LAB 1

GENETIC ALGORITHM

1. Initialize Parameters :
   - Population Size P
   - Mutation Rate m
   - No. of Generation G
   - Bound for x : lower_bound, upper_bound

2. Create initial population :
   population = [random value (lower_bound, upper_bound) for i in range(P)]

3. For generation = 1 to G do :
   (a) Evaluate fitness
   fitness_value = [fitness_function(x) for (x) in population]
   (b) Track best solution :
   best_fitness = max(fitness_values)
   best_solution = population[index of best_fitness in fitness value]
   (c) Create new population :
   new_population = [ ]

While size of new_population < P do :
- Select parents :
  parent 1, parent 2 = roulette_wheel_selection(population, fitness_values)
- perform crossover :
  offspring 1, offspring 2 = crossover(parent 1, parent 2)

- Perform mutation:
  offspring 1 = mutate(offspring 1, lower_bound, upper_bound, M)
  offspring 2 = mutate(offspring 2, lower_bound, upper_bound, M)

- Add offspring to new population:
  new_population.append(offspring 1)
  new_population.append(offspring 2)

- Replace old population with new population:
  population = new_population[:P]

4. Output:
   Return best_solution and best_fitness.

— **Output** —

Best Solution: x = 9.977045552950002
Best Fitness: f(x) = 99.54437968659277

Date / /
Page 3

LAB-2

PARTICLE-SWARM-OPTIMIZATION (PSO)

```
Function Rastrigin(x):
    A = 10
    Return A * length(x) + sum(x - i^2 - A *
                cos(2 * PI * x) FOR each x in x)

Function Particle swarm optimization(func,
    n_particles, n_dimension, n_iterations,
    inertia_weight, cognitive_coeff,
    social_coeff, bounds):
    Initialize lower_bound, upper_bound from bounds
    Initialize position Randomly In [lower_bound,
                upper_bound] For n particles
    Initialize velocities Randomly IN [-1, 1] For n-particle
    Initialize pbest_positions = positions
    Initialize pbest_scores = [func(p) For each p In
                Position]
    g_best_position = pbest_position [INDEX of Min (p_best_
                score)]
    g best_score = MIN(pbest_scores)

For iteration From 1 To n_iteration:
    For i From 1 to n_particles:
        fitness = func(positions[i])
        If fitness < pbest_scores[i]:
            pbest_score[i] = fitness
            pbest_position[i] = position[i]
        If fitness < g best_scores:
            gbest_score = fitness
            g best_position = position[i]
```

For 2 [illegible] n particles:
r1 = Random Value b/w 0 & 1
r2 = " " " 0 & 1
velocities[i] = (inertia-weight * velocities[i] +
cognitive-coeff * r1 * (pbest position[i] - position[i]) +
social-coeff * r2 * (gbest position[i] - position[i]))
position[i] = CLIP(position[i] + velocities[i], lowerbound, upperbound)
print "Best fitness:", gbest-score
Return gbest-position, gbest-score
MAIN
best position, best-score = Particle & Swarm Optimization (Rastrigin, n particle-n-dimension, n-iterations, 0.7, 1.5, 1.5, (-5.12, 5.12))
Print "Best position found:", best position
Print "Best Fitness Score:", best-score

— Output —

Best fitness score: $2.864197 e^{-4}$
Best Position found: [-1.339250e-01 -3.553893e-01]

Date / /
Page 5

LAB-3

ANT-COLONY optimization for the travelling salesman problem:

```
Function eclidean_distance(city1, city2):
    return sqrt((city1.x - city2.x)^2 +
                (city1.y - city2.y)^2)
class ACO:
    function init(cities, num_ants, num_iterations,
                  alpha, beta, rho, Q):
        SET self.cities = cities
        SET self.num_ants = num_ants
        SET self.num_iterations = num_iterations
        SET self.alpha = alpha
        SET self.beta = beta
        SET self.rho = rho
        SET self.num_cities = length(cities)

        Initialize self.pheromone matrix with ones/num_cities
        Initialize self.distance matrix with zeroes

        for i from 0 to num_cities - 1:
            for j from i+1 to num_cities - 1:
                SET self.distances[i][j] = eclideandistance
                                           (cities[i].cities[j])
                SET self.distance[i][j] = self.distance[i][j]

    function choose_next_city(ant, visited):
        SET current_city = last(ant)
        Initialize probabilities list
```

for i from 0 to num-cities - 1:
    if i not in visited:
        set pheromone = self.pheromone [current_city][i] ^ self.alpha.
        set heuristic = (1.0 / self.distance [current_city][i]) ^ self.beta
        Append pheromone * heuristic to probabilities
Set total = sum(probabilities)
if total == 0:
    Return Random city not in visited
Normalize probabilities by dividing each by total
Return Random city based on probabilities

Function construct_solution():
    Initialize ant with random starting city
    Initialize visited set with starting city

    while length(ant) < num-cities:
        set next city = choose_next_city(ant, visited)
        Append next_city to ant
        Add next_city to visited
    Append starting city to ant
    Return ant

Function evaluate_solution(solution):
    set total_distance = 0

    For i from 0 to length(solution) - 2:
        total distance += self.distance[solution[i]]
                                [solution[i+1]]

    Return total distance

function update_pheromone(all_solutions):
Initialize pheromone_delta matrix with 0

function solve():
set best_solution = None
set best_distance = Infinity

For iteration from 0 to num_iterations - 1:
Initialize all_solutions list

call update_pheromone(all_solutions)
print "Iteration", iteration + 1, "best_Distance", best_distance
Return best_solution, best_distance.

#Main
Print best solution and best distance

Output -
Best solution: [0, 1, 3, 4, 2, 0]
Best distance: 12.106549

LAB - 4

CUCKOO SEARCH

```
# Step size based on Levy distribution
function Levy_Flight(λ):
    σ = (Γ(1+λ) * sin(π*λ/2) /
         (Γ((1+λ)/2) * λ * 2^((λ-1)/2)))^(1/λ)
    u = RandomNormal(0, σ)
    v = RandomNormal(0, 1)
    step = u / abs(v)^(1/λ)
    Return step

# Initialize nests randomly within bounds
function cuckoo_search(obj_function, bounds, n=25,
                       pa=0.25, max_iter=100):
    dim = Length(bounds)
    nests = RandomUniform(n, dim)
    for each dimension i:
        nests[:, i] = scale(nests[:, i], bound[i])

    fitness = [obj_function(nest) for nest in nests]

# main loop of optimization
    for iter = 1 to max_iter
        for each nest i in nests:
            new_nest = nests[i] + Levy_Flight(1.5) *
                       RandomNormal(0, 1, dim)
            new_nest = clip(new_nests, bounds)
            new_fitness = obj_function(new_nest)
            If new_fitness < fitness[i]:
                nest[i] = new_nest
                fitness[i] = new_fitness
```

```
# Abandon nests based on probability pa and create new ones.
    abandon_idx = Random.random(n) < pa
    for each nest i in abandon_idx:
    # create a new random nest within bounds
        nest[i] = Random.uniform(bounds)
        fitness[i] = obj_func(nests[i])
# Return best solution found
    best_idx = np.argmin(fitness)
    return nests[best_idx], fitness[best_idx]
def objective_function(x):
    return sum(xi**2 for each xi in x)
bounds = [(-10, 10), (-10, 10)]
best_solution, best_fitness = cuckoo_search(
                                objective_function, bounds)
print("Best Solution:", best_solution
print("Best Fitness:", best_fitness
```

Output

```
Best Solution = [-0.14023741  0.59049343]
Best fitness = 0.368234901
```

Date / /
Page 10

LAB-5

GREY-WOLF-OPTIMIZER (GWO)

Input —
- objective function $f(x)$
- search space bounds $[x_{min}, x_{max}]$.
- No of wolves n
- no of dimension d
- no of iteration t.

① Initialize population:
Randomly generates n wolves within the bounds $[x_{min}, x_{max}]$

② evaluate fitness:
compute fitness $f(x)$ of each wolf

③ Identify hierarchy:
Assign:
Alpha (α): Best wolf (lowest fitness)
Beta (β): second best wolf
Delta (δ): third best wolf

④ Iterative Optimization:
For $t = 1$ to T:
- compute parameter a:

$$a = 2 - 2 \cdot \frac{t}{T}$$

- For each wolf i:
- for each dimension j:
④ compute the influence of the α wolf:

$$D_\alpha = |C_i d_j - x_{i,j}|$$
$$X_i = d_j - A_i \cdot D_\alpha$$

(b) Compute the influence of the Beta wolf:

$$D_\beta = |C_2 \cdot B_j - x_{i,j}|$$
$$X_2 = B_j - A_2 \cdot D_\beta$$

(c) Compute the influence of the delta wolf:

$$D_\delta = |C_3 \cdot \delta_j - x_{i,j}|$$
$$X_3 = \delta_j - A_3 \cdot D_\delta$$

(4) Update position:

$$x_{i,j} = \frac{X_1 + X_2 + X_3}{3}$$

- Enforce bounds:
  Ensure $x_{i,j} \in [x_{min}, x_{max}]$
- Evaluate the fitness of each wolf & update hierarchy (α, β, δ)

(5) Output result:
Return the position of the α wolf & its fitness $f(\alpha)$

Output

Best solution: $[-1.48263e^{-11} \quad -1.24732e^{-11} \quad 1.51277e^{-11} \quad 1.5433e^{-11} \quad 1.16834e^{-11}]$

Best score $= 9.89377e^{-22}$

Date / /
Page 12

LAB-6

PARALLEL - CELLULAR ALGO

```
Function fitness_function(x):
    Return x**2
Initialize num_cells = 10
"  grid_size = 1.0
"  iterations = 100
"  neighborhood_size = 1
Initialize cells as an array of num_cells with
random values b/w -grid_size and grid_size
for each iteration from 1 to iterations DO:
    # Evaluate fitness
    Initialize fitness as an empty array
    For each cell in cells DO:
        fitness[cell_index] = fitness_function(cell)

    # update states
    Initialize new_cells as a copy of cells
    For each cell_index from 0 to num_cell - 1 DO:
        neighbours = cells[max(0, cell_index -
                neighborhood_size) to min(num_cells,
                cell_idx + neighborhood_size + 1)
    # update cell based on neighbor
    new_cells[cell_idx] = average(neighbours) +
            random_value b/w -0.1 to 0.1
    cells = new_cells
# output
best_cell = cell in cells with the min. fitness value
Print "Best solution found:" + best_cell
Print "Fitness:" + fitness_function(best_cell)
```

Output → best solution found: -0.11165744
fitness: 0.012467

LAB-7

GENE EXPRESSION - AIGO

```
Function fitness_function(x):
    Return x^2

function binary_to_decimal(binary_str):
    Return (integer(binary_str, 2) / (2^length(binary_str) - 1)) * 10 - 5

Initialize population_size = 20
    num_genes = 10
    mutation_rate = 0.1
    crossover_rate = 0.7
    generations = 100

Initialize population as an array of population_size
    with random binary strings of length num_genes

For each generation from 1 to generation DO:
    # evaluate fitness
    Initialize fitness as an empty array
    For each individual in population DO:
        fitness[individual_index] = fitness_function(
                binary_to_decimal(individual))
    total_fitness = SUM(fitness)
    Initialize probabilities as an empty array
    For each f in fitness DO:
        probabilities[index] = f / total_fitness

    selected = Random_choice(population, size=pop_size,
                probability=probabilities)
```

#crossover

Initialize offspring as an empty array

For i from 0 to population_size - 1 step 2 Do:

If Random value < crossover_rate then:

point = RandomInteger(1, num_genes)

offspring.append(selected[i][:point] + selected[i+1][point:])

offspring.append(selected[i+1][:point] + selected[i][point:])

Else:

offspring.append(selected[i])

offspring.append(selected[i+1])

#mutation

For i from 0 to population_size - 1 Do:

If RandomValue < mutation_rate then:

point = Random_Integer(0, num_genes - 1)

offspring[i] = offspring[i][:point]) + (If offspring[i][point] = '0', then '1' Else '0') + offspring[i][point+1:])

population = offspring

#output

best_individual = individual in population with min fitness value

best_fitness = fitness_function(binary_to_decimal(best_individual))

Print "Best solution found:" + binary_to_decimal(best_individual)

Print "Fitness:" + best_fitness

Output

Best solution found: -4.872922761

Fitness: 23.741376352