LAB-1

Summarising optimisation problems:

1. Genetic Algorithm

It simulates the process of natural selection. A random population p is selected and a string of random characters represent chromosones. Fitness of population is determined, selection of the fittest is performed then crossover & mutation is performd until the best fitness level is reached.

2. Particle Swarm Optimisation

It is inspired by the social behaviour of bird flocking or fish schooling.
Each particle with the swarm evaluates its position with the help of a fitness function, adjusting their velocity and position based on best position and converge on most optimal solution.

3. Ant Colony Optimisation

It is based on the idea of foraging of ants, which is used to find solutions for the complex travelling salesmans problem. It executes by mimicking the behaviour of how ants choose the shortest path between their food sources and their nest.

Introduction to Genetic Algorithms

- **Historical content:**
  In the early 1950s, experiments in AI and evolutionary computation, with Nils Barcelli creating the first software mimicking reproduction and mutation.
  Genetic algorithms were developed by John Holland in the 1960s inspired by the principles of natural selection & evolution (Darwin's theory).

- **Abstract**
  GAs are powerful optimisation tools inspired by the processes of natural evolution (selection, mutation & crossover). By simulating the process of natural selection, GAs search for optimal solutions.

- **Working Process (with terminologies):**
  In a GA, the **fitness function** first evaluates how well each chromosome performs in solving the problem.
  - Then **chromosomes**, which are potential solutions encoded as arrays, passed through solution process.
  - **Selection** favours the fitter chromosomes for reproduction, increasing the likelihood of producing better offspring.
  - **Crossover** combines two selected parent chromosomes.

• Mutation then introduces random changes to some chromosomes to achieve genetic diversity & preventing premature convergence.

These steps are performed until optimal solution is reached.

We usually perform maximising or minimising of a particular function. There is a fitness value $f(x)$ associated to every x value & a selection probability.
Selection is performed wrt the function to find the optimal solution & further crossover & mutation of these 5 bit string are performd to generate new fitter population.

Date - 16/10/24

PROGRAM - 1

Program - Implement a Genetic Algorithm using python to solve basic optimisation problem.

```
import numpy as np
def func(x):
    return x**2
Population_size = 30
num_generation = 50
mutation_rate = 0.1
crossover_rate = 0.1
range_min, range_max = -10, 10

def initialise_population(size, min_val, max_val):
    return np.random.uniform(minval, maxval, size)

def evaluate_fitness(population):
    return np.array([func(x) for x in population])

def select_parents(Population, fitness)
    total_fitness = np.sum(fitness)
    selection_probs = fitness / total fitness
    return np.random.choice(population, size=2
                            P = selection_probs)

def crossover(Parent1, parent2):
    if (np.random.rand() < crossover_rate:
        return (parent1 + parent2)/2
    return parent1.
```

```
def mutate(offspring, min_val, max_val):
    if np.random.rand() < mutation_rate:
        return np.random.uniform(min_val, max_val)
    return offspring

def genetic_algorithm():
    population = initialise_population(population_size,
                                       range_min, range_max)
    best_solution = None
    best_fitness = -np.inf

    new_population = []
    for _ in range(population_size):
        parent1, parent2 = select_parents(population, fitness)
        offspring = crossover(parent1, parent2)
        offspring = mutate(offspring, range_min,
                           range_max)

        new_population.append(offspring)

    population = np.array(new_population)
    return best_solution, best_fitness

best_x, best_value = genetic_algorithm()
print("best x: {best_x}, Max. value of f(x):
      {best_value}")
```

Output:

best x: -9.98

Max value of f(x): 99.611

Date: 23/10/24

PROGRAM - 2

Particle Swarm Optimisation

The technique is inspired by the flocking behaviour of birds, swarming of insects and schooling of fish. It is a heurestic search algorithm to find optimal solution to a problem.

Algorithm:

1. Initialisation: Define the problem domain & initialize the population of particles randomly.
2. Evaluation: Evaluate the fitness of each particle (based on its position)
3. Update the personal best: Update the personal best position of each particle.
4. Update the global best position based on the best of all particles.
5. Update velocity and position of each particle based on its current position and velocity, and the personal & global best positions.
6. Termination: Repeat steps 2-5 for a fixed number of iterations or until a stopping criterion is met i.e target fitness value is achieved.

```
import numpy as np

def objective_function(x):
    return np.sum(x**2)

class particle:
    def __init__(self, bounds):
        self.position = np.array([np.random.uniform
                        (bound[0], bound[1]) for bound in bounds])
        self.velocity = np.zeros_like(self.position)
        self.best_pos = self.position.copy()
        self.best_value = float('inf')

    def update_personal_best(self, value):
        if value < self.best_value:
            self.best_value = value
            self.best_pos = self.best_position.copy()

def Pso(objective_function, bounds, num_particles=30,
        num_iterations=20, inertia_weight=0.7,
        cognitive_coefficient=1.5, social_coefficient=1.5):
    particles = [Particle(bounds) for _ in range(num_particles)]
    global_best_position = none
    global_best_value = float('inf')

    for particle in particles:
        fitness_value = objective_function(particle.position)
        particle.update_personal_best(fitness_value)
```

```
if fitness_value < global_best_value
    global_best_value = fitness_value
    global_best_position = particle.position.copy()

for particle in particles:
    inertia_component = inertia_weight * particle_velocity
    cognitive_component = cognitive_coef * np.random.random()
                          * particle.best_position - particle.position)
    social_component = social_coef * np.random.random() *
                       (global_best_position - particle.position)

    particle.velocity = inertia_component + cognitive_compo
                        + social_component

    particle.position += particle.velocity

    for i in range(len(bounds)):
        particle.position[i] = np.clip(particle.position[i],
                               bounds[i][0], bounds[i][1])

    print(f"Iteration {iteration + 1}/{max_iterations}
          Global best value: {global_best_value}")

return global_best_position, global_best_value

bounds = [(-5, 5), (-5, 5)]

best_position, best_value = pso(objective_function, bounds)
```

```
print(f"Best Position: {best_position}")
print(f"Best value: {best_value}")
```

Output:

Iteration 1/20, Global Best Value: 1.0462397424940557

⋮

Best Position → [-0.00476255 -0.001899]

Best value: 2.6288113052694566e-05

Sp.T
23/10/24

PROGRAM - 3

Ant colony optimisation

It is a probabilistic technique for solving computational problems that can be reduced to finding good paths through graphs.

Algorithm:

Initialise necessary parameters
Generate ant population
Find the fitness values associated with each ant
Find best solution through selection methods
Update pheromone trial

Code:

```
import numpy as np
import random

def create_distance_matrix(n_cities):
    np.random.seed(0)
    matrix = np.random.randint(1,100, size=(n_cities,
                                            n_cities))
    np.fill_diagonal(matrix,0)
    return matrix

n_cities = 10
n_ants = 20
n_iterations = 50
alpha = 1
beta = 2
evaporation_rate = 0.5
initial_pheromone = 1
```

```
distance_matrix = create_distance_matrix(n_cities)
pheromone_matrix = np.ones((n_cities, n_cities)) * initial_
                                                    Pheromone

class Ant:
  def __init__(self, n_cities):
    self.n_cities = n_cities
    self.route = []
    self.distance_travelled = 0

def select_next_city(self, current_city, visited):
  probabilities = []
  for city in range(self.n_cities):
    if city not in visited:
      pheromone = pheromone_matrix[current_city][city]
                  ** alpha
      heuristic = (1/distance_matrix[current_city][city]
                  ** beta
      probabilities.append(pheromone * heuristic)
    else
      probabilities.append(0)

  probabilities = np.array(probabilities)/sum(probabilites)
  next_city = np.random.choice(range(self.n_cities),
              p=probabilities)
  return next_city

def find_route(self):
  current_city = random.randint(0, self.n_cities-1)
  self.route = [current_city]
```

```
visited = set(self.route)

self.distance_travelled += distance_matrix[current_city]
                                          [next_city]

def update_pheromone(ants):
    global pheromone_matrix
    pheromone_matrix *= (1 - evaporation_rate)

for ant in ants:
    for i in range(len(ant.route) - 1):
        city_from = ant.route[i]
        city_to = ant.route[i + 1]
        pheromone_matrix[city_from][city_to] += 1.0 /
                                    ant.distance_travelled
        pheromone_matrix[city_to][city_from] += 1.0 /
                                    ant.distance_travelled

def ant_colony_optimization():
    best_route = None
    best_distance = float('inf')

    for iteration in range(n_iterations):
        ants = [Ant(n_cities) for _ in range(n_ants)]

        for ant in ants:
            ant.find_route()

        if ant.distance_travelled < best_distance:
            best_distance = ant.distance_travelled
            best_route = ant.route
```

```
Update pheromones (
print(f"Iteration {iteration+1}: Best distance =
                                {best_distance}")
return best_route, best_distance

best_route, best_distance = ant_colony_optimisation().
print(f"Best route found: {best route} with distance.
                          {best_distance}")
```

Output:

Iteration 1: Best distance = 187

Iteration 2: Best distance = 130

Iteration 3: Best distance = 130

:

Iteration 50: Best distance = 118.

Best route found = [7, 4, 2, 6, 1, 3, 8, 9, 0, 5, 7]
with distance: 118.

Date: 13/11/24 PROGRAM-4

Cuckoo Search Algorithm.

This algorithm is based on the brood parasitic behaviour of cuckoo species.

Usage:

It is used to optimise solutions in cloud computing, data mining, software, IoT and pattern recognition.

Code:

```
import numpy as np
def levy_flight (beta, dim):
    sigma_u = (np.math.gamma (1+beta) * np.sin
              (np.pi * bta /2) / (np.math.gamma((1+beta)/2
              * beta * 2 ** ((beta-1) /2))) ** (1/beta)
    u = np.random.normal (0, sigma_u, dim)
    v = np.random.normal (0, 1, dim)
    step = u / np.abs (v) ** (1/beta)
    return step

def objective_function (x):
    return np.sum (x ** 2)

def cuckoo_search ( obj_func, dim, population_size,
                   max_iter, alpha=0.01, beta=1.5,
                   Pa = 0.25):
```

```
nests = np.random.uniform(-10,10, (population_size, dim))
fitness = np.apply_along_axis(obj_func, 1, nests)

best_nest_idx = np.argmin(fitness)
best_nest = nests[best_nest_idx]
best_fitness = fitness[best_nest_idx]

for iteration in range(num_iter):
    for i in range(population_size):
        step = levy_flight(beta, dim)
        new_nest = nests[i] + alpha * step
        new_nest = np.clip(new_nest, -10, 10)
        new_fitness = obj_func(new_nest)
    if new_fitness < fitness[i]:
        nests[i] = new_nest
        fitness[i] = new_fitness

for i in range(population_size):
    if np.random.rand() < pa:
        nests[i] = np.random.uniform(-10,10,dim)

current_best_idx = np.argmin(fitness)
current_best_fitness = fitness[current_best_idx]

if current_best_fitness < best_fitness:
    best_fitness = current_best_fitness
```

```
        best_nest = nests[current_best_idx]
    print(f"Iteration {iteration+1}/{max_iter}, Best Fitness:
        {best_fitness}")
    return best_nest, best_fitness

#parameters
dim = 2
population_size = 20
max_iter = 100
alpha = 0.01
beta = 1.5
Pa = 0.25

best_solution, best_value = cuckoo_search(objective_function,
    dim, population_size, max_iter, alpha, beta, Pa)

print(f"Best solution found: {best_solution}")
print(f"Objective function value: {best_value}")
```

Output:

```
Iteration 1/100, Best Fitness: 14.158607731707642
Iteration 2/100, Best Fitness: 1.8299126964581 87
Iteration 3/100, Best Fitness: 1.8299126964581 87
        :                :
Iteration 100/100, Best Fitness: 0.17868650188931 65

Best solution found: [3.77880199 -3.5320835]
```

PROGRAM-5

GREY WOLF OPTIMISER

This algorithm is inspired by the social behaviour and hunting technique of grey wolves. The purpose of this algorithm is to find the optimal solution by simulating the social structure and hunting tactics of grey wolves in wild.

Usage: It is used in engineering design problems for structural optimisation and mechanical design. In machine learning, it is used in feature selection and training neural networks.

Code:

```
import numpy as np
def fitness_function(position):
    return np.sum(position**2)

def grey_wolf_optimisation(fitness_function, dim, n_wolves,
max_iter, lower_bound, upper_bound):

    wolves = np.random.uniform(lowerbound, upperbound,
                               (n_wolves, dim))
    fitness = np.apply_along_axis(fitness_function, 1, wolves)

    alpha_pos = wolves[np.argmin(fitness)]
    alpha_score = np.min(fitness)
    fitness[np.argmin(fitness)] = float("inf")

    beta_pos = wolves[np.argmin(fitness)]
    beta_score = np.min(fitness)
```

```
fitness[np.argmin(fitness)] = float("inf")

delta_pos = wolves[np.argmin(fitness)]
delta_score = np.min(fitness)

for t in range(n_wolves):
    r1, r2 = np.random.random(dim), np.random.random(dim)
    A1, C1 = 2*a*r1 - a, 2*r2

    r1, r2 = np.random.random(dim), np.random.random(dim)
    A2, C2 = 2*a*r1 - a, 2*r2

    r1, r2 = np.random.random(dim), np.random.random(dim)
    A3, C3 = 2*a*r1 - a, 2*r2

    D_alpha = np.abs(C1*alpha_pos - wolves[i])
    D_beta = np.abs(C2*beta_pos - wolves[i])
    D_delta = np.abs(C3*delta_pos - wolves[i])

    X1 = alpha_pos - A1*D_alpha
    X2 = beta_pos - A2*D_beta
    X3 = delta_pos - A3*D_delta

    wolves[i] = (X1 + X2 + X3)/3

wolves = np.clip(wolves, lower_bound, upper_bound)

fitness = np.apply_along_axis(fitness_function, 1, wolves)

alpha_idx = np.argmin(fitness)
if fitness[alpha_idx] < alpha_score:
    alpha_pos = wolves[alpha_idx]
    alpha_score = fitness[alpha_idx]
```

```
beta_idx = np.argmin(fitness)
if fitness[beta_idx] < beta_score and fitness[beta_idx]
    > alpha_score:
    beta_pos = wolves[beta_idx]
    beta_score = fitness[beta_idx]

delta_idx = np.argmin(fitness)
if fitness[delta_idx] < delta_score and fitness[delta_idx]
    > beta_score:
    delta_pos = wolves[delta_idx]
    delta_score = fitness[delta_idx]

printf(f"Iteration {t+1}/{max_iter}, Best Fitness:
    {alpha_score}")

return alpha_pos, alpha_score

dim = 5
n_wolves = 30
max_iter = 100
lower_bound = -10
upper_bound = 10

best_position, best_score = grey_wolf_optimisation(fitness_function,
dim, n_wolves, max_iter, lower_bound, upper_bound)
print("Best Position:", best_position)
print("Best Score:", best_score)
```

Output:

```
Best Position: [-0.53923125  0.6760124  -1.84727724
               -3.96609915  -0.30903728]
Best Score: 1.1558289302571314
```

PROGRAM-6

Parallel Cellular Algorithm

It is designed to solve complex problem that can be broken down into local, independent task executed.

The application of this algorithm are in image processing, physical simulation, biological simulation optimisation and machine learning.

Algorithm:

1) Initialisation : Define a grid G of size n x m where each cell G(x,y) has initial state from a set S

$S = \{0, 1\}$ for binary states.

$G = \{G(x,y) \mid 0 \leq x < n, 0 \leq y, n\}$

2) Define neighbourhood : Define neighbourhood N(G(x,y)) for each cell G(x,y). Common choice, include Moore neighbourhood → all 8 surrounding cells 3×3 grid excluding the center.

3) Update Rule : Each cell new state G'(x,y) is complete based on its current state and the state of its neighbour using an update function.

$G'(x,y) = F(G(x,y), \{G(x',y') \mid (x',y) \in N(G(x,y)$

Example updates rule

4) Parallel Update : The update are computed for all cells (x,y) in parallel

$G'(x,y) = F(G(x,y), N(G(x_1, y_1)) \forall (x,y)$

5) Duration : Repeat the update iteration for T time step or until convergence

6) Termination : Stop the algorithm when the predefined no. of iteration is reached or when grid stabilises.

Iteration 1

[[0 0 1 0 1]
 [0 0 0 0 0]
 [0 0 1 1 0]
 [1 0 1 0 1]
 [0 0 0 0 1]]

Iteration 2

[[0 1 1 1 0]
 [1 0 1 0 0]
 [0 1 1 0 1]
 [1 1 1 1 0]
 [0 1 0 1 0]]

PROGRAM - 7

Gene Expression Algorithm

This is an evolutionary algorithm that draws attention from the biological process of gene expression. Its primary purpose is to optimise complex problems and devise solutions computational challenges by simulating the mechanisms of evolution and gene transcription of living organis

Usage: Used in optimisation problems, data minin and machine learning, symbolic regression; control systems, genetic programming, bioinformatics

Algorithm:

1) Generate an initial population P(0) of N chromosomes (individuals). Each chromosome C_i is represented as a string of symbols (genotype)

($i := \{g_1, g_2 \dots g_L\}$, L = length of chromosome

g_i is a gene related from a predefined terminal:

$T = \{x_1, x_2, x_3 \dots x_n\}$ $F = \{+, -, \times, \div\}$

2) Decode the chromosome: Convert the chromosome C_i into a phenotype using a decoding process.

$f(x) = x_1 + (x_2 \times x_3) \dots$

3) Fitness Evaluation: Evaluate the fitness $F(C_i)$ of each chromosome using a problem specific

fitness function.

$$F(C_i) = \frac{1}{n} \sum_{j=1}^{n} (y_j - f(x_j))^2$$

y_i = actual output for input x_j

$f(x_j)$ = Predicted output for C_i

minimize the fitness function.

4. Selection : Select the top-performing chromosomes for a population based on their fitness $F(i)$. Use methods like Roulette wheel selection or tournament selection.

5. Genetic Operations :

i) Crossover :

$$O_1, O_2 = \text{Crossover}(C_1, C_2)$$

ii) Mutation :

$$C_i' = \text{Mutate}(C_i)$$

iii) Transportation : Move parts of the new chromosomes to a new location within the chromosome.

6. Replacement : Replace the old population $P(t)$ with new population $P(t+1)$ based on fitness :

$$P(t+1) = \{O_1, O_2 \cdots O_N\}$$

7. Termination : Repeat steps 2-6 for a no. of generations G or until termination criterion is met.

Initial Population :

Generation 1:

Chromosone 0 : 2509.7331
1 : 2657.1045
2 : Inf
3 : 2509.7331
4 : 2546.5782

Final:
Fitness:
Chromosone 0: 2582.4173
1 : 2582.4173
2 : 3147.2067
3 : 1289.2067
4 : 1416.6804

Best Solution:
Chromosone Fitness : 1259.2067