INDEX

Name: Sanjeet P. Pandit (1BM22CS241)

Subject: BIS Class: — Div: — Roll No.: —

School: —

Sl. No.	Date	Title	Page No.
1	3/10/24	Explaining all algorithms	10 SSP 14/11/24
2	24/10/24	Genetic Algorithm	
3	7/11/24	Particle Swarm Optimization	
4	14/11/24	Ant Colony Optimization for TSP	10 SSP 21/11/24
5	21/11/24	Cuckoo Search (CS)	
6	28/11/24	Grey Wolf optimizer (GWO)	
7	18/12/24	Parallel Cellular Algorithm	
8	18/12/24	Optimization via gene expression	

Page No.
Date. / /

(1) Genetic Algorithm for optimization problems:

```
function GeneticAlgorithm():
    Initialize parameters (population_size (100), mutation_rate, crossover_rate,
    num_generations, range_min, range_max) (10%) (70%)
    population = InitializePopulation(population_size (50), range_min (-10), range_max (10))

    for generation in 1 to num_generations:
        fitness = EvaluateFitness(population)
        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[ArgMax(fitness)]
        print("Generation", generation, "Best solution = ", best_solution,
              "Fitness = ", best_fitness)

    return best_solution, FitnessFunction(best_solution)
```

Page No.
Date. / /

→ (fitness) evaluation:
function takes population as input and computes fitness for each individual by applying fitness-function. It returns an array of fitness values (return np.array([fitness-function(x) for x in population])

→ total-fitness = np.sum(fitness)
selection-probs = fitness / total-fitness
return population [np.random.choice(len(population), size=2, p=selection-probs)]

(Selection) using roulette wheel
→ It calculates total fitness
→ computes probability of selection for each individual
→ uses these probabilities to select two parents for reproduction

→ (Crossover)
→ this function combines two parents to create offspring
→ If random number less than crossover rate, it returns weighted combination of two parents
→ If not, it returns parent unchanged (no crossover)

→ (Mutation)
→ If random number less than mutation rate, it generates new random individual within specified range
→ If not, it returns individual unchanged

(2) Particle Swarm Optimization (PSO)

(1) Define problem

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

where $A = 10$, n = dimensionality of input vc

x_i is value of ith variable in input vector

Goal: to minimize the function

(2) Initialize parameters

$n \rightarrow$ no. of vehicles

$w \rightarrow$ Inertia weight (impact of prev velocity on curr velocity b/w 0.4 and 0.9)

$c1 \rightarrow$ cognitive coeff (controls particles own best position)

$c2 \rightarrow$ social coeff (best known position by swarm)

(3) Initialize particles

$x, \rightarrow$ position

$v \rightarrow$ velocity

personal best position (p_i)

personal best fitness ($f(p_i)$)

(4) Evaluate fitness

Rastrigin function:

fitness for curr position: score = f(x_i);

if score < personal_best_score[i]:

update personal_best_position[i] to x_i

update personal_best_score[i] to score

(5) Update velocities and positions

generate random values, r1, r2, between 0 and 1

update velocity (w*v_i(t) + c1*r1*(pers_best_pos[i] - pos[i]) + c2*r2*(global_best_pos - pos[i])

update position

Page No.
Date. / /

(6) Iterate
for each iteration t (t = 1 to num_iterations):
for each particle i (i = 1 to num_particles):
calculate fitness score at new position
if score is better than personal best, update
global best solution

(7) Output best solution
After, return global best position & score
global minimum occurs at $x_i = 0$ (all $x \Rightarrow 0$)

i.e $An + (0 - A(n)) = An - An = 0$

SPD
7/11/24.

Page No.
Date. / /

(3) Ant Colony optimization for Travelling Salesman problem

(i) Initialize TSP with cities coordinates
A list of cities with their (x,y) coordinates
eg city 0 : (0,0) → starting point
city 1 : (1,5)
city 2 : (3,1)
city 3 : (5,3)
city 4 : (6,6)
Distance matrix : between each pair of cities
$d[i,j] = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

(ii) Calculate distance matrix for all pairs of cities

$$dist = \begin{pmatrix} 0 & d_{01} & d_{02} & d_{03} & d_{04} \\ d_{10} & 0 & d_{12} & d_{13} & d_{14} \\ d_{20} & d_{21} & 0 & d_{23} & d_{24} \\ d_{30} & d_{31} & d_{32} & 0 & d_{34} \\ d_{40} & d_{41} & d_{42} & d_{43} & 0 \end{pmatrix}$$

(dist ij represents euclidean distance b/w i & j)

(iii) Initialize Ant Colony Parameters
→ No. of ants - how many ants explore solution space in each iteration, each ant construct a tour through cities
→ Pheromone Importance (alpha) - higher val means ants will prefer those paths with stronger pheromone trails
→ Distance Importance (beta) - dist b/w cities, higher val encourages ants to choose shorter paths
→ Pheromone Evaporation Rate (rho) - pheromones naturally evaporate over time. A higher evaporation rate leads to faster decay of pheromones, preventing alg. from getting stuck in local optima
→ Iterations - no. of iterations alg will run, each iteration involves multiple ants and exploring cities and updating pheromone trails (100 iterations, 10 ants)

(iv) Iterate over solution construction process (Main ACO Loop)
for each iteration (1 to max-iterations), alg performs following steps:
(a) Initialize ants for ants solution and costs (Here costs update pheromone trails)
(b) construct soln for each ant
(choose-next-city())

$$P(i) = \frac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum_{k \in \text{unvisited cities}} (\tau_{ik})^{\alpha} \cdot (\eta_{jk})^{\beta}}$$

τ_{ij} → pheromone level from city i to j
η_{ij} → heuristic val ($\eta_{ij} = \frac{1}{d_{ij}}$) (inverse of dist b/w cities i & j)
α, β → parameters (Pheromone & dist)
visits cities until all are visited & calculates tour's cost
(c) update best soln (if shorter, becomes best soln)
(d) update pheromones (based on 5) also

$$\tau_{ij} \leftarrow \tau_{ij} + \frac{1}{\text{cost from } i \text{ to } j}$$

(e) Print iteration process

(v) output best solution and cost
best soln → sequence of cities that constitutes shortest tour
best cost → total dist of tour

best route → [0, 1, 4, 3, 2] (order in which cities are visited)
best cost → 19.35 (0 → 1 → 4 → 3 → 2 → 0)

($d_{01} = 5.1$, $d_{14} = 5.1$, $d_{43} = 3.16$, $d_{32} = 2.83$, $d_{20} = 3.16$)
($\rightarrow 5.1 + 5.1 + 3.16 + 2.83 + 3.16 = 19.35$)

Sol
14/11/24

(4) Cuckoo Search (CS):

```
Function CuckooSearch(Func, D, N, MaxIter, pa, alpha):
  // Func: objective fxn (Σx²) f(x) = x1² + x2² + ... + xD² (Sphere fn)
     D: dimensionality of problem (no. of decision variables)
     N: no. of nests (population size)
     MaxIter: max no. of iterations
     pa: prob of discovery (fraction of worst nests to replace)
     alpha: step size for levy flight (scaling factor) //

     nests = RandomlyInitialize(N, D)  // N nests, each with
                                          D random values in [0,1]
     fitness = EvaluateFitness(nests, Func)

     best_nest = nests[IndexOfMinFitness(fitness)]
     best_fitness = Min(fitness)

     for iteration = 1 to MaxIter:
        // Generate new solutions (nests) using levy flights
        for each nest i in nests:
           stepsize = alpha * LevyFlight(D)
           new_nest = nests[i] + step_size
           new_fitness = Func(new_nest)

           if new_fitness < fitness[i]:
              nests[i] = new_nest      // replace old nest with new one
              fitness[i] = new_fitness // update fitness of nest

        // Abandon worst nests & replace them with new
        random nests
        worst_nests = SelectWorstNests(fitness, pa)
        ReplaceWithRandomNests(worst_nests, D)
        EvaluateFitness(worst_nests, Func)
```

Page No.
Date. / /

//update best soln found so far
current_best_index = IndexofMinFitness(fitness) //index of best nest
if fitness[current_best_index] < best_fitness:
    best_nest = nests[current_best_index] //update best nest
    best_fitness = fitness[current_best_index]

Return best_nest, best_fitness

function LevyFlight(D):
    return RandomNormal(0, 1, D) //D-dim step size

function RandomlyInitialize(N, D):
    return Random(N, D)

function EvaluateFitness(nests, Func):
    Return Apply(Func, nests)

Function SelectWorstNests(fitness, pa):
    Return SortByFitness(fitness)[-int(pa*N):] //select worst pa*N nests
    // identify worst nests (highest fitness values)

Function ReplaceWithRandomNests(worst_nests, D):
    For each worst_best in worst_nests
        worst_nest = Random(D) //Replace worst nest with new random solution

Output:
Best solution: [0.0751 0.0287 0.1106 0.0438 0.1728]
Best Fitness (Objective value): 0.0505

8/8
21/11/24

Page No.
Date. / /

(5) Grey Wolf Optimizer (GWO)

Inputs:

objective fun : f(x) to minimize/maximize

n : no. of wolves (population size)

max_iter : no of iterations

Bounds : lower & upper limits of search space

outputs:

Best soln (Alpha wolf's position)

Fitness value (quality of solution)

(1) Initialize Alpha, Beta, Delta

alpha_posn = [0, 0, ... 0] (dimensional vector)

beta_posn = [0, 0, ... 0]

delta_posn = [0, 0, ... 0]

alpha_score = infinity

beta_score = infinity

delta_score = infinity

(2) Generate Initial population

→ Randomly initialize positions of n wolves within bounds

for each wolf in population:

generate random posn within bounds

(3) Evaluate fitness

→ calculate fitness of each wolf using obj fun f(x)

for each wolf in population:

compute fitness = f(posn)

(Random coeffs (A, C) ensure balance between exploration & exploitation)

(4) Assign Alpha, Beta, Delta (compare fitness values & update alpha, beta, delta pos)

for each wolf:

if fitness < alpha_score:

update delta → beta, beta → alpha

else if fitness < beta_score:

update delta → beta

else if fitness < delta_score:
update delta

(5) Iterative optimization
for t=1 to Max_iter:
(1) update posns:
each wolf posn is updated using influence of alpha, beta, and delta wolves
(2) Boundary check:
ensure updated posn remains within bounds
(3) Re-evaluate fitness
update alpha, beta, delta posns & scores based on new fitness values

(6) End iterations
→ After completing max_iter, return alpha wolf's position and fitness as best solution
return alpha_pos (best soln) & alpha_score (fitness)

eg $f(x) = x^2$ in $[-10, 10]$
random posns (-7, 5, 2)
⇒ $f(x) = 5^2 = 25$ (α)
$f(x) = (-7)^2 = 49$ (β) & so on
After many iterations, α converges to min soln
(eg $x = 0 \rightarrow f(x) = 0$)

Applications -
Engineering (eg design optimization)
Data Analysis (feature selection)
ML (hyperparameter tuning)

Page No.
Date. / /

Output:

eg Alpha-posn (Best posn): [0.2, -0.1, 0.0, 0.3, -0.05]
Alpha-score (Best score): 0.155
$\left(\sum_{i=1}^{5} x_i^2\right)$ indicating algorithm found a near-optimal solution close to global minimum

---

(6) Parallel Cellular Algorithms & Programs:

```
Function objective-function(x):
    return sum([xi^2 for xi in x])

Function get-neighbours(grid, i, j, grid-size):
    neighbors = []
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if (di != 0 or dj != 0):
                ni = (i + di) mod grid-size
                nj = (j + dj) mod grid-size
                neighbors.append((ni, nj))
    return neighbors

function update-cell(cursol, neighbolsol, perturbation):
    for each dim d in sol
        NewSol = neighborsol + RANDOM(-perturbation range, +range)

return new-solution
```

Page No.
Date. / /

function parallel_cellular_Algorithm(grid_size, max_iter, sol, lb, ub):
    init grid with random sols in [lb, ub]
    init fitness values for all cells using obj_fxn
    set global_best_sol = none
    set global_best_fitness = infinity

    for t = 1 to max_iter:
        create an empty new_grid for updated sols
        for i ← 0 to grid_size - 1:
            for j ← 0 to grid_size - 1:
                neighbors = get_neighbors(grid, i, j, grid_size)
                best neighbors = grid[i][j]
                best_fitness = fitness[i][j]
                for (ni, nj) in neighbors:
                    if fitness[ni][nj] < best fitness
                        best_fitness ← grid[ni][nj]
                        best_neighbor ← fitness[ni][nj]
                new sol = update_rule
                new fit = obj_fxn
                if newfit < fitness[i][j]
                    new_grid[i][j] ← new_sol
                    fitness[i][j] ← new_fit
                grid ← newgrid
                else new grid[i][j] ← grid[i][j]

        grid ← newgrid

        if fitness[i][j] < global_best_fitness:
            global_best_sol = grid[i][j]
            global_best_fitness = fitness[i][j]

    return global_best_sol, global_best_fitness

(7) optimization via gene expression algorithms

```
population ← generate N random chromosomes
max_iter ← no. of iterations
mutation_rate ← rate of change in genetics
crossover_rate ← probability of crossover bw parents
best_sol ← optimal gene
best_fitness ← fitness value of optimal gene

for generation in range (0, max_generations)
    phenotypes ← []
    for each chromosome in population:
        phenotype ← Decode genes of chromosome
        Append phenotype to phenotypes

    fitness_scores ← []
    for each phenotype in phenotypes:
        fitness ← evaluate fitness of phenotype
        Append fitness to fitness_scores
        if fitness < best_fitness:
            best_sol ← phenotype
            best_fitness ← fitness
    parents ← select parents based on fitness_scores

    offspring ← []
    for i in range (0, length(parents), 2)
        if random() < crossover_rate:
            child1, child2 ← perform crossover on
                parents[i] and parents[i+1]
        else:
            child1 ← parents[i]
            child2 ← parents[i+1]
```

Page No.
Date. / /

Append child1 and child2 to offspring

for each child in offspring:
    if random() < mutation_rate:
        mutate child

population ← offspring
print generation and best fitness