INDEX

Name SANJANA SURESH Sub.

Std.: Div. Roll No.

Telephone No. E-mail ID.

Blood Group. Birth Day.

Sr.No.	Title	Page No.	Sign./Remarks
01	LAB - 01 - 03/10/24 Learnt about all given algorithms		8
02	LAB - 02 24/10/24 Genetic algorithm for optimization problem		10
03	LAB - 03 07/11/24 Particle Swarm Optimization		10 SSB 14/11/24
04	LAB - 04 14/11/24 Ant Colony Optimization		
05	LAB - 05 21/11/24 Cuckoo Search	—	SSB 21/11/24
06	LAB - 06 28/11/24 Grey wolf optimizer		
07	LAB - 07 Parallel Cellular Algorithms		
08	LAB - 08 Optimization via Gene Algorithm Expression		

03/10

ALGORITHM 1

GENETIC ALGORITHM FOR OPTIMIZATION PROBLEMS

* INTRODUCTION

- Genetic algorithm (GA) is a search based optimization technique based on the principles of Genetics and Natural selection.
- used to solve and find optimal / near optimal solution.
- GA simulate the process of natural selection – "survival of the fittest" among individuals of consecutive generations to solve a problem.
- In GA, we have a pool / population of possible solutions to a given problem. These solutions undergo recombination and mutation, producing new children and this process is repeated over a few generations.
- Each individual is assigned a fitness value and fitter individuals are given a higher chance to mate and yield more fitter individuals.

Population Initialization → Fitness factor → Crossover → mutation → Selection

* APPLICATION

① Optimization : GA are used where we have to maximize / minimize a given objective function value under a given set of constraint

② Neural network : GA are used to train recurrant neural network

③ DNA analysis : GA's are used to determine structure of DNA

④ Travelling Salesman Problem.

⑤ Image processing

* ALGORITHM 2 : PARTICLE SWARM OPTIMIZATION (PSO)

- Particle swarm optimization is inspired by nature and is based on the social behaviour of birds in a flock or behaviour of fish.
- Population based algorithm for search and optimization.
- Simulation to discover the pattern in which birds fly and their formations and grouping during flying activity. as they search for food or suitable place to settle.
- Sociobiologists believe a school of fish / flock of birds that move in a group can "profit from the experience of all other members".
- While simulating movement of a flock of birds, we can also imagine each bird is to help us find the optimal solution and the best solution found by flock is the best solution in solution space.
- In PSO, a group of particles is used to represent potential solutions to a problem. Each particle has a position in search space, velocity.
- Social behaviour of the swarm guides the search for optimal solution. Each particle in the swarm is influenced by current position of other particles as well as its own best solution.

* APPLICATIONS

1. Energy storage optimization
2. Flood control and routing
3. Medical image and segmentation

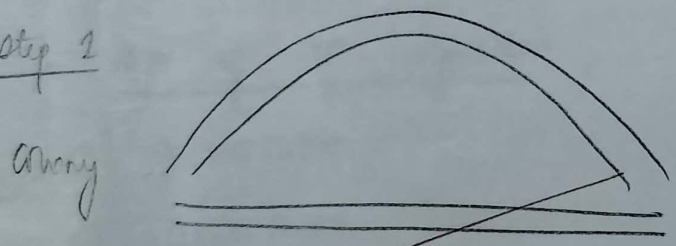
* ALGORITHM 3 : ANT COLONY OPTIMIZATION FOR TRAVELING SALESMAN PROBLEM (ACO)

- ACO is an example of swarm intelligence algorithms. Goal is to design intelligent multi-agent system by observing collective social behaviour of ants.

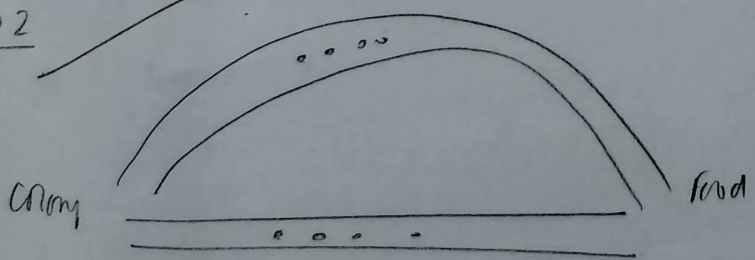
PRINCIPLE OF ACO

- Ants are social insects that live in colonies instead of living individually
- For communication, they use pheromones — chemicals secreted by ants on the soil and ants from the same colony can smell them and follow.
- To get food, ants use shortest path possible from food source to colony. Ants going in search of food secret pheromones and other ants follow this pheromone to follow shortest route. Since more ants use the shortest route so concentration of pheromones increase, rate of evaporation to other paths will decrease — 2 major factors to determine shortest path.

Step 1

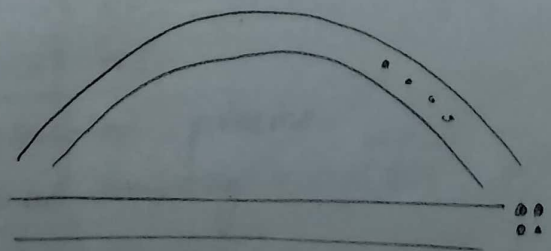


Step 2



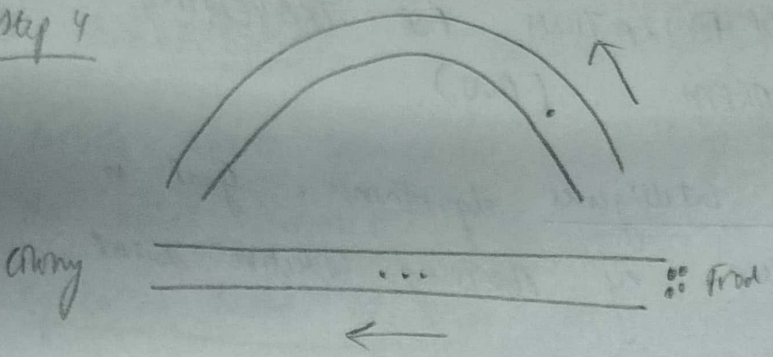
Ants split into 2 groups to find food.

Step 3



Ants which follow shortest route reach food first and pheromone concentration is high.

Step 4



More ants will return to ants using shortest path due to increased pheromones.

* APPLICATIONS

1. Scheduling problem
2. Image Processing
3. Data mining

* ALGORITHM 4 — CUCKOO SEARCH (CS)

INTRODUCTION

- CS is an evolutionary optimization algorithm and was inspired by the species of bird - cuckoo. and their aggressive reproduction strategy.
- cuckoo birds lay their eggs in the nests of other host birds. If the host bird recognizes the eggs as not being their own then it will either throw the egg away or simply empty the nest and build a new one.
- CS is based on three idealized important rules —

(1) Each cuckoo lays one egg at a time and dumps its egg in a randomly chosen nest.

(2) The best nest with high quality eggs will carry over to the next generation.

(3) The number of available hosts' nests is fixed, and egg laid by a cuckoo is discovered by host bird with a probability $p_a \in [0,1]$

- Steps of CS algorithm

(1) Initialization
(2) Levy Flight
(3) Fitness Calculation
(4) Termination

* APPLICATIONS

(1) Optimization problems
(2) Cloud computing
(3) IoT
(4) Pattern recognition.

Complete Remaining by next Algorithm Class

3/10

* ALGORITHM 5 : GREY WOLF OPTIMIZER (GWO)

- GWO algorithm is a nature inpired metaheuristic ~~th~~, based on the behaviour of a pack of wolves to find an optimal solution.
- Grey wolve hunt large preys in pack and rely on cooperation among individual wolve. Two interesting apects —

① Social hierarchy ② hunting mechanism

- The complex social hierarchy includes delta, gamma, beta and alpha wolves representing the best solution candidates at each iteration.
- To find the best answer, the hunting behaviour of grey wolves are picked. The beta and gamma wolve follow the alpha wolves lead when they circle their meal. The delta wolf is the one who attacks first.
- Workflow

Initialization → Fitness Evaluation → Encircling prey → Attack

↓

Update position ← Check fitness ← Stopping criterion ← Output

* APPLICATIONS

① Data mining
② Image and signal processing
③ Energy management
④ Machine Learning

ALGORITHM 6 – PARALLEL CELLULAR ALGORITHMS

- Parallel cellular algorithms are computational models based on the idea of dividing a problem into smaller units or cells that operate simultaneously and often independently.
- Inspired by cellular automata, where the cells interact with their neighbours based on local rules.
- Parallization in cellular algorithms is achieved by distributing these cells across multiple processing units, allowing for concurrent execution of many operations.
- Two main concepts – ① Cellular Automata
  ② Parallel processing

* APPLICATIONS

① Image procesing
② Fluid dynamics
③ Biological system
④ Physics simulation

ALGORITHM 7 – OPTIMIZATION VIA GENE EXPRESSION ALGORITHMS

- GEA (Gene Expresion Algorithm) is inspired by biological process, particularly gene expresion in living organisms and belong to family of evolutionary algorithms.
- GEA's optimize problems by mimicking the way genes in DNA express themselves to develop specific traits in organisms.

* Optimization Process.

Population Initialization → Expression → Fitness Evaluation → Selection

↓

Crossover & Mutation ← Next generation

* Applications

1. Machine Learning
2. Data Mining
3. Resource Allocations
4. Engineering Optimization.

24/10

PROGRAM - 1

GENETIC ALGORITHMS FOR OPTIMIZATION PROBLEMS

(1) Define the problem
- Objective function selected is $f(x) = x^2$

(2) Initialize the parameters such as population size, mutation rate, crossover rate and number of generations.

```
Population_size ← 100
Mutation_rate ← 0.1
Number_of_generation ← 50
Bound ← [-10.0, 10.0]
```

(3) Create initial population of candidate solutions within specified bound

```
def initialize_population (bounds, number_of_generations) :
    for i from 1 to number_of_generation :
        individual = random_uniform (bounds[0], bound[1])
```

(4) Evaluate the fitness of each individual in the population using objective function

```
function evaluate_fitness (population)
    for individual IN population :
        score = objective_function (individual)
```

(5) Select individual based on their fitness to reproduce. Selection is based on the fitness scores using roulette wheel selection method.

```
function roulette_wheel ( population, score)
    total_fitness = sum (scores)
    probabilities = 1 - (score / total_fitness)
    selection = Random ( population, probabilities
```

⑥ Perform crossover between selected individuals to produce offspring. Characteristics of two parents are combined to generate new solutions.

```
function crossover (parent1, parent2, alpha = 0.5) : // combination
    offspring = alpha * parent1 + (1 - alpha) * parent2
```

⑦ Apply mutation to the offspring to maintain genetic diversity

```
function mutation (individual, bounds, mutation_rate)
    if random() < mutation_rate  // to check for mutation
        ~~return~~ mutation occurs.
```

⑧ Entire population is replaced with the newly generated population for the next generation. — iteration

```
function genetic_algorithm (bounds, no_of_generation, population_mutation)
    pop = initialize_population (bounds, population_size)
    for gen in range (no_of_generation)
        scores = evaluate_fitness (pop)

    for _ in range (population_size)
        parent1 = roulette_wheel_selection (pop, scores).
        offspring = crossover (parent1, parent2).
```

⑨ Continue iterations till population number is reached (~50)

Sneha
24/10/24

01/11/24

PROGRAM - 2

PARTICLE SWARM OPTIMIZATION (PSO)

① Define the problem

- create a mathematical function to optimize
- The fitness function selected is

$f1 = x1 + 2*x - x2 + 3$

$f2 = 2 * x1 + x2 - 8$

Objective function = $f1^2 + f2^2$

② Initialize parameters

The number of particles - NumParticles, set the inertia weight as w,
cognitive coefficient c1
social coefficient c2
number of iterations maxIter

③ Initialize particles and PSO parameters

def pso (func, dim, bounds, num_particles = 20, max_iter = 100, w = 0.5, c1 = 1.5, c2 = 1.5):

particles = [Particle (dim, bounds) for _ in range (num_particles)]

- For each particle in Particles, find the particle best position and particle best value., its position and velocity.

④ Iterate and update particles

For a set number of iteration max_iter, evaluate the fitness of each particle using fitness function. Update the global best position and value if any particle has found a better solution.

for particle in particles:
particle. evaluate (func)
if particle.best_value < global_best_value:
position = particle.best_value

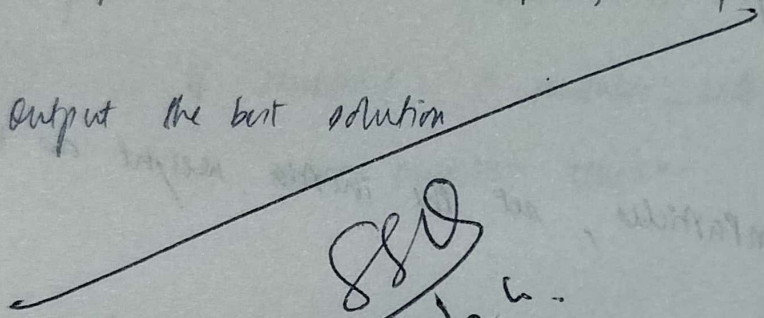
⑤ Update the best solution

After completing the iteration, return the best position and value found by the swarm.

```
for particle in particles:
    particle.update_velocity(global_best_position, w, c1, c2)
    particle.update_position(bounds)
```

```
best_position, best_value = pso(fitness_function, dim, bounds)
```

⑤ Output the best solution



888

7/11/24

14/11/24

PROGRAM - 3

ANT COLONY OPTIMIZATION

Algorithm

① **Define the problem**

Create a city class to store coordinates and calculate distance

```
class City :
    def distance_to (self, other) :
        return math.sqrt (self.x - other.x) **2 + (self.y - other.y)**2
```

② **Initialize the parameters**

Take input from the user —

num_ants	–	The number of ants.
alpha	–	value for pheromone importance
beta	–	value for heuristic importance
rho	–	Pheromone evaporation rate.
Q	–	Pheromone deposit constant.
max_iter	–	maximum number of iterations.

③ **Construct the solutions**

Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.

```
function construct_solution (cities, alpha, beta, pheromone, heuristic)
    visited = [False] * self.cities.
    current_city = random.randint (0, self.num_cities - 1).
    solution.append (current_city)
    visited [current_city] = True.
```

Start from a random city and for each ~~city~~ step, select a new_city to visit using select_next_city function. Return the sequence.

④ Update pheromone

After all ants have constructed their solutions, update the pheromone levels

```
function update_pheromone (pheromone, solutions, costs, rho, q)
    pheromone_evaporate = (1 - rho) * pheromone.
    for i in range (num_ants):
        pheromone [i][j] += q / cost
        pheromone [j][i] += q / cost.
```

⑤ Iterate.

```
for i in range num_ants.
    solution = construct_solution()
    cost = calculate_cost()
    all_solutions.append (solution)
    all_costs.append (cost)

if cost < best_cost :
    best_solution = solution
    best_cost = cost.
```

⑥ Output the best solution

```
print ("Best solution : {best_solution}")
print ("Best cost : {best_cost}")
```

// α - controls the influence of pheromone trail on the probability of selecting the next city.

higher α $\Rightarrow$ more ants likely to follow

β - controls the influence of heuristic information (inverse of distance b/w cities)

SRB

21/11/24

PROGRAM - 4

CUCKOO SEARCH

① Define the problem

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

② Initialize parameters

```
function cuckooSearch(num_iterations, num_nests, pa):
    nests = random(num_nests)
    fitness = evaluate_fitness(nests)
    pa = 0.25
    best_nest = nests(fitness.indexof min())]
    best_fitness = ~~evaluate~~ fitness[min_index]
```

③ Evaluate fitness

```
for i = 1 to max_iterations:
    for each nest in nests:
        new_nest = nest + levy_fight()
        new_fitness = evaluate_fitness(new_nest).

        if new_fitness < fitness[nest]
            nest = new_nest
            fitness[nest] = new_fitness.
```

④ Abandon worst nests

```
worst_nests = select_worst_nests(fitness, pa)
for nest in worst_nests:
    nest = random_initialization()
    fitness[nest] = evaluate_fitness(nest)
```

(i ⑤ Update best solution

```
current_best_index = index_of_min(fitness)
current_best_fitness = fitness[current_best_index]
if current_best_fitness < best_fitness:
    best_fitness = current_best_fitness
    best_nest = nests[current_best_index].
```

① Return the best solution

```
return best_nest, best_fitness.
```

OUTPUT

Best solution found : [0.00027413]

Best value : 7.5146110733 08e-08.

APPLICATIONS

① Feature selection in ML
② Aerospace engineering - optimizing design of aircraft, wings
③ Training deep neural networks
④ Drug design - optimize molecular structure

SBO
2/11/24

28/11/24

PROGRAM - 5

GREY WOLF OPTIMIZER

① Define the problem

Create a mathematical function to optimize.

```
def fitness_function(x):
    return np.sum(x**2)
```

② Initialize parameters

Set the number of wolves and the number of iterations. and randomly generate the positions of the wolves within the defined search space.

```
def initialize(search_space, num_wolves, num_iterations):
    dimensions = len(search_space)
    search_space = np.array([[-5,5], [-5,5]])
    num_wolves = 10
    max_iterations = 100.
```

③ Initialize population

Generate an initial population of wolves with random positions.

```
wolves = np.zeros((num_wolves, dimensions))
for i in range(num_wolves):
    wolves[i] = np.random.uniform(search_space[:,0],
                                  search_space[:,1],

return wolves
```

④ Evaluate fitness and update positions

```
def gwo_algorithm(search_space, num_wolves, max_iterations):
    alpha_wolf = np.zeros(dimensions)
    beta_wolf = np.zeros(dimensions)
    gamma_wolf = np.zeros(dimensions)

    for iteration in range(max_iteration):
        a = 2 - (iteration / max_iterations) * 2

        for i in range(num_wolves):
            fitness = fitness_function(wolves[i])
            if fitness < alpha_fitness:
                gamma_wolf = beta_wolf.copy()
                gamma_fitness = beta_fitness
                beta_wolf = alpha_wolf.copy()
                beta_fitness = alpha_fitness()
                alpha_wolf = wolves[i].copy()
                alpha_fitness = fitness

            elif fitness < beta_fitness:
                gamma_wolf = beta_wolf.copy()

            elif fitness < gamma_fitness:
                gamma_fitness = fitness.
```

⑤ Update positions of wolves

```
for i in range(num_wolves):
    for j in range(dimensions):
        r1 = np.random.random()
        r2 = np.random.random()

        A1 = 2*a*r1 - a
```

```
D_alpha = np.abs(C1 * alpha_wolf[j] - wolves[i,j])
X1 = alpha_wolf[j] - A1 * D_alpha
X2 = beta_wolf[j] - A2 * D_beta
X3 = gamma_wolf[j] - A3 * D_gamma
wolves[i,j] = (X1 + X2 + X3)/3
```

⑥ Print best solution

```
optimal_solution = gwo_algorithm(search_space, num_wolves, max)
print("Optimal solution:", optimal_solution)
```

OUTPUT

```
Optimal fitness : 2.581673e-30
Optimal solution : [-1.0f909e-15, -1.15122e-15]
```

APPLICATIONS

① Image processing and computer vision
② Medical Image Analysis
③ Network design and optimization

PROGRAM - 6

⑤ PARALLEL CELLULAR ALGORITHMS

Algorithm :

Algorithm cellular Optimization

// Input : Grid Size (G), Dimensionality (D), Search space bounds (S), Max iterations (N), fitness function (F)

Output : Best position and its fitness value.

① Define the problem

```
def fitness_function(position):
    return sum(x ** 2)
```

② Initialize parameters

```
grid_size = (10, 10)
dim = 2
minx, maxx = -10.0, 10.0   # search space bounds
max_iteration = 50
```

③ Initialize population

```
def initialize_population(grid_size, dim, minx, maxx):
    population = np.zeros((grid_size[0], grid_size[1], dim))
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            fitness_grid[i,j] = fitness_function(population[i, j])
```

④ Update status

```
def update_cell(population, fitness_grid, i, j, minx, maxx):
    neighbours = get_neighbours(i, j)
    best_new_position = population[best_neighbour[0], best_neighbour[1] + np.random
    new_position = np.clip(new_positi
```

⑤ Iterate

```
population = initialize_population(grid_size, dim, minx, maxx)
for iteration in range(max_iterations):
    fitness_grid = evaluate_fitness(population)

for i in range(grid_size[0]):
    for j in range(grid_size[1]):
        new_population[i, j] = update_cell(population, fitness_grid, i, j, minx, maxx)
```

⑥ Output the best solution

```
best_position = population[[best_index[0], best_index[1]]
best_fitness = np.min(fitness_grid)
print("Best position found", best_position)
print("Best fitness found", best_fitness)
```

OUTPUT

```
Best position found : [0.03121, 0.08516]
Best fitness found : [6.2945e-05]
```

PROGRAM - 7

(5) OPTIMIZATION VIA GENE EXPRESSION ALGORITHMS

1 Algorithm

Algorithm GeneExpressionAlgorithm
// Input : Population of size (P), Number of genes (G), Mutation rate (
Crossover rate (C), Number of generation (N), Search space
Fitness function (F)

Output : Best solution and its fitness value

(

① Define the problem

```
def fitness_function(x):
    return sum(x_i ** 2)
```

② Initialize parameters

```
population_size = 50
num_genes = 5
mutation_rate = 0.1
crossover_rate = 0.7
num_generations = 100
search_space = (-10, 10)
```

③ Initialize population

```
def initialize_population(pop_size, num_gene, search_space):
    return np.random.uniform(search_space[0], search_space[1])
```

④ Evaluate fitness

```
def evaluate_fitness(population)
    return np.array([fitness_function(individual) for individual in population]).

def tournament_selection(population, fitness):
    for i in range(len(population)):
        selected.append(population[i] if fitness[i] > fitness[i] else population[j])
```

```
def crossover (parent1, parent2, rate) :
    if random.random() < rate :
        child1 = np.concatenate ((parent1[:point], parent2[point:]))
        child2 = np.concatenate ((parent2[:point], parent1[point:]))

def mutate (individual, rate, search_space) :
    for i in range (individual)) :
        if random.random () < rate :
            individual[i] += np.random.normal (0, 1)

    return individual
```

⑤ Iterate

```
for generation in range (num_generation) :
    fitness = evaluate_fitness (population)
    selected_population = tournament_selection (population, fitness)
    ~~crossover~~ next_generation [] == cross_over
    next_generation = [mutate ( ind, mutate_rate, search space)
```

⑥ Output:

```
print (f" Best solution found : {best solution}")
print (f" Best fitness : { best_fitness }")
```