Roulette wheel selection optimization:

Pseudo code:

Objective function: f(x) = x^2

1. Initialise parameters:
   - population size = 20
   - num_generation = 50
   - mutation_rate = 0.1
   - crossover_rate = 0.7
   - value_range = (-10, 10)

2. Initialise population
   - Create a population
   function initialise_population(size, value_range)
   return np.random.uniform(value_range[0], value_range[1], size)

3. - Evaluate fitness
   Function evaluate_fitness(population):
   return np.array([objective_function(x) for x in population])

4. - Selection
   Function selection(population, fitness):
   probabilities = fitness / fitness.sum()
   return population[np.random.choice(len(population), size=2, p=probabilities)]

5. - crossover
   function crossover(parent1, parent2)
   if np.random() < crossover_rate:
   return(parent1 + parent2) /2
   return parent1 if np.random.rand() < 0.5
   else parent 2

```
// mutation
function mutate(indiv, mut_rate, value_rang
    if np.random.rand() < mut_rate:
        return np.random.uniform(
    value_range[0], value_range[1])
    return individual

// genetic algo
function genetic_algo():
    pop ← init_population(pop_size,
            value_range)
    best_sol ← None
    best_fitness ← -np.inf

    for gen in range(num_gen)
        fitness ← eval_fit(population)
        current_best_i = np.argmax(fit
        if fitness[current_best_i] > best_fit
            best_fit = fitness[current_best
            best_sol = population[current_
                    best_i]

    for _ in range(pop_size):
        par1, par2 = selection(pop, fitnes
        offspring ← crossover(par1, par2)
        offspring ← mutate(offspring,
                mut_rate, value_range)
    pop = np.array(new_pop)
    return best_sol, best_fitness
```

24/10/24

Particle Swarm Optimization

Pseudocode:

1. Initialise swarm:
   - for each particle i in swarm:
     - initialise position (x_i) randomly within problem bounds
     - initialise velocity (v_i) randomly
     - set personal best position pbest_i = x_i
     - set personal best value f(pbest_i)

2. Set global best (gbest)
   - set gbest = position of particle with best value f(pbest_i)

3. for each iteration (t = 1 to max_iteration)
   a. for each particle i:
      - Evaluate fitness f(x_i) at current position x_i
      - If f(x_i) < f(pbest_i)
        - update pbest_i = x_i
        - update f(pbest_i) = f(x_i)
      - If (f(pbest_i) < f(gbest)):
        - update gbest = pbest_i

   b. for each particle i
      - Update velocity v_i using the formula
      v_i(t+1) = w * v_i(t) + c1 * r1 * (pbest_i - x_i) + c2 * r2 * (gbest - x_i)
      where:
      - w = inertia weight (controls exploration vs exploitation)
      - c1, c2 = cognitive and social learning factors

- r1, r2 = random value between 0 and 1
- pbest_i = personal best position of particle i
- gbest = global best position of the swarm.

c. for each particle i:
- update position x_i using formula
  x_i(t+1) = x_i(t) + v_i(t+1)
- Ensure that x_i stays within bounds (boundary handling)

f. Output global best position gbest and its corresponse value f(best).

02/4/24

Ant colony optimization:

Travelling salesman Problem

1) Initialise parameters :
   - n cities, alpha, beta, rho, Q, iterations, ant count, initial-pheromone

2) generate cities with random coordinates

3) calculate distance matrix: distance[i][j] = Euclidean distance between cities i and j

4) Initialise pheromone matrix: pheromone[i][j] = initial pheromone

5) define functions:
   - calc distance(city1, city2): Return Euclidean distance
   - choose next city (current visited, pheromone, distances):
     - Compute probability for each unvisited city
     - Return city with highest probability
   - simulate ants (pheromone, distances):
     - for each ant, construct a tour by selecting cities based on pheromone and distance
     - Calculate tour length and update best tour if shorter
     - Return best tour and best length
   - update pheromones (pheromone, all tours, best_tour, best_length, Q, rho)
     - Evaporate pheromones: pheromone * (1-rho)
     - Deposit pheromone on all tours based on tour quality
     - Reinforce pheromone on best tour

6. Main loop (ACO algorithm):
   - Initialise best_tour and best_length
   - for each iteration (1 to iterations):
     - Run simulate_ants() to get best_tour and best_length
     - update best solution if necessary
     - call update_pheromones()

7 Return best_tour, best_length

8 Display best_tour and plot_path.

→ Initialise parameters
n_cities = 10
α = 1.0
b = 5.0
rho = 0.5
Q = 100
iterations = 100
ant_c = 1000.0
init_pheromone = 1.0

np.random.seed(42)
cities = np.random.rand(n_cities, 2)

→ function calc_distance(city1, city2):
return np.sqrt((city1[0] - city2[0])**2
+ (city1[1] - city2[1])**2)

→ function choose_next_city(ant, city, visited,
pheromone, dist)

```
for _ in range(ant_count):
    visited = set()
    tour = [np.random.randint(0,
            no_cities - 1)]
    visited.add(tour[0])

    while len(visited) < cities:
        cur_city = tour[-1]
        next_city = choose_next_city(
        tour.append(next_city)
        visited.add(next_city)

    if tour_len < best_len:
        best_len = tour_len
        best_tour = tour

def update_pheromone(pheromone, all_cities,
        best_tour, best_len, 0.rho)
    pheromone *= (1 - rho)
    for tour, len in all_tours
        for i in range(len(tour)-1):
            pheromone[tour[i]][tour[i+1]] +=
                Q/len
            pheromone[tour[i+1]][best_tour[i]
                = Q/len
    for i in range(len(best_tour)-1):
        pheromone[best_tour[i]]
            [best_tour[i+1]] += Q/best
        pheromone[best_tour[i+1]]
            [best_tour[i]] = Q/best_len
```

```
def aco_tsp:
    for iteration in range(iterations):
        best_tour, best_length, all_tours
        = simulate_ants(pheromone,
        distance)
        if best_length < best_overall_le
            best_overall_leng = best_len
            best_overall_tour = best_tou
        update_pheromone(pheromone,
        all_tours, best_tour, best_length,
        n_ants)
    return best_overall_tour, best_overall_
    length

best_tour, best_length = aco_tsp()
```

21/11/24

Cuckoo Search (CS)

1. Initialise Parameters:

n = 20

pa = 0.25

max_t = 1000

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

2. Initialise Population (random nest positions):

- for each nest, randomly initialize position within bounds-

- nest[i] = random_pos_within_bounds(bou

3. Evaluate fitness

- for each nest, evaluate its fitness:

fitness[i] = objective_fn(nests[i])

def cuckoo_search:

4. Iterate until max_ite or convergence

(a) - for each nest 'i' perform its levy flights:

- step = levy_flight()

- new_nest[i] = nests[i] + step * (nests[i] - best_nest)

- Ensure the nest is within bounds:

- new_nest[i] = np.clip(new_nest[i], bounds[:,0], bounds[:,1])

(b) Evaluate new solution (fitness):

- Evaluate fitness for each nest:

fitness_new[i] = objective_function (new_nests[i])

(c) update nests:

- If new nest has better fitness:

replace old nest

- if fitness[i] < fitness[i], update

- new

nests[i] and fitne

(d) Abandon worst nests

- Find worst nests based on fitness:
  worst_nests = sorted_indices_of_worst_fitness()
- For each worst nest, replace it with new random position:
  - nests[worst_nests[i]] = random_pos_within_bounds(bounds)
  - fitness[worst_nests[i]] = objective_fn(nests[worst_nests[i]])

(e) Track Better solution:

- If better solution found update best solution:
  - if fitness[min_fitness_indx] < best_fitness, update best_nest and best_fitness

8- Return Best solution
  output best solution and its fitness value
  - return best_nest, best_fitness

Go def levy_flight:
  s = Random sample from Normal distribution:
  $s \sim N(0, 1)$
  step = alpha * s / |s| ^ (1/3)
  new_nest_i = nest_i + step
  return new_nest_i

GREY WOLF OPTIMIZER

classmate
Date
Page 13

objective_function(x)
Input: Positive vector 'x'
Output: fitness value
fitness = np.sum(x**2)
return fitness

initialize_parameters(search_space, num_wolves, dimensions)
Input: Search space (bounds), number of wolves, and dimensions
Output: matrix of initialized positions for wolves
→ extract bounds of search space
lower_bound, upper_bound = search space
→ generate random wolf positions
wolves = np.random.uniform(lower_bound, upper_bound, (num_wolves, dimensions))
→ Return wolves

update_positions(wolves, alpha_pos, beta_pos, delta_pos, a, search_space)
→ get search space bounds
→ for each wolf in wolves:
for each dimension in wolf:
- generate random coefficients
r1, r2 = np.random.random(), np.random.random()
- compute influence from alpha
A1 = 2 * a * r1 - a
C1 = 2 * r2
D_alpha = abs(C1 * alpha_pos[d] - wolves[i, d])
X1 = alpha_pos[d] - A1 * D_alpha

- Repeat for beta and delta values to compute x2 and x3
- Average contributions to update position:

new_position[i, d] =
(x1 + x2 + x3)/3

3. Ensure new position is within bounds:

new_positions[i, :] = np.clip
(new_positions[i, :],
lower_bound, upper_bound,
(~~lower_bound, upper_bound~~))

4. Return updated positions

function grey_wolf_optimizer
(objective_function, search_space,
num_wolves, dimensions,
max_iterations)

→ Initialise wolves

→ compute initial fitness of wolves:

wolves = initialise_parameters
(search_space, num_wolves,
dimensions)

→ fitness = np.apply_along_axis
(objective_function, 1, wolves)

→ Identify alpha, beta, delta wolves

sorted_indices = np.argsort(fitness)

alpha_pos, alpha_score =
wolves[sorted_indices[0]],
fitness[sorted_indices[0]]

beta_pos, beta_score = values
[sorted_indices[1]], fitness[sorted_
indices[1]]

delta_pos, delta_score = values
[sorted_indices[2]], fitness[sorted_
indices[2]]

→ for each iteration (t in range (max_
iteration)):
a = 2 - t * (2 / max_iterations)
update values position.

compute new fitness
update alpha, beta, delta
print progress:

f. Between alpha_pos and alpha_score

Q/?
Iteration 100/100, Best score: 1.9858085 5e-

Best solution found:
[-4.383735e-17, -1.31664 35e-15,
-2.055029e-16, 1.098284e-17]

Best Score:

Parallel Collision Algo

```
import numpy as np
from multiprocessing import Pool

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

def initialize_pop(grid_size, num_cells):
    return [np.random.uniform(-5, 5, grid_size)
            for _ in range(num_cells)]

def update_cell(idx, population, grid_size):
    row, col = divmod(idx, int(np.sqrt(
                len(population))))
    neighbours = []

    if row > 0: neighbours.append(
        population[(row-1)*grid_size + col])
    if row < grid_size - 1 + col])

    if col > 0: neighbours.append(
        population[(row+1)*grid_size + (col
                                        -1)])

    if col < grid_size - 1: neighbours.append(
        population[row*grid_size + (col+1)])

    new_state = population[idx] + 0.5 *
        np.mean(neighbours, axis=0) +
        np.random.uniform(-0.1, 0.1,
            grid_size)
    return new_state
```

```
def parallel_cellular_algo(grid_size,
num_cells, num_iterations):
    popu = initialize_pop(grid_size, num_cells
    best_sol, best_fit = None, float("inf")

for _ in range(num_iterations):
    fitness = [objective_function(cell)
        for cell in population]
    best_idx = np.argmin(fitness)

if fitness[best_idx] < best_fitness:
    best_solution, best_fitness =
        population[best_idx],
        fitness[best_idx]

with Pool() as pool:
    population = pool.starmap(update_cell,
        [(i, pop, grid_size) for in
        range(num_cells)])

return best_solution, best_fitness
```

O/p: Iteration 1:

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

Iteration 2:

```
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

GENE Expression Algo


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

def init_pop():
    return np.random.uniform(
        -10, 10, (50, 5))

def evaluate_fitness(pop):
    fitness = np.array([optimize_fun
        (ind) for ind in pop])
    return fitness

def select_parents(pop, fitness):
    inverted_fitness = 1/(fitness + 1e-6)
    selected_prob = inverted_fitness/
        np.sum(inverted_fitns)
    selected_ind = np.random.choice
        (np.arange(50), size=50,
        p=selected_prob)
    return pop[selected_ind]

def crossover(parents):
    offspring = np.empty_like(parents)
    for i in range(0, 50, 2):
        p1, p2 = parents[i], parents[i+1]
        if np.random.rand() <
CROSSOVER_RATE:
            alpha = np.random.rand()
            offspring[i] = alpha * p1 +
                (1-alpha) * p2
            offspring[i+1] = alpha * p2 +
                (1-alpha) * p1
        else:
            offspring[i], offspring[i+1] =
                p1, p2
```

```
def mutate(offsp):
    for i in range(50)
        if np.random.rand() < 0.1:
            mutation_point = np.random.
                randInt(0, 100)
            offspring[i][mut_point] =
                np.clip(offspring[i]
                [mut_point], -10, 10)
    return offspring

def gene_expression(genes):
    return genes.

def gene_expression_algo():
    pop = init_pop()
    best_sol = NONE
    best_fitness = float('inf')
    for gen in range(50)
        fitness = evaluate_fitness(pop)
        current_best_idx = np.argmin
        if fitness[current_best_idx] < best_fitness
            best_fitness = fitness[current_best_idx]
            best_sol = population[current_best
                                  _idx]
        parents = select_pair(pop, fitness)
        offsp = crossover(parent)
        offsp = mutate(offsp)
        pop = gene_exp(offsp)
    return best_sol.
```

O/P:-

Best soln: Chromosome [1, 3, '+', 2, 1, 9, '*', '*', 3], Fitness: 1259207