INDEX

Name Santosh H J

Standard Section E Roll No. 244

Subject BIS

SL No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	3/10/24	Exploration of all algorithms	9	SSJ 14/11/24
2.	24/10/24	Genetic Algorithm For Optimization problems.		
3.	7/11/24	Particle Swarm Optimization for Function Optimization.		
4.	14/11/24	Ant Colony Optimization for the Travelling Salesman problem	10	SSJ 14/11/24
5.	21/11/24	Cuckoo Search		SSJ 21/11/24
6.	18/12/24	Grey Wolf Optimizer (GWO)		
7.	18/12/24	Parallel cellular Algorithms and Programs.		
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 likehood 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: vehical routing, network routing etc.

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.

Lévy Flights:
- CS uses Lévy 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 Algorithm
3/10

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 operates 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

1> 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, FitnessFunction(best_solution)
```

```
population_size = 100
num_generations = 50
mutation_rate = 0.1
crossover_rate = 0.7
x_range = (-10, 10)

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.
```

2) 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_positions[i]
                    - positions[i]) + c2*r2*(global_best_position
                                     - position[i])
```

```
positions[i] = positions[i] + velocities[i]

# evaluate fitness of the new position
fitness = objective_function(position[i])

# Update personal best
if fitness < personal_best_scores[i]:
    personal_best_scores[i] = fitness
    personal_best_positions[i] = positions[i]
```

4. Update Global best

```
min_score_index = np.argmin(personal_best_scores)
if (personal_best_scores[min_score_index] < global_best_score:
    global_best_score = personal_best_scores[min_score_index]
    global_best_position = personal_best_positions[min_scores_index]

return global_best_position, global_best_score
```

5. Initialize values

```
num_particles = 30
max_iter      = 100
w = 0.7          # inertia weight
c1 = 1.5         # cognitive coefficient
c2 = 1.5         # social coefficient
dim = 1
```

8/8

7/11/24

3) Ant colony Optimization for the Travelling Salesman Problem:

1. Initialize necessary parameters

N (Number of ants) ← 20

M (Number of cities) ← 10

alpha ← Influence of pheromone (b/w 1 & 2)

beta ← Influence of heuristic (—— " ——)

rho ← pheromone evaporation rate (b/w 0 & 1)

Q ← pheromone deposit factor

max_iterations ←

2. Initialize pheromone trail (T) on edges

T(i,j) = T0 for all edges (i,j)

Initialize cost or distance matrix D

D(i,j) ← the distance between cities i and j

3. for i in range(max_iterations):

    for k in range(N):

        path[k] ← [ ]

        visited[k] ← [ ]

        current_city[k] ← random_start_city

        visited[k].add(current_city)

    // construct solution by moving from city to city

    while not all cities are visited by ant k do:

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

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

// 6 select the next city j based on calculated prob

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

```
path[k].append(j)
visited[k].add(j)
current_city[k] ← j
```

// 8 Calculate the length of the path for ant k

```
Length[k] ← total distance
```

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

```
If Length[k] < best_length:
    best_solution ← path[k]
    best_path ← Length[k]
```

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

```
for edge(i,j) do:
    T(i,j) ← (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) ← T(i,j) + Q/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[:] + 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

8/9/25

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 $(i = 1, 2, \ldots, n)$

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

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

For i in range (N):

a. compute the value of A1, A2, A3 and C1, C2, C3

$A1 = a * (2 * r1 - 1)$, $A2 = a * (2 * r2 - 1)$, $A3 = a * (2 * r3 - 1)$

$C1 = 2 * r1$, $C2 = 2 * r2$, $C3 = 2 * r3$

b. compute X1, X2, X3

$X1 = alpha\_wolf.position - A1 * abs(C1 * alpha\_wolf\_position - ith\_wolf\_pos)$

$X2 = beta\_wolf\_position - A2 * abs(C2 * beta\_wolf\_position - ith\_wolf.position)$

$X3 = gamma\_wolf\_position - A3 * abs(C3 * gamma\_wolf\_position - ith\_wolf\_position)$

c. compute new solution & it's fitness

$Xnew = (X1 + X2 + X3) / 3$

$fnew = fitness(Xnew)$

d. Update the ith_wolf greedily

if (fnew < ith_wolf.fitness)

ith_wolf.position = xnew

ith_wolf.fitness = fnew

End for

#compute new alpha, beta and gamma

alpha_wolf = wolf with least fitness value

beta_wolf = wolf with second least fitness value

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 ParallelCellularAlgorithm:

```
Define objectiveFunction()
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 states
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
```

Function 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 objectiveFunction()

    #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(individuals.genes)
        EvaluateFitness(individual)

    TrackBestSolution()

o   output BestSolution..
```

Date: Page:

```
Function Initialize population(populationsize, NumGenes):
    Return [GenerateRandom Genes(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(best Solution, individual)
```