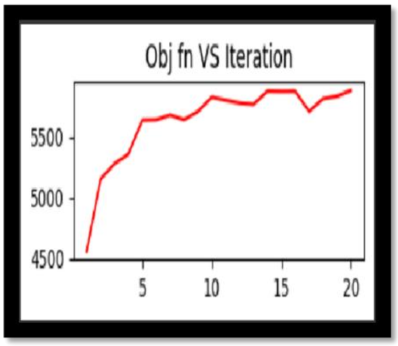
- Personal Best:



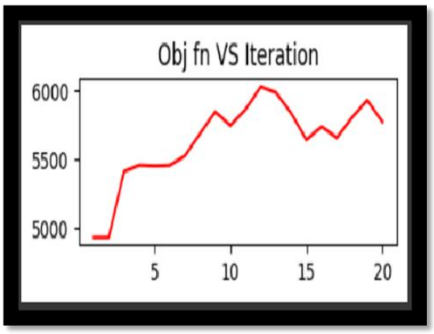
PB = 0.03



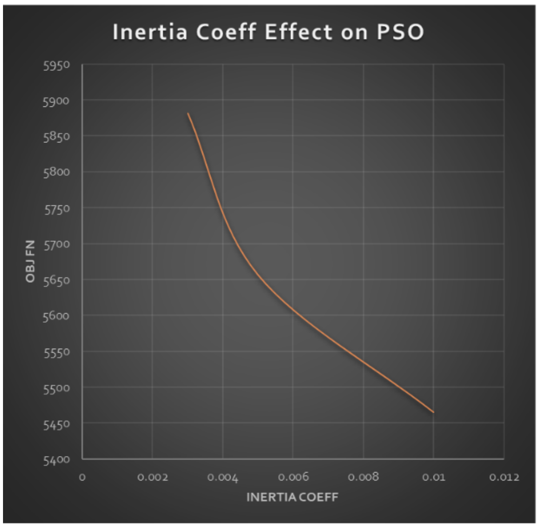
PB = 0.05



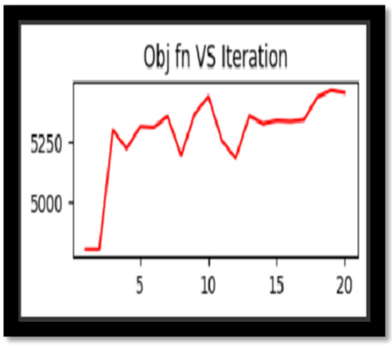
PB = 0.07



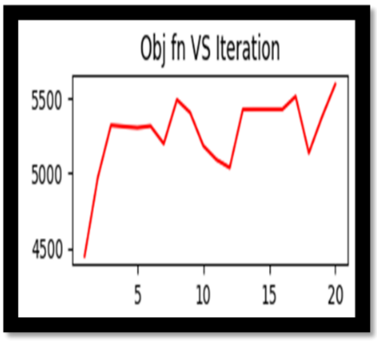
- Inertia Coeff.



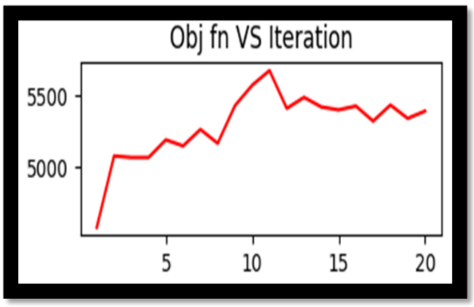
Inertia = 0.01



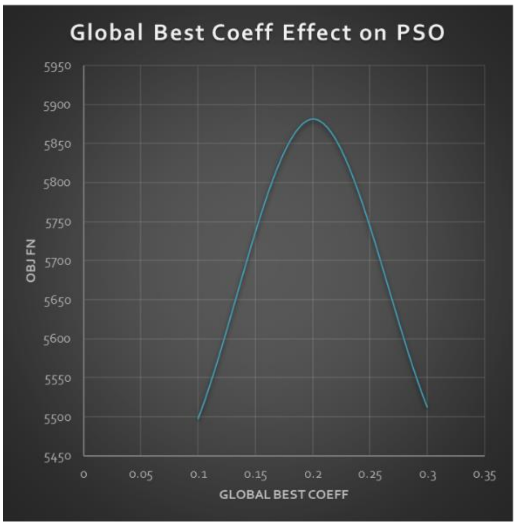
Inertia = 0.03



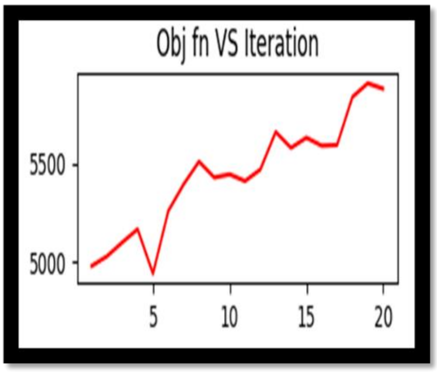
Inertia = 0.05



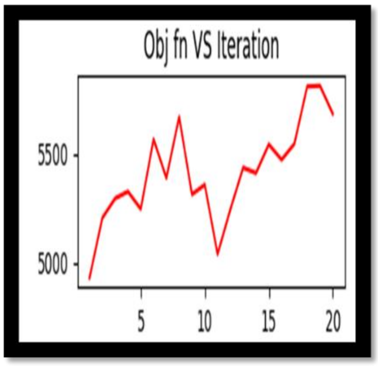
- Global Best



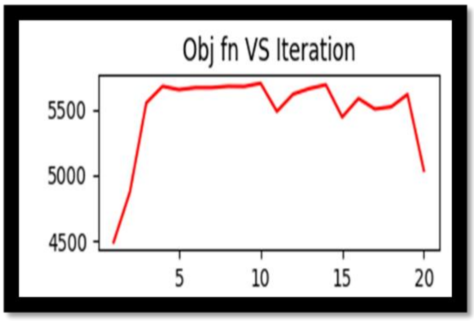
GB = 0.15



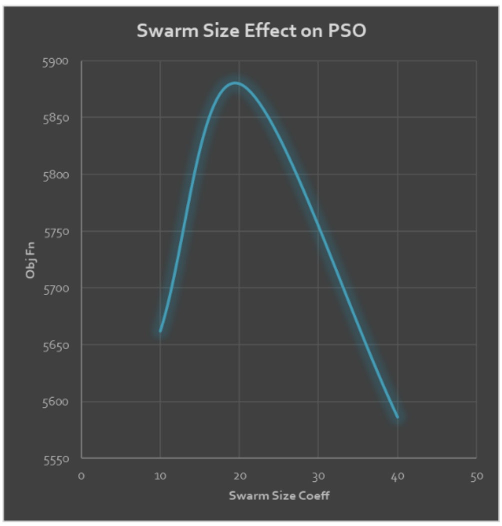
GB = 0.2



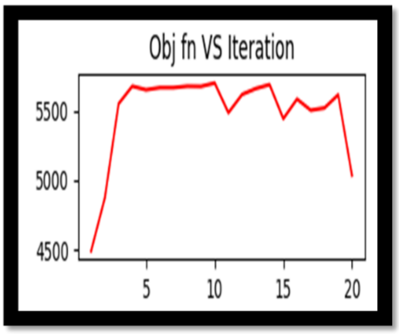
GB = 0.25



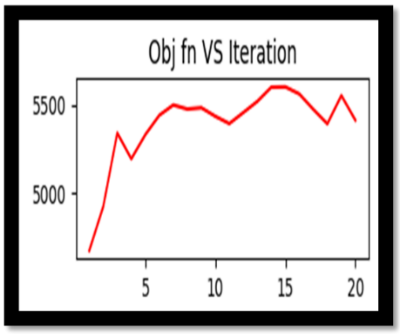
- Swarm Size



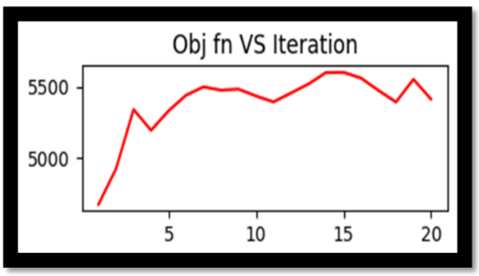
S.S = 10



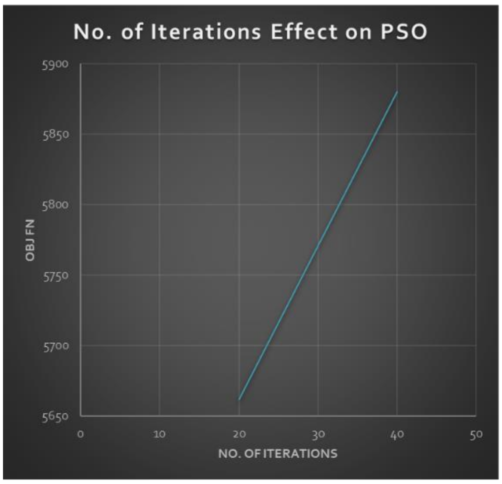
S.S = 20



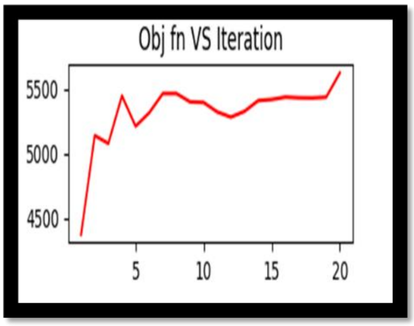
S.S = 40



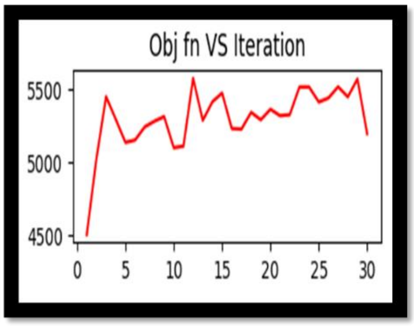
- No. of Iterations



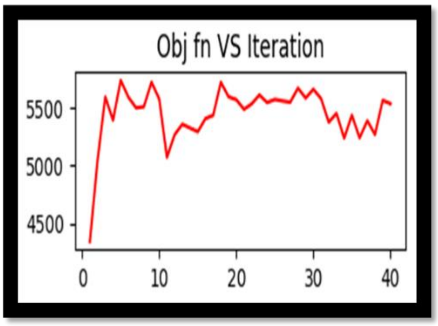
IN = 20



IN = 30



IN = 40



3.4 Grey Wolf Optimization (GWO)

is a new meta-heuristic optimization technology. Its principle is to imitate the behavior of grey wolves in nature to hunt in a cooperative way.

3.4.1 Codes

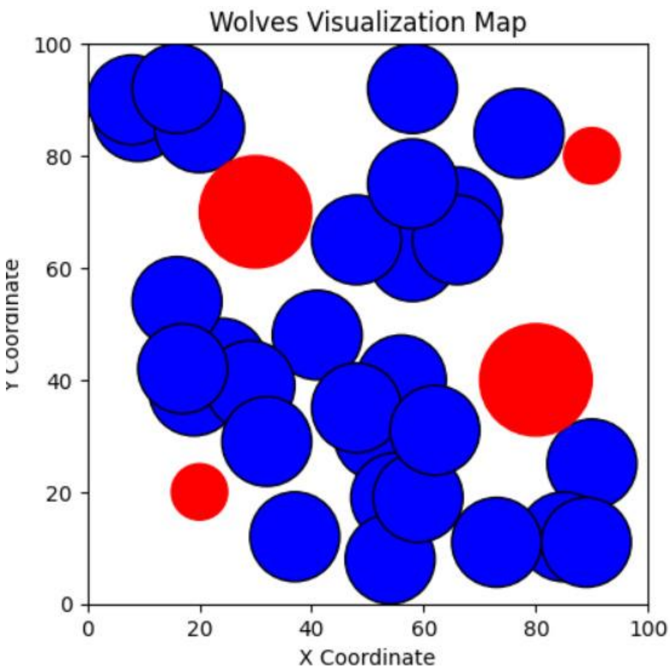
```
for i in range(i_max):
    a = 2*(1 - i/i_max)
    A = [Circle(random.uniform(-a, a), random.uniform(-a, a), tower_radius) for i in range(towers_number)]
    C = [Circle(random.uniform( 0, 2), random.uniform( 0, 2), tower_radius) for i in range(towers_number)]

    if magnitude(A) >= 1:   prey_sign = -1
    else:                   prey_sign =  1

    alpha_effect = alpha_wolf.copy()
    beta_effect  = beta_wolf.copy()
    delta_effect = delta_wolf.copy()

    # Update the omega wolves positions
    for ow in range(3, wolves_number):
        for t in range(towers_number):
            alpha_effect[t].x = alpha_wolf[t].x - prey_sign*abs(A[t].x) * abs(C[t].x*alpha_wolf[t].x - towers_group[io[ow]][t].x)
            alpha_effect[t].y = alpha_wolf[t].y - prey_sign*abs(A[t].y) * abs(C[t].y*alpha_wolf[t].y - towers_group[io[ow]][t].y)

            beta_effect[t].x  = beta_wolf[t].x  - prey_sign*abs(A[t].x) * abs(C[t].x*beta_wolf[t].x  - towers_group[io[ow]][t].x)
            beta_effect[t].y  = beta_wolf[t].y  - prey_sign*abs(A[t].y) * abs(C[t].y*beta_wolf[t].y  - towers_group[io[ow]][t].y)

            delta_effect[t].x = delta_wolf[t].x - prey_sign*abs(A[t].x) * abs(C[t].x*delta_wolf[t].x - towers_group[io[ow]][t].x)
            delta_effect[t].y = delta_wolf[t].y - prey_sign*abs(A[t].y) * abs(C[t].y*delta_wolf[t].y - towers_group[io[ow]][t].y)

            towers_group[io[ow]][t].x = (alpha_effect[t].x + beta_effect[t].x + delta_effect[t].x)/3.0
            towers_group[io[ow]][t].y = (alpha_effect[t].y + beta_effect[t].y + delta_effect[t].y)/3.0

            # Feasibility Check:
            while towers_group[io[ow]][t].feasibility_check() != True:
                towers_group[io[ow]][t].randomize()

    # Update the fitness values:
    fitness_values = [objective_function(towers_member) for towers_member in towers_group]

    io = list(np.argsort(fitness_values))   # Sort the members

    # Update the Wolves Group:
    alpha_wolf   = towers_group[io[0 ]].copy()
    beta_wolf    = towers_group[io[1 ]].copy()
    delta_wolf   = towers_group[io[2 ]].copy()

    # Store the Best Solution in scope of ALL Solutions:
    if objective_function(towers_group[0]) > objective_function(Best_towers):
        Best_towers = towers_group[0].copy()
```

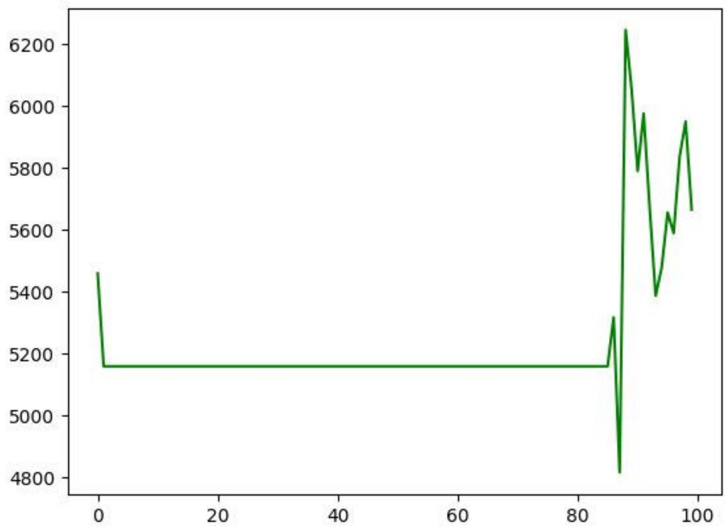
3.4.2 Results

@ no. of towers = 30 , tower radius = 8

No.	Pop Size	Iterations	Cost
1	5	100	6060.00
2	5	15	6050.00
3	10	100	6200.00

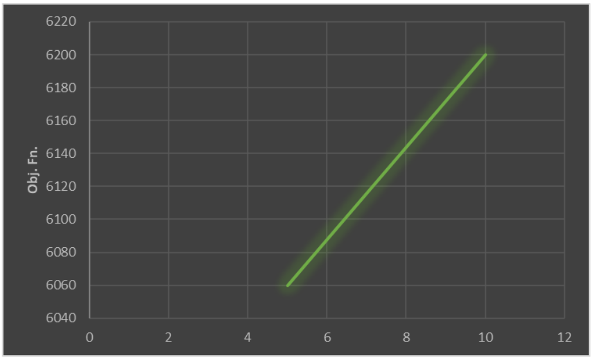
❖ **Best Result:**

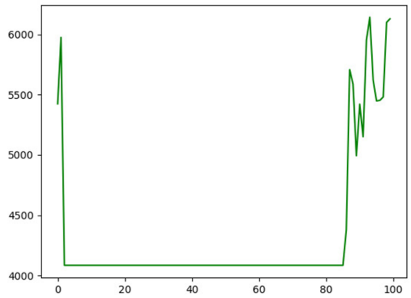




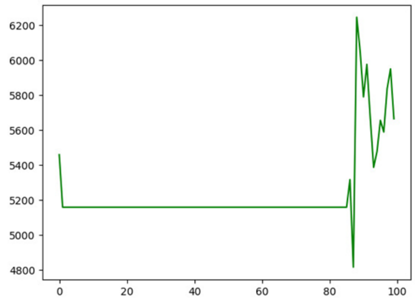
❖ Parameters Effect:

- **Population Size**



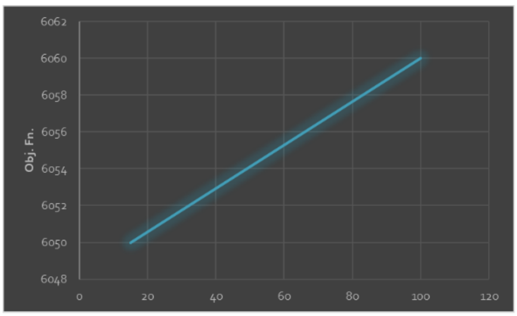


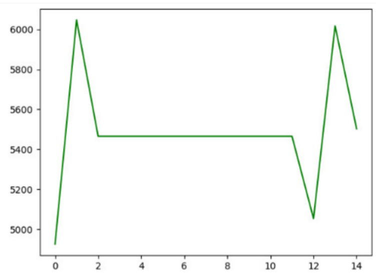
PS = 5



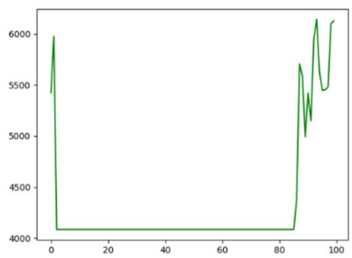
PS = 10

- **No. of iterations**





N = 15



N = 100

4 Conclusion

@ population size(SA not) = 200, no. of towers = 30 , tower radius = 8

	SA	GA	PSO	GWO
Min. Cost	6055.65	**7350.00**	**6030.00**	6200.00
Time(min)	**12**	03:20	**00:20**	01:00
Iterations	**500**	200	**20**	100

Note:

SA has the slowest time, and the GA has best Result, but PSO is the fastest and worst at the same time and the GWO is in average.