

<p style="text-align: center;">Name Santosh H T</p> <p style="text-align: center;">Standard E Section Roll No. 244</p> <p style="text-align: center;">Subject B I S</p>				
SL No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	3/10/24	Exploration of all algorithms		
2.	24/10/24	Genetic Algorithm For Optimization problems.	g	88Q 16/11/24
3.	7/11/24	Particle Swarm Optimization for Function Optimization.		
4.	14/11/24	Ant Colony Optimization for the Travelling Salesman problem	10	88Q 16/11/24
5.	21/11/24	Cuckoo Search		88Q 21/11/24
6.	18/12/24	Grey Wolf Optimizer (GWO)		88Q 19/12/24
7.	18/12/24	Parallel cellular Algorithms and Programs.		19/12/24
8.	18/12/24	Optimization via Gene Expression Algorithm.		

## Algorithms:

- 1) Genetic Algorithms for Optimization Problems:
- GA are inspired by the natural process of selection and genetics.
  - The fittest individuals are selected for reproduction to produce next generation.
  - GA's implementation includes:
    - choosing a fitness function for optimization, population size, mutation rate, crossover rate, no of generations,
    - Evaluate the fitness function, choose the fittest individuals for reproduction, combine pairs of selected individuals to produce offspring..
    - Apply mutation to maintain genetic diversity.
    - Output the solutions found during iterations.

## Applications: Neural Networks.

- 2) Particle Swarm Optimization for Function Optimization:

The PSO is used for finding optimal solutions by simulating the social behaviour of birds or fish.

### Overview:

- PSO is inspired by social behaviour of birds flocking. Just as birds follow their neighbors and adjust their positions based on personal & collective experiences.
- The goal is to optimize a math fn, in this case Rastrigin function.

Particles: Each particle represents a potential solution, have a position (representing a candidate solution) and a velocity (indicating how they move).

### Best Positions:

The entire swarm tracks the global best position & value found by any particle in the swarm.

O/P: After completing the iterations it outputs the best position found & its corresponding value.

Applications: Engineering design, scheduling problems.

### 3) Ant Colony Optimization for the TSP:

- The foraging behaviour of ants has inspired the development of optimization algorithms.
- The ACO simulates the way ants find the shortest path between food sources and their nest.

#### Pheromone Trails:

- Ants communicate and find paths to food sources using pheromones, which are chemical substances they deposit on the ground.
- The intensity of pheromone trails influences the likelihood of other ants following the same path. The more ants that travel a path, the stronger the pheromone becomes.

- The ACO balances (searching new paths) & (refining known paths) through pheromone levels and heuristic info (such as distance or cost)
- When choosing the next city, ants use a probabilistic approach. The randomness allows for exploration of diverse paths.

#### Applications:

Routing problems : vehicle routing, network routing

#### 4) Cuckoo Search (CS):

Cuckoo Search is a nature-inspired optimization algorithm based brood parasitism. This behaviour involves laying eggs in the nests of other birds leading to optimization of survival strategies.

##### Levy Flights:

- CS uses Levy flights, a random walk pattern where the step lengths follow a heavy-tailed distribution. This allows the algorithm to explore the search space effectively and facilitates both local & global search.

→ Similar to natural selection, the algorithm selects the best solutions based on their fitness. The worst nests may be abandoned and replaced with new random solutions maintaining diversity in population.

Applications: Machine learning: Feature selection.

Complete Algorithms  
8/3/18

### 5) Grey Wolf Optimizer (GWO):

The GWO is a nature inspired optimization algorithm that stimulates the social hierarchy & hunting strategies of grey wolves, including alpha, beta, delta & omega roles. This algorithm is designed to solve continuous optimization problems by mimicking the collaborative hunting behaviour of grey wolves, where alpha wolves lead the search for prey & beta & delta wolves assist in refining the search. GWO iteratively updates the positions of wolves to approximate the best solution by encircling and attacking the target.

Its simplicity and effectiveness make it useful for optimizing complex functions in fields like machine learning, data analysis and engineering design.

### 6) Parallel Cellular Algorithms & Programs:

Parallel cellular algorithms are inspired by the functioning of geological cells that operate in a highly parallel and distributed manner, using principles from cellular automata and parallel computing to solve complex optimization problems. Each cell in a grid represents a potential solution, interacting with neighboring cells to update its state based on pre-defined rules, which models the diffusion of info across the cellular grid.

This enables effective exploration of search space & makes these algorithms particularly suitable for large scale optimization problems.

The fitness is evaluated using the optimized function.

24/10/24

## ▷ Genetic Algorithm For Optimization Problems:

Pseudocode:

function GeneticAlgorithm():

// Initialize parameters

Initialize parameters( population-size, mutation-rate,  
crossover-rate, num-generations, range-min, range-max)

// create initial population

population = InitializePopulation( population-size, range-min,  
range-max)

for generation in 1 to num-generations:

// Evaluate fitness

fitness = EvaluateFitness( population)

new-population = [ ]

// generate new population

for i from 1 to population-size/2 :

parent1, parent2 = Selection( population, fitness)

offspring1, offspring2 = Crossover( parent1, parent2)

new-population.append( Mutate( offspring1) )

new-population.append( Mutate( offspring2) )

population = new-population

best-fitness = Max( fitness )

best-solution = population [ Arg Max( fitness ) ]

return best solution. FitnessEvaluation( best-solution )

population\_size = 100

num\_generations = 50

mutation\_rate = 0.1

crossover\_rate = 0.7

x\_range = (-10, 20)

def initialize\_population(size, x\_range):

return np.random.uniform(x\_range[0], x\_range[1], size)

def objective\_function(x):

return -x\*\*2

def crossover(parent1, parent2):

return (parent1 + parent2) / 2 # simple averaging fn

def mutate(offspring, mutation\_rate, x\_range):

if np.random.rand() < mutation\_rate:

return np.random.uniform(x\_range[0], x\_range[1])

return offspring.

## 27 Particle Swarm Optimization for Function Optimization.

### 1. Objective Function

```
def objective_function(x):  
    return x**2
```

### 2. PSO Algorithm:

```
def pso(num_particles, max_iter, w, c1, c2, dim):  
    # initialize particles' positions & velocities  
    positions = np.random.uniform(-10, 10, (num_particles, dim))
```

# Random velocities

```
velocities = np.random.uniform(-1, 1, (num_particles, dim))
```

```
personal_best_positions = positions.copy()
```

```
personal_best_scores = np.array([objective_function(pos)  
                                for pos in positions])
```

```
global_best_position = personal_best_positions[np.argmin  
                                               (personal_best_scores)]
```

```
global_best_score = np.min(personal_best_scores)
```

### 3. Main PSO Loop:

```
for iter in range(max_iter):
```

~~for i in range(num\_particles):~~

# update velocity and position

$r1, r2 = np.random.rand(2)$  # random number for each particle

$velocities[i] = w * velocities[i] + c1 * r1 * (personal\_best\_position[i] - positions[i]) + c2 * r2 * (global\_best\_position - positions[i])$

$\text{positions}[i] = \text{positions}[i] + \text{velocities}[i]$

# evaluate fitness of the new position

$\text{fitness} = \text{objective\_function}(\text{position}[i])$

# Update personal best

if  $\text{fitness} < \text{personal\_best\_scores}[i]$ :

$\text{personal\_best\_scores}[i] = \text{fitness}$

$\text{personal\_best\_position}[i] = \text{positions}[i]$

#### 4. Update global best

$\text{min\_score\_index} = \text{np.argmin}(\text{personal\_best\_scores})$

if  $(\text{personal\_best\_scores}[\text{min\_score\_index}] < \text{global\_best\_score})$ :

$\text{global\_best\_score} = \text{personal\_best\_scores}[\text{min\_score\_index}]$

$\text{global\_best\_position} = \text{personal\_best\_positions}[\text{min\_score\_index}]$

return  $\text{global\_best\_position}, \text{global\_best\_score}$

#### 5. Initialize values

$\text{num\_particles} = 30$

$\text{max\_iter} = 100$

$\omega = 0.7$  # inertia weight

$c_1 = 1.5$  # cognitive coefficient

$c_2 = 1.5$  # social coefficient

$\text{dim} = 1$

8/18

7/11/24

### 3) Ant colony Optimization for the Travelling Salesman Problem:

1. Initialize necessary parameters

$N$  (Number of ants)  $\leftarrow 20$

$M$  (Number of cities)  $\leftarrow 10$

$\alpha$   $\leftarrow$  Influence of pheromone (blw 1 & 2)

$\beta$   $\leftarrow$  Influence of heuristic ( — " — )

$\rho$   $\leftarrow$  pheromone evaporation rate (blw 0 & 1)

$\eta$   $\leftarrow$  pheromone deposit factor

max-iterations  $\leftarrow$

2. Initialize pheromone trail ( $T$ ) on edges

$T(i, j) = T_0$  for all edges  $(i, j)$

Initialize cost or distance matrix  $D$

$D(i, j)$  is  $\Theta$  the distance between cities  $i$  and  $j$

3. for  $i$  in range(max-iterations) :

for  $k$  in range( $N$ ) :

path $[k] \leftarrow []$

visited $[k] \leftarrow []$

current-city $[k] \leftarrow$  random-start-city

visited $[k].add(\text{current-city})$

// construct solution by moving from city to city

while not all cities are visited by ant  $k$  do :

$$P(i, j) = \frac{[T(i, j)^\alpha * \eta(i, j)^\beta]}{\sum(T(i, j)^\alpha * \eta(i, j)^\beta)}$$

where  $\eta(i, j) = 1/D(i, j)$  (inverse distance)

// 6 select the next city  $i$  based on calculated prob

// 7 Add city  $i$  to the ant's path and mark it as visited

path $[k]$ .append( $i$ )

visited $[k]$ .add( $i$ )

current-city $[k] \leftarrow i$

// 8 calculate the length of the path for ant  $k$

Length $[k] \leftarrow$  total distance

// 9 Update the best solution if the current path is shorter

If Length $[k] < \text{best-length}$ :

best-solution  $\leftarrow$  path $[k]$

best-path  $\leftarrow$  Length $[k]$

// Update pheromone trail after all ants have completed their tours

for edge( $i, j$ ) do:

$T(i, j) \leftarrow (1 - \rho) * T(i, j)$

// Deposit pheromone trail after all ants have completed their tours

for each ant  $k$  do:

for each edge  $(i, j)$  in path $[k]$  do:

$T(i, j) \leftarrow T(i, j) + \Delta / \text{Length}[k]$

return best-solution, best-length.

#### 4) Cuckoo search (CS) :

```
import numpy as np
```

step 1: Initialise nests with random positions

```
def initialize_nests(N, dim, bounds):
```

```
    return np.random.uniform(bounds[0], bounds[1], (N, dim))
```

step 2: Fitness Evaluation function

```
def evaluate_fitness(nests, objective_function):
```

```
    return np.array([objective_function for nest in nests])
```

step 3: Levy Flight to generate new solution

```
def levy_flight(nests, alpha=1.0):
```

```
N,
```

```
for i in range(N):
```

```
    u = np.random.normal(0, 1, dim)
```

```
    v = np.random.normal(0, 1, dim)
```

```
    step = u / (np.abs(v) ** (1.0 / alpha))
```

```
    new_nests[i] = nests[i] + step * 0.01
```

```
return new_nests.
```

```
def get_best_nest(nests, fitness):
```

```
    best_idx = np.argmin(fitness)
```

```
    return nests[best_idx], fitness[best_idx]
```

```
def cuckoo_search(N, dim, bounds, max_iter=100, pa=0.25, alpha=1.0):
```

```
# Initialize nests
```

```
nests = initialize_nests(N, dim, bounds)
```

```
# Evaluate fitness of initial nests
```

```
fitness = evaluate_fitness(nests, objective_function)
```

```
# Track the best nest
```

```
best_nest, best_fitness = get_best_nest(nests, fitness)
```

```
# Iterate over max_iter
```

```
for i in range(max_iter):
```

```
# Generate new solutions via levy flights
```

```
new_nests = levy_flight(nests, alpha)
```

```
new_fitness = evaluate_fitness(new_nests, objective_fn)
```

```
nests, fitness = abandon_worst_nests(nests, fitness,
```

```
new_nests, new_fitness, pa)
```

```
# Track the best solution found
```

```
current_best_nest, current_best_fitness =
```

```
get_best_nest(nests, fitness)
```

```
# If the current soln is better, update the best solution
```

```
if current_best_fitness < best_fitness:
```

```
best_nest = current_best_nest
```

```
best_fitness = current_best_fitness
```

```
return best_nest, best_fitness
```

N = 20 # Number of nests

dim = 2 # problem dimension

bounds = (-5, 5) # search space bounds

28/11/24

## 5) Grey Wolf Optimizer (GWO):

Pseudocode:

Step 1: Randomly initialize Grey Wolf optimization population of N particles  $x_i$  ( $i=1, 2, \dots, n$ )

Step 2: Calculate the fitness of each individuals  
sort grey wolf population based on fitness value  
alpha-wolf = wolf with least fitness value  
beta-wolf = wolf with second least fitness value  
gamma-wolf = wolf with third least fitness value

Step 3: For Iter in range(max\_iter):

calculate the value of  $a$

$$a = 2 * (1 - \text{Iter}/\text{max\_iter})$$

For i in range(N):

a. compute the value of  $A_1, A_2, A_3$  and  $C_1, C_2, C_3$

$$A_1 = a * (2 * r_1 - 1), A_2 = a * (2 * r_2 - 1), A_3 = a * (2 * r_3 - 1)$$

$$C_1 = 2 * r_1, C_2 = 2 * r_2, C_3 = 2 * r_3$$

b. compute  $x_1, x_2, x_3$

$$x_1 = \text{alpha-wolf.position} -$$

~~$$A_1 * \text{abs}(C_1 * \text{alpha-wolf-position} - \text{i-th-wolf-pos})$$~~

~~$$x_2 = \text{beta-wolf-position} - A_2 * \text{abs}$$~~

~~$$(C_2 * \text{beta-wolf-position} - \text{i-th-wolf-position})$$~~

~~$$x_3 = \text{gamma-wolf-position} - A_3 * \text{abs}$$~~

~~$$(C_3 * \text{gamma-wolf-position} - \text{i-th-wolf-position})$$~~

c. compute new solution & its fitness

$$x_{\text{new}} = (x_1 + x_2 + x_3) / 3$$

$$f_{\text{new}} = \text{fitness}(x_{\text{new}})$$

2. Update the  $i$ th\_wolf greedily

if ( $f_{new} < i$ th\_wolf.fitness)

$i$ th\_wolf.position =  $x_{new}$

$i$ th\_wolf.fitness =  $f_{new}$

End for

# compute new alpha,beta and gamma

alpha\_wolf = wolf with least fitness values

beta\_wolf = wolf with second least fitness values

gamma\_wolf = wolf with third least fitness value

End for

Step 4: Return best wolf in the population.

## 6) Parallel cellular Algorithms and programs

Function Parallel Cellular Algorithm:

Define objective Function()

InitializeGrid(GridSize)

For each cell in the grid:

Randomly initialize cell's position in the solution space

# Evaluate Fitness

For each cell in the grid:

EvaluateFitness(cell)

# Update state

For iteration=1 to MaxIterations:

For each cell in the grid:

neighbours = GetNeighbours(cell, neighbourhood)

newstate = calculateNewState(cell, neighbours)

cell.state = newState

For each in the grid:

EvaluateFitness(cell)

TrackBestSolution()

Output BestSolution()

Initialize parameters

Set GridSize = (rows, cols)

Set NumCells = rows \* cols

Set MaxIterations = 1000

Set convergenceThreshold = 0.001

unction InitializeGrid(Gridsize):

Initialize cells with random positions

Function EvaluateFitness(cell):

cell.fitness = ObjectiveFunction(cell.state)

Function GetNeighbours(cell, neighbourhood):

return the list of neighbouring cells.

Function calculateNewState(cell, neighbours):

Return new state for the cell based on predefined rules.

Function TrackBestSolution():

bestsolution = min(bestSolution, cell)

## 7 Optimization via Gene Expression Algorithms:

Function GeneExpressionAlgorithm:

# step 1

Define objective function()

# step 2: Initialize parameters

Set population = 100, NumGenes = 10, MutationRate = 0.05,  
crossoverRate = 0.7, NumGenerations = 1000

# step 3: Initialize population

population = InitializePopulation(populationSize, NumGenes)

# step 4: Evaluate Fitness

For each individual in population:

EvaluateFitness(individual)

For generation = 1 to NumGenerations:

selected = SelectIndividuals(population)

offspring = crossover(selected)

MutateOffspring(offspring, MutationRate)

for each individual in offspring:

individual.solution = DecodeGenes(individual.genes)

EvaluateFitness(individual)

TrackBestSolution()

o output BestSolution.

Function InitializePopulation (populationSize , NumGenes):

Return [ GenerateRandomGenes(NumGenes) ]

Function EvaluateFitness(individual):

individual.fitness = Objective Function (individual.genes)

Function crossover(selected):

Return [ PerformCrossover(selected[i] , selected[i+1]) ]

Function MutateOffspring (offspring, MutationRate):

For each individual in offspring:

If Random () < MutationRate:

Mutate(individual)

Function TrackBestSolution():

bestsolution = min (bestSolution, individual)

BB  
1234  
1234