Sanketh. M. Hanasi

1BM22CS242

Bio-Inspired Systems LAB

| Sl.No | Date | Title | Remarks |
|---|---|---|---|
| 1 | 3/10/24 | Exploring BIS Algorithms | SSH 14/11/2024 |
| 2 | 24/10/24 | Genetic Algorithm | |
| 3 | 7/11/24 | Partical Swarm Optimization | |
| 4 | 14/11/24 | Ant colony Optimization | SSH 21/11/24 |
| 5 | 21/11/24 | Cuckoo search | |
| 6 | 28/11/24 | Grey wolf optimization | |
| 7 | 18/12/24 | Parallel Cellular Algorithm | |
| 8 | 18/12/24 | Optimization via Gene Expression Algorithms | |

3/10/24

1) Genetic Algorithm for Optimization Problems:

- Genetic Algorithms are optimization techniques which mimic natural selection to evolve a population of potential solutions over generations, aiming to find the optimal solution to a problem.
- The key components of GA:
  - → Population: A set of candidate solutions, each represented as a string of characters, integers, or bits.
  - Fitness Function: A function that evaluates the quality of each candidate solution.
  - selection: The process of selecting the fittest individuals to reproduce
  - crossover: The process of combining the genetic information of two parents to create new offspring
  - Mutation: It introduces genetic diversity by making small changes to an individual's structure
  - Termination: The process continues until a convergence criteria are met.

→ Applications: Neural Networks, Image Processing, wireless sensor networks.

→ Steps:

1. Define the Problem: Identifying the mathematical formula.
2. Initialize Parameters: Population, mutation rate, crossover rate, number of generations.
3. Evaluate fitness: Evaluate the fitness of each individual.
4. Selection: Selecting the best individuals on their fitness based on Roulette wheel, tournament selection.
5. Crossover & Mutation: Create new generation of solutions through crossover & mutation.
6. Repeat: Continue the process until the termination criteria is met.

2) Particle Swarm Optimization Problem:

- Particle swarm optimization (PSO) is a population based optimization problem or technique inspired by the behavior of bird flocking and fish schooling. It is a stochastic optimization algorithm that searches for the optimal solution by iteratively updating the positions of particles in a search space.
- Key components of PSO: Particles, Velocity, Position, Fitness Function, Global best, Personal Best.
- Steps:
  1. Initialization: Initialize a population of particles with random positions & velocities.
  2. Evaluation: Evaluate the fitness of each particle's position using fitness function.
  3. Update velocity: Update the velocity of each particle based on its personal best, global best & current velocity.
  4. Update Position: Update the position of each particle based on its new velocity.
  5. Termination: Stop the algorithm when a satisfactory solution is found or a maximum number of iterations is reached.
- Applications: Robotics - path planning, Neural Network Training

3). Ant Colony optimization for the Travelling Salesman Problem:

. Ant colony optimization (ACO) is a heuristic optimization technique inspired by the foraging behaviour of ants. It is a population-based optimization algorithm that searches for the optimal solution by iteratively updating the pheromone trails & constructing solutions. It is effective for solving travelling salesman problem (TSP), where the goal is to find the shortest possible route that visits a set of cities and returns to the origin city.

- Steps:

1. Initialization: A population of ants is initialized & each ant will construct a solution by traversing.
2. Pheromone Initialization: A pheromone matrix is initialized to represent the attractiveness of each path. Initially all paths have equal pheromone lvls
3. Ant solution construction
4. Update the evaporation.
5. Termination.

Applications: Logistics & supply chain, Telecommunication.

3/10
Complete Algorithm

4) Cuckoo Search:

Cuckoo search is a metaheuristic optimization algorithm inspired by the brood parasitism of certain cuckoo species. Developed by Xin-She Yang & Suash Deb in 2009, it uses behaviors such as:

1. Brood Parasitism: Cuckoos lay eggs in other birds' nests, leading to host birds unknowingly raising cuckoo chicks.
2. Levy Flights: Random walks allow the algo to explore both local & global search spaces preventing it from getting stuck in local optima.
3. Host Bird Behaviour: Poor solutions are replaced, simulating the host abandoning nests with foreign eggs.

Algorithm steps:

1. Initialization: Create initial col$^n$ (nests)
2. Generate New Solutions: Use Levy flights to explore the serch space.
3. Replacement: Randomly replace some nests to maintain diversity.
4. Iteration: Repeat until the stopping condition is met.

Applications:

- engineering design
- machine learning, neural networks
- combinational optimization

5) Grey Wolf Optimizer (GWO):

The Grey Wolf Optimizer algorithm is a swarm intelligence algorithm inspired by the social hierarchy & hunting behaviour of grey wolfes
It models optimization by mimicking the roles of alpha, beta, delta & omega wolves, where alpha & beta wolves lead the search, & beta & delta wolves refine it.

Steps:
1. Define Problem: Set the optimization function
2. Initialize Parameters: Choose the number of wolves & iterations
3. Initialize Population: Randomly generate initial positions of wolves.
4. Evaluate Fitness: Assess each wolf's fitness
5. Update positions: Adjust wolf positions based on alpha beta & delta wolves
6. Iterate: Repeat until convergence or a set number of iterations
7. Output Best solution: Return the optimal solution found

Applications: data analysis, machine learning.

6) Parallel Cellular Algorithm (PCA):

PCAs are inspired by biological cells that operate in a parallel, distributed manner. They use cellular automata principles, where each cell represents a potential solution & interacts with the neighbors, enabling efficient exploration of search space. PCAs are ideal for large-scale optimization problems & can be enhanced with parallel computing.

Steps:

1. Define Problem: Set up the optimization function
2. Initialize Parameters: Choose cell count, grid size, neighborhood & iterations.
3. Initialize Population: Randomly place cells in the solution space.
4. Evaluate Fitness: Assess each cell's fitness.
5. Update States: Adjust cell states based on neighbors & rules.
6. Iterate: Repeat convergence or set iterations.
7. Output Best Solution: Return the best solution found.

7) Gene Expression Algorithm (GEA):

GEA mimic biological gene expression, encoding solutions as genetic sequences that evolve through selection, crossover, mutation & expression. GEA effectively handles complex optimization tasks across various domains.

Steps:

1. Define Problem
2. Initialize Parameters
3. Initialize Population
4. Evaluate Fitness
5. Selection
6. Crossover & mutation
7. Gene Expression
8. Iterate
9. Output Best Solution

Applications: Data Analysis, Machine learning

24/10/24

1) Genetic Algorithm for Optimization Problems:

```
function GeneticAlgorithm():
    Initialize parameters (population_size, mutation_rate,
    crossover_rate, numgenerations, range_min, range_max)
    population = InitializePopulation (population_size,
    range_min, range_max)

    for generation in 1 to num_generations:
        fitness = EvaluateFitness (population)
        new_population = []

        for i from 1 to population_size/2:
            parent1, parent2 = selection (population, fitness)
            offspring1, offspring2 = Crossover(parent1, parent2)
            new_population.append (Mutate (offspring1))
            new_population.append (Mutate (offspring2))

        population = new_population
        best_fitness = Max(fitness)
        best_solution = population [ArgMax (fitness)]

    return best_solution, FitnessFunction (best_solution)
```

function fitness_function (x):
    return $x^2$

function initialize_population (size, range_min, range_max):
    return [random (range_min, range_max) for each i in range (size)]

function selection (population, fitness):
    selection_probs [ ]
    probability = fitness_function / total_fitness
    parent1 = select individual from population based on selection
    parent2 = select individual from population based on selection
    return parent1, parent2

function Crossover (parent1, parent2):
    If Random value < crossover Rate:
        crossoverPoint = Random in btw 1 and chromsemeken-1
        child 1 = parent 1 up to crossoverpoint + Parent2 after crossover
        child 2 = parent 2 upto crossoverpoint + Parent1 after crossover
    Else:
        child 1, child 2 = Parent1, Parent 2
    Return child 1, child 2

Function mutate (chromosome):
    For each bit in chromosome:
        If random value < Mutation Rate:
            Flip the bit
    Return chromosome

7/11/24

## 2) Partical Swarm Optimization :

FUNCTION Rastrigin(x):
return 10* length(x) + sum(xi^2 - 10* cos(2*pi*xi) for each xi in x

Class PSO :
Function init (objective function, n_particles, n_dimensions, max_iter, bounds):
Set self.objective_function = objective_function
Set self.n_particles = n_particles
Set self.n_dimensions = n_dimensions
Set self.max_iter = max_iter
Set self.bounds = bounds
Initialize self.objective positions as random positions in bounds for n_particles x n_dimensions
Initialize self.velocities as random velocities for n_particles x n_dimensions
Initialize self.pbest_positions as self.positions
Initialize self.pbest_values as the fitness of self.positions using objective function

Set self.gbest_position = position with the min fitness
set self.gbest_value = min fitness from pbest_values

Function update_velocity(i):
Randomly generate r1, r2 (uniform random numbers in range [0,1])
inertia = self.w * self.velocities[i]
cognitive = self.c1 * r1 * (self.pbest_positions[i] - self.positions[i])
social = self.c2 * r2 * (self.gbest_position - self.positions[i])
Set self.velocities[i] = inertia + cognitive + social

```
Function update_position(i):
    Update position: self.positions[i] += self.velocities[i]
    self.positions[i] = np.clip(self.positions[i], self.bounds[0],
    self.bounds[1])

Function optimize():
    for t = 1 to max_iter:
        for i = 1 to n_particles:
            self.update_velocity(i)
            self.update_position(i)

            fitness = self.objective_function(self.positions[i])

            if fitness < self.pbest_values[i]:
                self.pbest_values[i] = fitness
                self.pbest_positions[i] = self.positions[i]

    min_pbest_value = min value from pbest_values
    if min_pbest_value < self.gbest_value:
        Set self.gbest_value = min_pbest_value
        Set self.gbest_position = self.positions[i]

    return self.gbest_position, self.gbest_value
```

7/11/24

14/11/24

3) Ant Colony Optimization (ACO):

```
# initialize parameters
n_cities, n_ants, iterations = initialize parameters (ints)
alpha, beta, rho, Q = initialize parameters (floats)

cities = [[random.randint(0,100), random.randint(0,100)]
          for _ in range(num_cities)]
# distance & pheromone matrix
def dist(c1, c2):
    return math.sqrt((c1[0]-c2[0])**2 + (c1[1]-c2[1])**2)

dist_matrix = [[dist(cities[i], cities[j]) for j in range(n_cities)]
               for i in range(n_cities)]

pheromone_matrix = [[1.0 for _ in range(n_cities)] for _ in range
                    (n_cities)]
# best solution initialization
best_path, best_distance = None, float('inf')
# ACO loop:
for _ in range(iterations):
    all_paths, all_distances = [], []
    # ant path construction
    for _ in range(n_ants):
        unvisited = list(range(n_cities))
        path = [curr_city := random.choice(unvisited)]
        unvisited.remove(curr_city)
        # probabilities
        while unvisited:
            probs = [pheromone_matrix[curr_city][i]** alpha
                     * (1.0/dist_matrix[curr_city][i])** beta
                     for i in unvisited
                     ]
```

```
        total_prob = sum(probs)
        probs = [p / total_prob for p in probs]
        next_city = random.choices(unvisited, probs)[0]
        path.append(next_city)
        unvisited.remove(next_city)
        current_city = next_city

    path.append(path[0])
    total_dist = sum(dist_matrix[path[i]][path[i+1]]
                     for i in range(n_cities))
    all_paths.append(path)
    all_distances.append(total_distance)

    if total_distance < best_distance:
        best_path, best_distance = path, total_dist

for i in range(n_cities):
    for j in range(n_cities):
        pheromone_matrix[i][j] *= (1-rho)
        if [i,j] in zip(best_path, best_path[1:]+best_path[0]):
            pheromone_matrix[i][j] += Q / best_distance
```

8/10/24
14/11/24

21/11/24

## Cuckoo search Algorithm:

```
Function CuckooSearch (Func, D, N, MaxIter, Pa, alpha):
    nests = RandomlyInitialize (N, D)
    fitness = EvaluateFitness (nests, Func)
    best_nest = nests [MinFitnessIndex (fitness)]
    best_fitness = Min (fitness)
    for iteration in range (1, MaxIter):
        for each nest i in nests:
            step_size = alpha * LevyFlight(D)
            new_nest = nests[i] + step_size
            new_fitness = Func (new_nest)

            if newFitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

        worst_nests = GetWorstNests (fitness, pa)
        ReplaceWorstNests (worst_nests)

        current_best_index = MinFitnessIndex (fitness)
        if fitness [current_best_index] < best_fitness:
            best_nest = nests [current_best_index]
            best_fitness = fitness [current_best_index]

    Return best_nest, best_fitness

Function LevyFlight(D):
    Return RandomNormal (0, 1, D)
```

```
Function RandomlyInitialize(N,D):
    return Random(N,D)

Function EvaluateFitness(nests, Func):
    return Apply(Func, nests)

Function MinFitnessIndex(fitness):
    return ArgMin(fitness)

Function GetWorstNests(fitness, pa):
    return SortByFitness(fitness)[-int(pa*N):]

Function ReplaceWorstNests(worst_nests):
    for each worst_nest in worst_nests:
        worst_nest = Random(N,D)
```

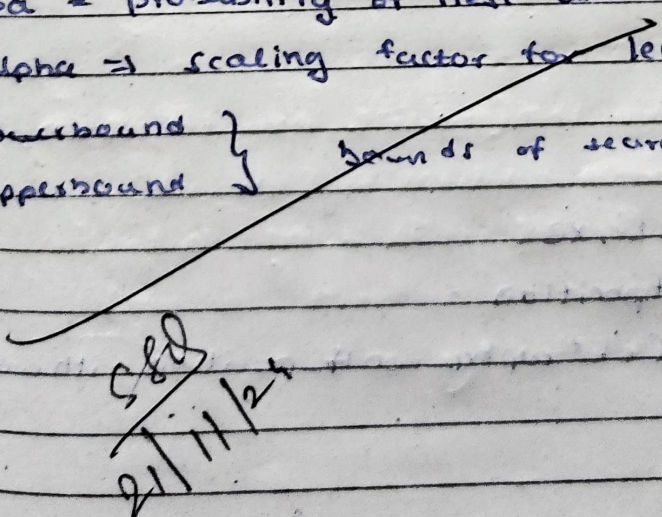
D - dimensionality of the problem

N - no of nests

pa = probability of nest abandonment

alpha => scaling factor for levy flight

lowerbound, upperbound } bounds of search space



6) Grey wolf Optimization:

Randomly initialize Grey wolf population of N particles $X_i$ $(i = 1, 2, \ldots, n)$

Calculate the fitness value of each individuals

sort grey wolf population based on fitness value,

alpha_wolf = wolf with best/least fitness value

beta_wolf = wolf with second least fitness value

gamma_wolf = wolf with third least fitner value

For Iter in range(max_iter):

calculate the value of a

a = 2 * (1 - Iter / max_iter)

For i in range(N):

a. compute the values of $A_1, A_2, A_3$ and $C_1, C_2, C_3$

$A_1 = a * (2 * r1 - 1)$

$A_2 = a * (2 * r2 - 1)$

$A_3 = a * (2 * r3 - 1)$

$C_1 = 2 * r1$

$C_2 = 2 * r2$

$C_3 = 2 * r3$

b. compute x1, x2, x3

x1 = alpha_wolf_position -

A1 * abs (c1 * alpha_wolf position - ith_wolf position)

x2 = beta_wolf_position -

A2 * abs (c2 * beta_wolf_position - ith_wolf_position)

x3 = gama_wolf_position -

A3 * abs (c3 * gamma_wolf_position - ith_wolf_pos)

c. Compute new solution and its fitness

x_new = (x1 + x2 + x3) / 3

fitness(i) = fitness(x_new)

d. Update the ith_wolf greedily

if (fnew < ith_wolf.fitness)

ith_wolf.position = x_new

ith_wolf.fitness = fnew

End-for

// compute new alpha, beta and gamma

sort grey wolf population based on fitness values

alpha_wolf = wolf with least fitness value

beta_wolf = wolf with second least fitness value

gamma_wolf = wolf with third least fitness value

End For

Return best wolf in the population

6) Parallel cellular Algorithms & Program:

```
Function objective_function(x):
    return sum(xi^2 for xi in x)

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

function update_rule(curr_sol, neighbor_sol, perturbation_range):
    for each dimension d in solution:
        new_solution[d] = neighbor_sol[d] + RANDOM(
        -perturbation_range, perturbation_range)
    return new_solution

function Parallel_Cellular_Algorithm(grid_size, max_iter, sol_dim, lb, ub):
    initialize grid with random solutions in [lb, ub]
    initialize fitness values for all cells using objective_function
    set global_best_sol = None
    set global_best_fitness = infinity

    for t = 1 to max_iter:
        create an empty new_grid for updated cells
        for i <- 0 to grid_size - 1:
            for j <- 0 to grid_size - 1:
                neighbors = get_neighbors(grid, i, j, grid_size)
```

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

new_sol = update_rule(grid[i][j], best_neighbor,
                      perturbation_range=0.1)
new_fit = objective_function(new_sol)

if new_fit < fitness[i][i]
    new_grid[i][j] ← new_sol
    fitness[i][i] = new_fit
else
    new_grid[i][i] ← grid[i][j]

if fitness[i][j] < global_best_fitness:
    global_best_sol = grid[i][i]
    global_best_fitness = fitness[i][j]

grid ← new_grid

return global_best_sol, global_best_fitness
```

7) Optimization via Gene Expression Algorithms:

```
function objective_function(sol):
    return sum(x^2 for x in sol)

function gene_exp(gene_seq):
    return gene_seq   #direct mapping

function selection(population, fitness, selecting_size):
    sorted_idx = sorted(range(len(fitness)), key=lambda k: fitness[k])
    return [population[i] for i in sorted_idx[:selection_size]]

function crossover(parent1, parent2, crossover_rate):
    if random.random() < crossover_rate:
        point <- random.randint(1, len(parent1)-1)
        child1 <- parent1[:point] + parent2[point:]
        child2 <- parent2[:point] + parent1[point:]
    else
        child1, child12 = parent1, parent2
    return child1, child2

function mutation(individual, mutation_rate, lb, ub):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = random.uniform(lb, ub)
    return individual

function gene_exp_algo(population_size, gene_len, crossover_rate,
mutation_rate, max_gen, lb, ub):

    population=[
                [random.uniform(lb, ub) for _ in range(gene_len)]
                for _ in range(population_size)
    ]
```

```
fitness = [objective_function(gene_exp(ind)) for ind in population]
best_sol = population[0]
best_fit = fitness[0]

for gen in range(1, max_gen+1):
    selected_popl = selection(population, fitness, population_size
                              //2)

    offspring = []
    for i in range(0, len(selected_popl), 2):
        parent1 = selected_popl[i]
        parent2 = selected_popl[(i+1) % len(selected_popl)]
        child1, child2 = crossover(parent1, parent2, crossover_rate
        offspring.append(child1)
        offspring.append(child2)

    for i in range(len(offspring)):
        offspring[i] = mutation(offspring[i], mutation_rate, best

    population = selected_popl + offspring
    fitness = [objective_function(gene_exp(ind)) for ind in
               population]

    for i in range(len(population)):
        if fitness[i] < best_fitness:
            best_sol = population[i]
            best_fit = fitness[i]

return best_sol, best_fitness
```