INDEX

Name Ishik Jain STD — SEC — ROLL NO: — SUB: —

IBM22CS356

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1	26/9	Genetic Algorithm	1	26/9/24
2	3/10	PSO Algorithm	2	3/10/24
3	24/10	genetic Algorithm	10	
4	7/11	PSO Algorithm	01	07-11-24
5	14/11	Ant colony optimization	10	
6	21/11	cuckoo search	10	21.11
7	28/11	grey wolf Algorithm	10	28.11
8	28/11	Parallel cellular Algo	—	
9	16/12/24	optimization via gene Exp Algo		

26/9/24

Lab 1

Genetic algorithm

adaptive heuristic search technique inspired of natural selection

1) Initialization:
   Define parameters such as pop size no of generation and mutation rate

2) Fitness evaluation eg $f(x) = x^2$.

3) Selection of Individuals on their fitness score. fitter individual have hieghet chance

4) crossover → to produce offspring

5) Mutation (eg 0.1) → Introduce random changes to offspring and mutation.

6) Iteration: repeat evaluation

7) Termination: end process

26/9/24

3/10/24

PSO particle swarm optimization

- Population Based stochastic Algo
  - → Random population
  - → fitness value
  - → update population
- * Application inspired by social behaviour of bird flocking or fish schooling
  - → robotics
  - → Finance
  - → Image processing
  - → Telecommunication

def PSO(np, nD, It)

Initialize particles with random position and velocity

set PB_pos = position

set PB_score = fitness of each particle

set GB_pos = best PB_pos

set GB_score = best PB_score

optimization

for each t from 1 to It

for each p from 1 to NP:

calculate fitness

if fitness PBscore[p] then:

update PB_score[p] and PB_pos[p]

update GB_pos and GB_score if nessicary

for each p from 1 to np

update

return GB_pos, GBscore.

Applications:

Implement A genetic algorithm to MAximize A for ($f(x) = x^2$)

```
import numpy as np
def objective_fn(x):
    return (x**2)
population_size = 100
mutation_size = 0.01
crossover_rate = 0.7
num_generation = 50
x_min = -10
x_max = 10

def initialize(size):
    return np.random.uniform(x_min, x_max, size)

def evaluate(population):
    return objective_fn(population)

def select(population, fitness):
    selected_indius = np.random.choice(len(population),
    size = 2, replace = False
    return population[selected_indus[
        np.argmax(fitness_indus)
```

```
def crossover (parent1, parent2)
    if np.random.rand() < crossover
                                rate
        return (parent1 + parent2)/2
    return parent1

def mutate (individual):
    if np.random.rand() < mutation
                                rate
        return Individual + np.random
                uniform (-1,1)
    return Individual

def genetic_algorithm():
    population = initialize (population_size)
    best_soln = none
    best_fitness = -np.inf
return best fitness . best_soln
```

Best x

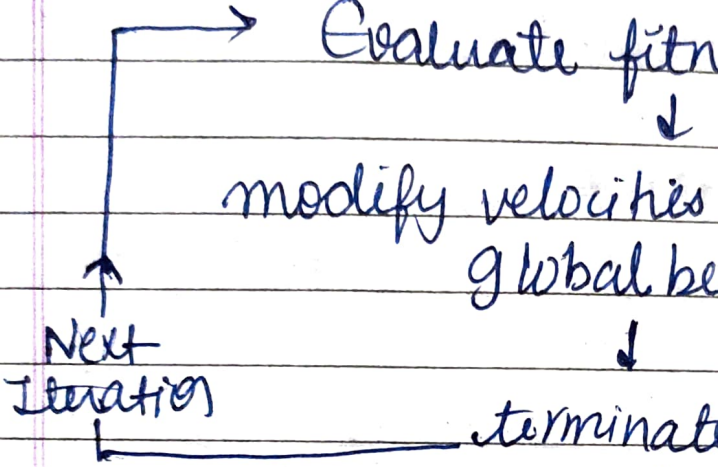
o/p = 9.96203495

Best value. 99.25334684

7/10/24

Lab-4

particle Swarm Algorithm



personal influence

$$V_{i+1} = w V_i + C_1 * rand() \cdot (PB_i - x_i) + C_2 \cdot rand() * (GB - x_i)$$

global Influ

$$x_{i+1} = x_i + V_{i+1}$$

```
import numpy as np
import matplotlib.pyplot as plt

def objective_fn(x):
    return np.sum(x**2 + x)

Initialize parameters
np, npar, mI, c1, c2 = 30, 2, 20, 0.5

Initialize global value
global_best_position = personal_best_position[np.argmin(personal_best_value)]
global_best_value = np.min(personal_best_value)

plt.figure(figsize=(10,8))
plt.xlim(-10,10)
plt.ylim(-10,10)
plt.xlabel('x axis')
plt.ylabel('y axis')
plt.title("PSO")
```

for i in range (num_particles)
r1 r2 = np.random.rand(2)
inertia = inertia_weight * particles_velocity[i]
cognitive = c1 * r1 * (personal_best_pos[i] - particle_pos[i])
social = c2 * r2 * (gbp - pp[i])
particles_velocity[i] = inertia + cognitive + social
pp[i] += particles_velocity[i]

Evaluate fitness

fitness = objective_fn (particles_position[i])
if fitness < personal_best_value[i]:
personal_best_value[i] = fitness
personal_best_position[i] = particles_position[i]

if fitness < global_best_value:
global_best_value = fitness
global_best_position = particles_pos[i]

plt.show()
print (global_best_position)
print (global_best_value)

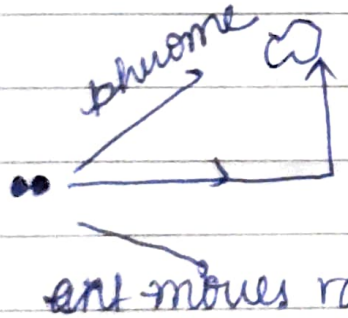
NP: hard.

The whole concept of ant colony optimization is to minimize the path and power consumption

⇓

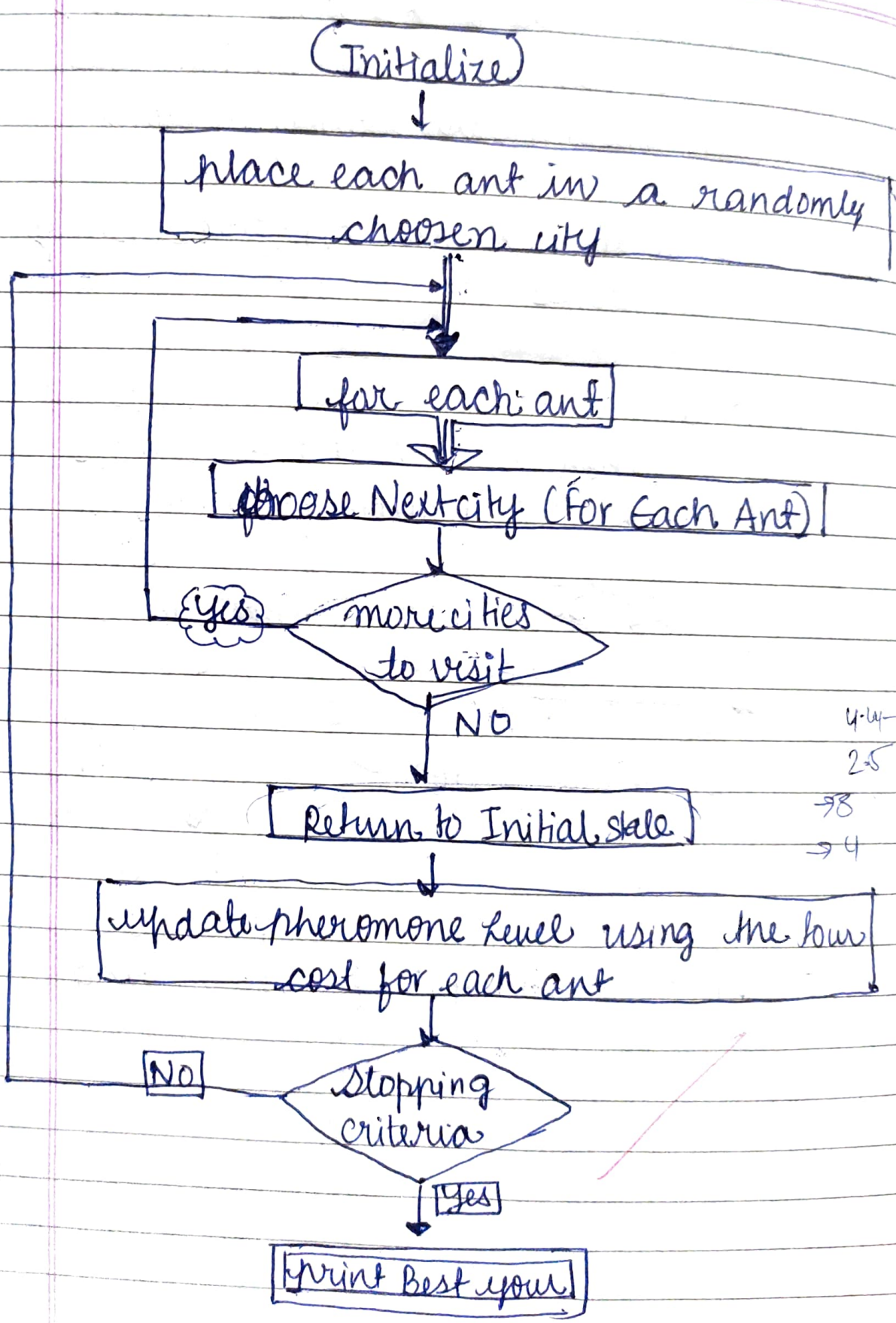
Less dependance on preexisting networks

ACO studies Artificial system and take Inspiration from real ant colonies
→ discrete optimization problems



pheromone trails
↓
shorter path
more pheromone
↓ trails
Ant follow Intense
pheromone trails

Autocatalytic +ve feedback algo


Initialize
place each ant in a randomly choosen city
for each ant
Choose Next city (For Each Ant)
more cities to visit
yes
NO
Return to Initial state
update pheromone level using the tour cost for each ant
Stopping criteria
NO
Yes
Print Best your

```
import numpy as np
import random
import matplotlib.pyplot as plt

cities = np.array([[0,0],[1,3],[4,3],[6,1],[3,0]])
num_ants, num_iteration = 5,100
alpha, beta, rho, initial_pheromone = 1.0, 2.0, 0.1, 0.0

def choose_next_city (current_city, visited):
    probs= [(pheromones [current_city][j]
                    ** alpha)
    * (1 / distance [current city][j] ** beta)
```

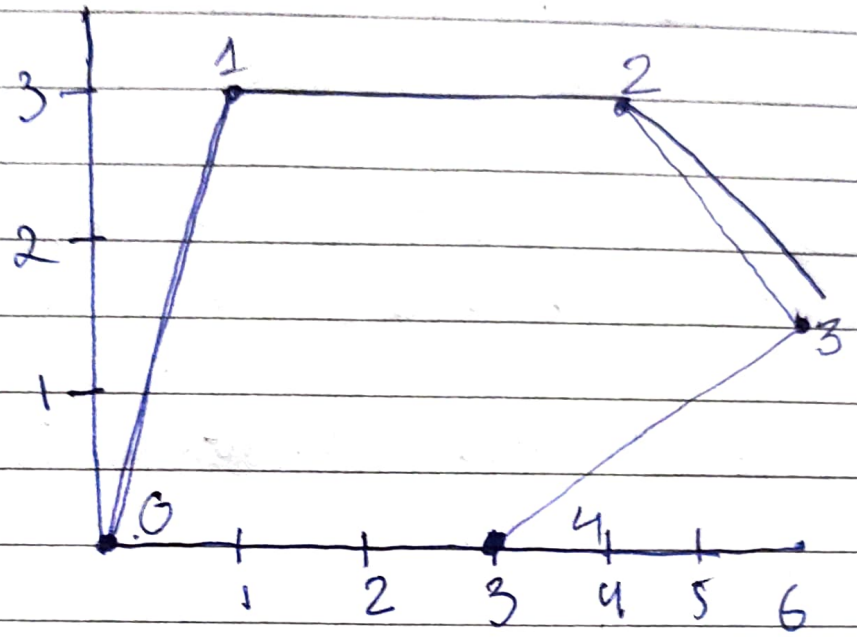
Algorithm:

```
best_route, best_distance = None, float ('inf')
    route, distance = [],[]
    for in range (num_iteration):
        routes, distance_list=[], []
    for _ in range (num_ants):
        route = [random.randint
while len(route) < num cities (0, num_cities-1)]

    d= sum[distance [route[i]]
                route [i+1]]
```

routes.append(route)
distance_list.append(distance)

update pheromes
plotting.



Best route ACO : 15:15 [illegible] units

Sen
14.11.22

```
# updation
    pheromones *= (1-rho)
    for route, dist in zip(routes, distance_list):
    for i in range(num cities):
        pheromones [route[i]][route[i+1]]
                = 1/dist
# ploting
    plt.scatter(cities[:,0], cities[:,1] color='red')
    for i, (x,y) in enumerate(cities):
        plt.text(x,y), f'{i}', fontsize=12,
                ha='right')
    for (i in range(num cities):
        plt.plot([cities[best route[i]][0]
                , cities[best route[i+1]][0]
    plt.show();
```

cuckoo search:

```
import numpy as np
from scipy.special import gamma
import random

//Rastrigin function
def rastrigin(x)
    A=10
    return (A*len(x) + sum(x_i**2 - A*np.cos(2*pi*x_i)
                for (x_i) in x)
```

// $f(x) = A \cdot n + \sum_{i=1}^{n} (x_i^2 - A \cdot \cos(2\pi x_i))$

```
//create random sol
def initialize_nest (num_nests, dim, lb, ub)
    nests = np.random.uniform (Lb, ub, (num_nests, dim))

    returns nest

def levy_flight (x, alpha=1.5, beta=1.0, delta=0.01)
    Sigma = (gamma(1+alpha) * np.sin(np.pi*alpha/2)
            gamma((1+alpha)/2) * alpha * 2**((alpha-1)/2)
                +2)))) **(1/alpha)
    u = np.random.normal (0, sigma, len(x))
    v = np.random.uniform (0, sigma, len(x))
```

step = u / np.abs(v)**(1/alpha)
newposition = x + delta * step
return newposition

def cuckoo_search(obfn, num_nests, num_itr,
dx=0.25 dim, lb, ub) alpha=0.25)
# nest = initialize_nest(num_nest, dim, lb, ub)
# fitness = np.array(objective_fn(nest) for nest in
nests)

best_nest_idx = np.argmin(fitness);
best_nest = nest[best_nest_idx];
best_fitness = fitness[best_nest_idx];

for iteration in range(num_itr):
for i in range(num_nest):
n_n = levy_flights(nest[i], alpha=alpha)
n_n = np.clip(n_n, lb, ub)
n_f = objective_fn(new_nest)

if new_fitness < fitness[i]:
nest[i] = n_n
fitness[i] = n_f

```
worst_nest_indices = np.argsort(fitness[-int(pa * num_nest):])
for idx in worst_nest_indices:
    nest[idx] = np.random.uniform(lb, ub, dim)
    fitness[idx] = objective_fn(nest[idx])

current_best_idx = np.argmin(fitness)
current_best_fitness = fitness[current_best_idx]
if current_best_fitness < best_fitness:
    best_fitness = current_best_fitness
    best_nest = nests[current_best_idx]

print("Iteration: {iteration+1}/{num_iterations}
      "best fitness: {best_fitness}")

return best_nest, best_fitness
```

num. nest = 25
num_iteration = 100
dim = 2
lb = -5.12
ub = +5.12
pa = 0.25
alpha: 1.5

best_sol, best_val = cuckoo_search (rastrigin,
num_nest, num_iteration,
dim, lower_bound
upper bond, pa, alpha)

✓ print(best solution)
✓ print (best value)

output
· [-0.00047078, 0.00177189]
[0.0006668312254056 73]

$$\Gamma(z) = \int_0^{\infty} t^{z-1} e^{-t} dt$$

$$\sigma = \left(\frac{\Gamma(1+\alpha) \cdot \sin(\pi\alpha/2)}{\Gamma\left(\frac{1+\alpha}{2}\right) \cdot \alpha \cdot 2^{\frac{\alpha-1}{2}}} \right)^{1/\alpha}$$

u → random variable from ND $\bar{x}$=0 to σ

V → 0 to 1

$$step = \frac{u}{|v|^{1/\alpha}}$$

new_pos = x + δ x step

21.11

28/11

Grey wolf Algorithm

Algorithm : Grey wolf optimization

1. Set the Initial value of population size n, parameter a, coefficient vector A, C and the max no of iteration MaxItn
2. set t=0 {counter Initialization}
3. for(i=1 ; i ≤ n) do
4. → Generate an initial population $x_i(t)$ randomly
5. → Evaluate the fitness fn of each search agent (sol) $f(x_i)$
6. end for
7. assing the value of 1st, 2nd 3rd best sol as x_α, x_β and x_δ respectively
8. repeat
   for (i=1 ; i ≤ n) do
   update each search agent in population as shown in eqn 12
   Decrease the parameter a from 2 to 0
   update the coefficient A and C shown on eqn
   Evalate fitness fn for search agent

15) update x_α x_β x_γ

16 set t = t+1 { Itr counter Increasing

17 until (t < MaxItr) { }

19 produce the best solution x_α

```
import numpy as np

def obj_fn(x)
    return np.sum(x**2)

def grey_w_o(obj_fn, dim, n_wolves=5
                max_itr=50, lb=-10, ub=+10):

    wolves = np.random.uniform(lb, ub, (n_wolves
                                        , dim))

    alpha = np.inf
    beta = np.inf
    delta = np.inf
    X_a = None
    X_b = None
    X_g = None

    for it in range(max_itr):
        fitness = np.array([objective_fn(wolf)
                            for wolf in values])

        sorted_index = np.argsort(fitness)
        X_a, X_b, X_b = wolves[sorted_index[:3]]
```

a = 2 - i * (2/max_itr)
new_wolves[]
for j in rang (n_wolves):
r1, r2 = np.random.rand(dim),
r2 = np.random.rand(dim),

A1 = 2*a*r1 - a
C1 = 2*r2
D_alpha = abs(c1 * x_alpha - wolves[j])
X1 = x_alpha - A1 * D_alpa

r1, r2 = np.random.rand(dim),
np.random.rand(dim)
A2, C2 = 2*a*r1 - a, 2*r2
D_beta = abs((2*x_b - wolves[j])
x2 = X_beta - A2 * D_beta

r1, r2 = np.random.rand(dim)
np.random.rand(dim)
A3, C3 = 2*a*r1 - a, 2*r2
D_Delta = abs(C3 * X_delta - wolves[j])
X3 = X_delta - A3 * D_Delta
new_wolf = (x1+x2+x3) / 3
new_wolves.append(np.clip(new_wolf, lb, ub))

print("best sol", best sol)
print ("best value :", best val)

{ hunting:
→ During hunting, the grey wolves encircle prey

$$D = |C - x_p(t) - A \cdot x(t)|$$

$$x(t+1) = x_p(t) - A \cdot D$$

coeff vector { $A = 2a \cdot r_1 \cdot a$, $C = 2 \cdot r_2$ } component linearly decreased

⇒
$$D_\alpha = |C_1 \cdot x_\alpha - X|$$
$$D_\beta = |C_2 \cdot X_\beta - X|,$$
$$D_\delta = |C_3 \cdot X_\delta - X|$$
$$x_1 = X_\alpha - A_1 \cdot (D_\alpha)$$
$$x_2 = X_\beta - A_2 \cdot (D_\beta)$$
$$X_3 = X_\delta - A_3 (D_\delta)$$

$$x(t+1) = \frac{x_1 + x_2 + x_3}{3} \}$$

20

15

Best sol : [0.20051618
-0.15836312
-0.23782372
-0.06212103
0.20312013

Best value : 0.16696254

1)
2) ✓

Lab 8

(parallel cellular automation-based algorithm for optimal robot route planning grid-based environment, ensuring collision free navigation while minimizing travel distance and computational time

```
import random
import numpy as np

def initialize_grid(N,M):
np.random.choice([0,1], size,([N,M])

def count_live_neighbours(grid,i,j,n,M)
    return sum(
        grid[ni,nj] for ni in range(i-1,i+2)
        for nj in range(ni != i or nj != j)

def update_cell(grid,new_grid,i,j,N,M):
    Live_neighbours = count_live_neighbours
                      (grid,i,j,N,M)
    if grid[i,j] == 1:
        new_grid[i,j] = 1 if live_neighbours
    else:
```

else:
new_grid[i,j] = 1 if live_neighbours (grid, i, j)

```
def printgrid(grid)
  for row in grid :
    print (''.join(map(str, row))
  print()

def parallel game (N,M,steps)
    grid = initialize_ grid(N,M)
      for _ in range(steps)
        printgrid ()
        for j in range (N)
      grid = new_grid
  print_grig(grid)

N,M = 5,5
steps = 5
final grid = 5
          = parallel of life(N,M, steps
```

Output

00000

01000

00011

01011

11101

00000

00000

00011

11000

11101

00000

00000

00000

10001

10100

00000

00000

00000

00000

Lab 9:- Bis Lab:-

```
import random
# Def knapsack problem (objective fn)
def knapsack problem (objective fn)
def knapsack_fitness (items, capacity, solution)
    total_wieght = sums [items[i][0] for i in
        range (len(solution) if solution[i]==1])
    total_value = sum[items[i][1] for i in range
        len(solution)) if solution[i]==1

# If total weight exceeds the capacity returns
if total_wieght > capacity
        return 0
return total value

# Gene expression algorithm (GEA)
    def-init-(self population size, num-items, mutation
        rate, crossover-rate, generation, capacity
        items)
    self.population_size = population size
    self.num-items = num-items
    self.mutation.rate = mutation-rate
    self.crossover_rate = crossover_rate
    self.capacity_rate = capacity
```

self.population = []

Initialize population with random sol (binary representation)

def initialize_population(self):

self.population = [[random.randint(0,1) for _ in range(self.population_size)]

# Evaluate fitness of population
def evaluate_fitness(self):
return [knapsack_fitness(self.items, self.capacity, individual) for individual in self.population]

Select individuals based on fitness (roulette wheel selection)

def selection(self):
fitness_values = self.evaluate_fitness()
total_fitness == 0: # Avoid division by zero
return random.choices(self.population, k=
selfpopulation_size)

```
# Crossover
def crossover(self, parent1, parent2):
    if random.random() < self.crossover_rate:
        crossover_point = random.randint(
            1, self.num_items - 1)
    return individual


# Evolve population over generation
def evolve(self):
    self.initialize_population()
    best_solution = None
    best_fitness = 0
    for gn in range(self.generation):
        # selection
        selected = self.selection()

# get user Input for gea parameter
def get_user_input
print("Enter population size)
best_solution, best_fitness = gea.evolve()
print("best solution"
```

Enter the no of Items 5
Enter weight and value of each item
Item 1: 10 20
Item 2: 30 40
Item 3: 50 60
Item 4 90 100
Item 5 70 80
Enter Knapsack capacity 90
population size 100
mutation rate 0.2
Crossover rate 0.9
Best solution (items selected): [1, 1, 1, 0, 0]
Best fitness (total value) = 120