INDEX

NAME: SAMRAAT DABOLAY STD: SEC: E ROLL NO: 236 SUB: BIS LAB

S. No.	Date	Title	Page No.	Teacher's Sign/Remarks
1.	3/10/24	Summary of 7 Algorithms		8
2.	24/10/24	Lab 1: Genetic Algorithm for optimization problems		10
3.	7/11/24	Lab 2: Particle Swarm Optimization		SS 14/11/24
4.	14/11/24	Lab 3: Ant colony Optimization		
5.	21/11/24	Lab 4: Cuckoo Search Algorithm		SS 21/11/24
6.	28/11/24	Lab 5: Grey Wolf Optimization		SS 28/11/24
7.		Lab 6: Parallel cellular Algorithms		
8.		Lab 7: Optimization via Gene Expression		

Algorithm 1 : Genetic Algorithm for Optimisation Problems

3/10/24

Introduction

- It is a search based optimisation technique on principle of genetics and natural selection
- Used to find and solve optimal / non optimal solutions
- Simulates process of natural selection among individuals of consecutive generations
- A population of possible solutions undergo recombinations and mutations, producing new children & this process is repeated over few generations
- Individual is assigned a fitness value & after each generation it yields more fitter individuals

Population Initialization → Fitness factor → Crossover → mutation → selecti

Application

(1) Optimisation : maximise / minimise given objective function under set of constraints

(2) Neural Network

(3) DNA analysis

(4) Travelling salesman

Algorithm 2 Particle Swarm Optimization (PSO)

Introduction

- PSO is based on social behavior of birds in a flock or fish behavior in a shoal
- Used for search and optimization
- Simulation to discover problem in which birds fly & their formations and grouping during flying, or the search their food
- Sociobiologists believe a flock of birds "profit from experience of all other members"
- While simulating movement of a flock of birds, each bird is used to help find optimal solution and this leads to best solution found in solution space
- In PSO, a group of particles is used to represent potential solutions
- Each particle in swarm is influenced by current position of other particles as well as its own best solution

Applications :

1. Energy storage optimization
2. Food control & routing
3. Medical image & segmentation

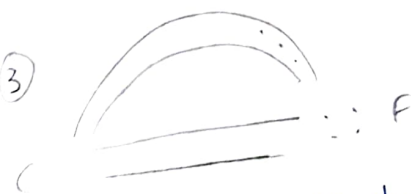
Algorithm 3 : Ant Colony Optimization for Travelling Salesman Problem (ACO)

Introduction :

- It is a swarm intelligence algorithm
- Goal is to design multi-agent system by observing behaviours of ants

Principle of ACO :

- Ants are social insects that live in colonies
- For communication, they use pheromones - chemical secreted by ants on soil
- To get food, ants use shortest path possible from food source to colony
- Each ant passes shortest path, so highest pheromones for that path and other paths decrease
- These two factors used to find shortest path



shortest path reach first & release pheromones

ants split to find food

more ants returning faster to C because more pheromones

Applications

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

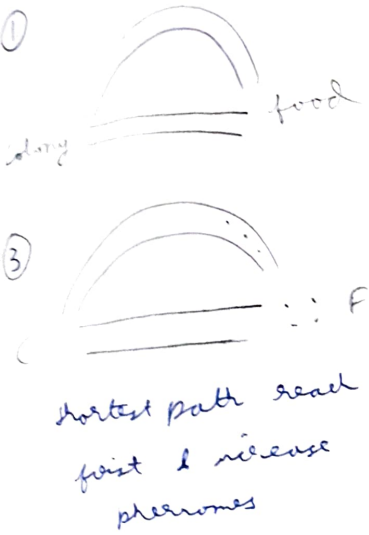
Algorithm 3 : Ant Colony Optimization for travelling Salesman Problem (ACO)

Introduction :

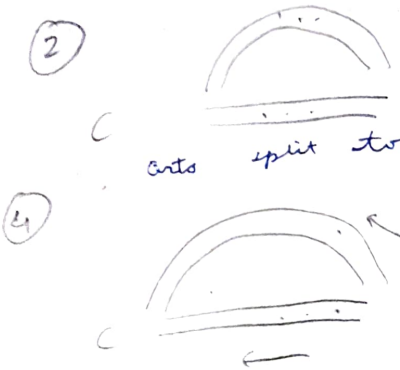
- It is a swarm intelligence algorithm
- Goal is to design multi-agent system by observing behaviour of ants

Principle of ACO :

- Ants are social insects that live in colonies
- For communication, they use pheromones - chemical secreted by ants on soil
- To get food, ants use shortest path possible from food source to colony
- Each ant ~~partak~~ ~~issas~~ chooses shortest path, so highest pheromones for that path and other paths decrease
- These two factors used to find shortest path



shortest path reach first & release pheromones



ants split to find food

more ants returning faster to C because more pheromones

Applications

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

Algorithm 4 - Cuckoo Search (CS)

Introduction

- CS is an evolutionary optimisation algorithm & was inspired by species of bird - cuckoo, & their aggressive reproduction strategy
- They lay their eggs in nests of other host birds If host bird recognises eggs as not their own, they will throw it away or build a new one
- Rules

① Each cuckoo lays 1 egg at a time and randomly chooses nest

② Best nest with highest quality eggs will carry over to next generation

③ The number of available host nest is fixed & egg laid by a cuckoo & egg laid is discovered by host bird with a probability of $P_a \in [0, 1]$

Steps

① Initialization
② Levy flight
③ Fitness calculation
④ Termination

Applications

① Optimisation
② Cloud computing
③ IoT
④ Pattern recognition

Algorithm 5 : Grey wolf optimizer (GWO)

Introduction :

- Nature inspired metaheuristic based on the behaviour of pack of wolves to find optimal solution
- Hunt in large packs & rely on cooperation among individual wolves
  (1) Social hierarchy (2) hunting mechanism
- Complex social hierarchy includes delta, gamma, beta & alpha wolves representing best solution candidates at each iteration
- To find best answer, the hunting behaviour of grey wolves are picked. The beta & gamma wolves follow the alpha wolves' lead when they circle their meal, delta wolf attacks first.

Workflow :

Initialization → Fitness Evaluation → Encircling prey → Attack → update position → check fitness → stopping criterion → Output

Applications :

(1) Data mining
(2) Image & signal processing
(3) Energy management
(4) Machine learning

Algorithm 6 : Parallel Cellular Algorithms

Introduction :

- Parallel cellular algorithm are computational model based on the idea of dividing a problem into smaller units or cells that operate simultaneously & oper independently

- Inspired by cellular automata, where the cells interact with their neighbours based on local rules

- Parallelization in cellular algorithm 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 processing
② Fluid dynamics
③ Biological System
④ Physics simulation

Algorithm 7 : Optimization via Gene Expression Algorithms

Introduction :

- GEA is inspired by biological processes, particularly gene expression in living organisms and belong to family of evolutionary algorithm
- GEA's optimise problems by mimicking the way genes in DNA express the metrics to develop specific ~~traits~~ traits in organisms

Optimization Process :

Population initialisation → Expression → Fitness Evaluation

↓

Selection ← Crossover & mutation ← Next generation

Applications

1. Machine learning
2. Data mining
3. Resource allocations
4. Engineering optimizations

24/10/24

Algorithm 1 : Genetic Algorithm for Optimization Problems

```
Initialize_population ( bounds, n) {
    random.uniform (bounds[0], bounds[1], 1) }

Evaluate - fitness (population) {
    ~~objective~~ function (i) for i in population }

Roulette - wheel ( pop, scores ) {
    total ← sum (scores)
    prob ← 1 - score / total for each score
    selection ← random.choice (len(pop), p)
    p ← prob / sum(prob)
    return pop [ selection ]

Crossover ( p1, p2, α ):
    return offspring = alpha * p1[0] + (1 - α) * p2[0]

Mutation ( individuals, bounds, rate ):
    if random() < rate :
        return [ random.uniform (bounds[0], bounds[1]
    return individual

Genetic - Algorithm ( bounds, iter, n-pop, rate ) :
    pop ← initialize population (bounds, n - pop)
    best, best_eval ← pop [0], function (pop [0])
```

```
for gen in range (iters):
    scores ← evaluate - fitness (pop)
    for i in range (n - pop):
        if scores[i] < best - eval:
            best, best - eval ← pop[i], scores[i]

    children = [ ]
    for _ in range (n - pop):
        p1 ← roulette - wheel (pop, scores)
        p2 ← roulette - wheel (pop, scores)

        child ← crossover (p1, p2)
        child ← mutation (child, bounds, rate)
        Add child to children list

        pop ← children
return [best, best - eval]
```

Input :

Range / bounds = [-10, 10]

Total iterations = 50

Population size = 100

Mutation rate = 0.1

Alpha = 0.5

Sneha SB
- 24/10/24

2 7/11/24

Algorithm 2 : Particle swarm Optimization

Eitress - function (x_1, x_2) :

$f_1 \leftarrow x_1 - 2(x_2) + 3$

$f_2 \leftarrow 2x_1 + x_2 - 8$

$z \leftarrow f_1^2 + f_2^2$

return z

Update - velocity (particle, velocity, pbest, gbest) :

Inertia

$w \leftarrow$ random. uniform (w-min, max)

Social & Cognitive components

$c_1 \leftarrow c$

$c_2 \leftarrow c$

$r_1 \leftarrow$ random. uniform (0, max)

$r_2 \leftarrow$ random. uniform (0, max)

new_velocity $\leftarrow$ [0 ..... num_particle]

for i $\leftarrow$ 0 to num particle

new_velocity [i] $\leftarrow$ w * velocity [i] +

c1 * r1 * (pbest [i] - particle [i])

+ c2 * r2 * (gbest [i] - particle [i])

return new_velocity

Update - position (particle, velocity) :

new_particle $\leftarrow$ particle + velocity

return new_particle

pso_2d (population, dimension, pos_min, pos_max, generation, fitness_criteria):

particles ← [[random.uniform (pos_min, pos_max) for j ← 0 to dimension] for i ← 0 to population]

pbest_pos ← particles

pbest_fitness ← fitness_function (p[0], p[1]) for all p in particles

gbest_index ← min (pbest_fitness)

gbest_pos ← pbest_pos [gbest_index]

velocity ← [0 ... dimension] for i in range of population

for t ← 0 to generation

if average (pbest_fitness) <= fitness_criteria

break

else

for n ← 0 to population

velocity [n] ← update_velocity (particles [n], velocity [n], pbest_pos [n], gbest_pos)

particles [n] ← update_position (particles [n], velocity [n])

pbest_fitness ← fitness_function (p[0], p[1])

gbest_index ← min (pbest_fitness)

gbest_pos ← pbest_pos [gbest_index]

2 Inputs :

A

population = 100

diversion = 2

pos - min = -100.0

pos - max = 100

generation = 100

fitness - criterion = 10e-6

SSO

7/11/24.

14/11/24

Algorithm 3 Ant colony Optimization for TSP

```
Inputcity ():
    n ← no of cities
    cities ← list of city
    for i ← 0 to n
        input city[i] x,y coords.
    return cities

Distance(p1, p2):
    return sqrt(sum((p1 - p2) ** 2))

Build_Path(n, pheromone, points, α, β):
    visited ← for i ← 0 to n all false
    visited[current] ← true
    current ← random(n - ~~points~~)
    path ← [current]
    path_length ← 0

    while false is visited:
        probabilities ← [0 .... n] all zeros array
        unvisited ← find point which is not visited
                    using array

        for i, unvisited_point is unvisited:
            pheromone_value ← pheromone[current, unvisited_point] ** α

            distance_value ← distance(points[current], points[unvisited_point]) ** β

            probabilities[i] = pheromone_value / distance_value
```

2

A

Sum and normalise probabilities

next_point ← random (unvisited, p = probabilities)

path.append (next_point)

path_length += distance (points[current_point], points[next_point])

visited[next_point] ← True

current_point ← next_point

return path, path_lengths

Update_pheromones (pheromone, paths, path_lengths, rho, Q):

pheromone ← pheromone * rho

for path, path_length in this arrays:

for i ← 0 to n-1

pheromone[path[i], path[i+1]] ← pheromone + Q/path_length

pheromone[path[-1], path[0]] += Q/path_length

return pheromone

Ant_Colony_Optimization (points, n_ants, n, α, β, rho):

n_points ← len(points)

pheromone ← [(1, 1), (1, 1) ...] for n_points

best_path_length ← ∞

for iter in range of n:

paths ← [ ]

path_lengths ← [ ]

```
for _ in range of n_ants:
    path, path_length ← Build_path(n,
                                   pheromone, α, β, points)
    paths.append(path)
    path_lengths.append(path_length)
    if path_length < best_path_length
        best_path ← path
        best_path_length ← path_length
pheromone ← update_pheromones(pheromones,
                              paths, path_lengths, rho, Q)
return best_path, best_path_length

points ← input_city()
```

Inputs:

points,

n_ants = 10

n_iters = 100

α = 1

β = 1

rho = 0.5

Q = 1

14/11/24.

2 21/11/24

Algorithm 4: Cuckoo Search Algorithm

```
objective_function(x):
    return x[0]**2

levy_flight(num, β=1.5):
    Sigma_u ← (gamma(1+β) * sin(π*β/2)) / (gamma((1+β)/2) * β * 2**((β-1)/2))
    u ← random.normal(0, Sigma_u, num)
    v ← random.normal(0, 1, num)
    return u / |v|**(1/β)

Cuckoo_Search(iter, num_nests, pa=0.25):
    num_dim ← 1
    nests ← random.rand(num_nests, num_dim)
    fitness ← objective_function(nests)
    best_nest ← nests[np.argmin(fitness)]
    best_fitness ← np.min(fitness)
    for _ in range(iter):
        for i ← 0 to num_nests:
            new_nest ← nests[i] + levy_flight(num_dim)
            new_fitness ← objective_function(new_nest)
            if new_fitness < fitness[i]:
                nests[i] ← new_nest
                fitness[i] ← new_fitness
```

```
worst_nests ← argsort(fitness)[-int(pa * num_nests):]
for j in worst_nests:
    nests[j] ← random(num_dim) * 10 - 5
    fitness[j] ← objective_function(nests[j])
current_best ← np.argmin(fitness)
current_best_fitness ← fitness[current_best]
if current_best_fitness < best_fitness:
    best_fitness ← current_best_fitness
    best_nest ← nests[current_best]
return best_nest, best_fitness
```

Inputs: num_iterations = 1000

num_nests = 25

pa = 0.25

β = 1.5

8/8

2 / 11/24

28/11/24

Algorithm 5 : Grey Wolf Optimization

```
Initialize_wolves (search_space, num_wolves):
    dim ← len (search_space)
    wolves ← zeros ((num_wolves, dimension))
    for i ← 0 to num_wolves:
        wolves[i] ← random.uniform (
                    search_space[:, 0],
                    search_space[:, 1])

    return wolves

Fitness_function (x):
    return np.sum(x**2)

GWO_algorithm (search_space, num_wolves,
               max_iterations):

    dim ← len (search_space)
    wolves ← initialize_wolves (search_space,
                                num_wolves)
    alpha ← zeros (dim)
    beta ← zeros (dim)
    gamma ← zeros (dim)
    alpha_fit = beta_fit = gamma_fit ← float('
    best_fit ← float ('inf')
```

```
for iter in range (max)
    a ← 2 - (iteration / max) * 2

    for i ← 0 to num_wolves:
        fitness ← fitness - function (wolves [i])
        if fitness < alpha - fitness :
            copy beta wolf & fitness to gamma
            copy alpha wolf & fitness to beta
            alpha - wolf ← wolves [i]. copy ( )
            alpha - fitness ← fitness

        elif fitness < beta - fitness :
            copy beta wolf & fitness to gamma
            beta - wolf ← wolves [i]. copy ( )
            beta - fitness ← fitness

        elif fitness < gamma - fitness:
            gamma - wolf ← wolves [i]. copy ( )
            gamma - fitness ← fitness

    if alpha - fitness < best - fitness :
        best - fitness ← alpha - fitness

    for i ← 0 to num - wolves :
        for j ← 0 to dims :
            r1 & r2 random floats
            A1 ← 2*a*r1 - a
            C1 ← 2*r2
            D_alpha ← abs (C1 * alpha - wolf [j]
                          - wolves [i, j])
            X1 ← alpha - wolf [j] - A1 * D_alpha
```

New r1 & r2 random floats

A2 ← 2*a*r1 − a

C2 ← 2*r2

D-beta ← abs (C2 * beta - wolf[j]
- wolves[i,j])

X2 ← beta - wolf[j] - A2 * D-beta

New r1 & r2 random floats

A3 ← 2*a*r1 − a

C3 ← 2*r2

D-gamma ← abs (C3 * gamma - wolf[j]
- wolves[i,j])

X3 ← gamma - wolf[j] - A3 * D-gamma

wolves[i,j] ← (X1+X2+X3)/3

wolves[i,j] ← clip (wolves[i,j],
search-space[j,0],
search-space[j,1])

Inputs:

search-space = array([-5,5], [-5, 5])

num-wolves = 10

max-iter = 100

Algorithm 6 Parallel Cellular Algorithms

```
custom_function(x):
    return sum(x ** 2)

Initialize_population(grid_size, dim, lower, upper)
    population ← random(lower, upper, (grid_size,
                 grid_size, dim))
    return population

Evaluate_fitness(population, fitness_function):
    fitness ← zeros(population.shape[:-1])
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i][j] ← ~~fitness~~ custom_function
                            (population[i,j])

    return fitness

Update_States(pop, fit, radius, lower, upper):
    grid_size, _, dim ← ~~population~~.shape
    new_pop ← pop.copy()
    for i in 0 to grid_size:
        for j in 0 to grid_size:
            neighbours ← get_neighbors(i, j, grid_size,
                         radius)
            best_neighbor ← None
            best_fitness ← float('inf')

            for ni, nj in neighbors:
                if fitness[ni, nj] < best_fitness:
                    best_fitness ← fitness[ni, nj]
                    best_neighbor ← (ni, nj)
```

```
        if best_neighbor:
            ni, nj ← best_neighbor
            new_population[i,j] ← pop[ni,nj] +
                                  random(-0.5, 0.5, dim)
            new_population[i,j] ← clip(new_population
                                  [i,j], lower, upper)

    return new_population

get_neighbors(i, j, grid_size, radius):
    neighbors ← []
    for di in range(-radius, radius+1):
        for dj in range(-radius, radius+1):
            ni, nj ← (i+di) % grid_size,
                     (j+dj) % grid_size
            if (di != 0 or dj != 0):
                neighbors.append((ni, nj))
    return neighbors

Parallel_cellular_Algorithm(grid_size, dim, lower,
    upper, max_iters, radius, fitness_func):
    population ← initialize_population(grid_size,
                 dim, lower, upper)
    fitness ← evaluate_fitness(population,
              fitness_function)
    min_fitness ← fitness.min()
    if min_fitness < best_fitness
        best_fitness ← min_fitness
```

```
best_sol ← None
best_fit ← ∞

for iter in max_iters:

    pop ← update_states(population, fitness,
            radius, lower, upper)

    fitness ← evaluate_fitness(pop,
            fitness_function)

    min_fitness ← fitness.min()

    if min_fitness < best_fitness:

        best_fitness ← min_fitness

        best_sol ← population[np.
            unravel_index(fitness.argmin(),
            fitness.shape)]

    print(iteration, fitness)

return best_sol, best_fit
```

Inputs:

grid_size = 10

dim = 1

lower = -5

upper = 5

max_iters = 100

radius = 1

Algorithm 7 : Optimization via Gene Expression

```
Function (x):
    return sum (x ** 2)

Initialize_Population (pop_size, num_genes, lower, upper):
    pop ← random (lower, upper, (pop_size, num_genes))
    return pop

Evaluate_fitness ( pop , ~~fitness~~ function ):
    fitness ← zeros ( pop.shape[0] )
    for i in range pop.shape[0] :
        fitness[i] ← ~~fitness~~ function ( pop[i] )
    return fitness

Selection ( pop , fitness , num_selected ):
    prob ← fitness / fitness.sum()
    indices ← random.choice ( range (len ( pop )),
                size = num_selected , p = prob )
    selected_pop ← pop [ indices ]
    return selected_pop

Crossover ( selected_pop , crossover ):
    new_pop ← [ ]
    num_individuals ← len (selected_pop)
    for i in range 0 to num_individuals - 1 , step 2 :
        p1 , p2 ← selected_pop [ i ], selected_pop [ i+1 ]
        if len (p1) > 1 and random < crossover :
            crossover_point ← random (1, len (p1) - 1)
            child1 ← concatenate ( p1 [: crossover_point
                p2 [ crossover_point : ] )
```

```
        child2 ← concatenate (p2[: crossover_point],
                              p1[crossover_point :])
        new_pop.extend([p1, p2])
    if num_individuals % 2 == 1:
        new_pop.append(selected_pop[-1])

    return new_pop

Mutation(pop, rate, lower, upper):
    for i in range(pop.shape[0]):
        if random < rate:
            gene ← random(0, pop.shape[1] - 1)
            pop[i, gene] ← random.uniform(lower,
                                          upper)

    return pop

tree_expression(individual, function):
    return function(individual)

tree_Expression_Algorithm(pop_size, num_gens, lower, upper,
    mut_gen, mutation_rate, crossover_rate, function):
    pop ← initialize_population(pop_size, num_gens,
                                lower, upper)

    ~~best_sol~~
    best_sol ← NIL
    best_fit ← ∞

    for gen in 0 to max_gen:
        fitness ← evaluate_fitness(pop, function)
        min_fitness ← fitness.min()
        if min_fitness < best_fit:
            best_fit ← min_fitness
            best_sol ← pop[min(fitness)]

        selected_pop ← selection(pop, fitness,
                                 pop_size // 2)
```

offspring_pop ← crossover(selected_pop, crossover_rate)

pop ← mutation(offspring_pop, mutation_rate, lower, upper)

print(generation, fitness)

return best_sol, best_fit

Inputs:

pop_size = 50

num_genes = 1

lower = -5

upper = 5

max_gen = 100

mutation_rate = 0.1

crossover_rate = 0.7

19/12/24