

Sanketh. M. Hanasi

IRM22CS242

Bio-Inspired Systems LAB

Sl.No	Date	Title	Remarks
1	3/10/24	Exploring BIS Algorithms	88%
2	24/10/24	Genetic Algorithm	14/11/2024
3	7/11/24	Particle Swarm Optimization	14/11/2024
4	14/11/24	Ant colony Optimization	88%
5	20/11/24	Cuckoo search	21/11/24
6	28/11/24	Grey Wolf Optimization	88%
7	18/12/24	Parallel Cellular Algorithm	88%
8	18/12/24	Optimization via Gene Expression Algorithm	19/12/24

Q) 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 network.

→ Steps:

1. Define the Problem: Identifying the mathematical formula.
2. Initialize Parameters: Population, mutation rate, crossover rate, number of generations.
3. Evaluate fitness: Evaluates the fitness of each individual.
4. Selection: Selecting the next individual 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 Traveling 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 trail & constructing solutions. It is effective for solving Traveling 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 levels.
3. Ant solution construction
4. Update the pheromone
5. Termination.

Applications: Logistics & supply chain, Telecommunications

8/10
Complete the flagon

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 colⁿ (nests)
2. Generate New Solutions: Use Levy flights to explore the search 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
- combinatorial 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 wolves.

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 position 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 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, and 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 until convergence
9. Output Best Solution

~~Applications: Data Analysis, Machine Learning~~

24/10/24

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 numgenerations:

 fitness = EvaluateFitness (population)

 new_population = [InitialPopulation]

 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:

 crossover_point = random int btw 1 and chromosome len

 child1 = parent1 up to crossover point + parent2 after crossover

 child2 = parent2 up to crossover point + parent1 after crossover

 else:

 child1, child2 = parent1, parent2

 return child1, child2

function mutate (chromosome):

 for each bit in chromosome:

 if random value < mutation_rate:
 flip the bit

 return chromosome

mutation_rate = 0.02 * initial_size - initial_size * 0.02 = 0.0196
initial_size = 1000
initial_size * mutation_rate = 1000 * 0.0196 = 19.6
initial_size + initial_size * mutation_rate = 1000 + 19.6 = 1019.6

2) Particle Swarm Optimization:

FUNCTION Rastrigin(x):

return $10 * \text{length}(x) + \sum (x_i^2 - 10 \cos(2\pi x_i))$ for each x_i 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 optimiz.e():
    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)

    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
```

8/8
7/11/29

3) Ant colony optimization (ACO):

initialize parameters

n_cities, n_ants, iterations ← initialize parameters (int)

alpha, beta, rho, Q ← initialize parameters (float)

cities = [random.randint(0,100), random.randint(0,100)]

for i 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, n_cities)]

best solution initialization

best_path, best_distance = None, float('inf')

ACO loop: ~~best path~~ → best path

for _ in range(iterations):

all paths, full distances = [1, E]

ant path construction

for i in range(n_ants):

unvisited = list(range(n_cities))

path = [curr_city := random.choice(unvisited)]

unvisited.remove(curr_city)

probabilities

while unvisited:

prob = [pheromone_matrix[curr_city][i]**alpha

* (Q/dist_matrix[curr_city][i])**beta

for i in unvisited

total_prob = sum(prob)

prob = [p / total_prob for p in prob]

next_city = random.choices(unvisited, prob)[0]

path.append(next_city)

unvisited.remove(next_city)

currcity = nextcity

paths.append(path[0])

total_dist = sum(dist_matrix[path[i]][path[i+1]] for i in range(n-1))

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-1):

for j in range(i+1, n):

pheromone_matrix[i][j] += 1, rho

if [i, j] in zip(best_path, best_path[1:] + best_path[:1]):

(i, j) pheromone_matrix[i][j] += 0.1 / best_distance

BB
19/11/21

: (0) edge 1, initial position

(1, 0, 0) minimum energy state

21/11/24

④ 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_nest_index = GetWorstNest (fitness , pa)

ReplaceWorstNest (worst_nest_index)

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 npoly(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

α - scaling factor for Levy flight

lowerbound } bounds of search space
upperbound }

6.8
2.1 11.24

28/11/24

6) Grey wolf Optimization:

Randomly initialize grey wolf population of N particles x_i ($i=1, 2, \dots, n$)

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 fitness value

for Iter in range(max_iter):

calculate the value of a :

$$a = 2 + (1 - \text{Iter} / \text{max_iter})$$

for i in range(N):

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

$$A_1 = a * (2^*x_1 - 1)$$

$$A_2 = a * (2^*x_2 - 1)$$

$$A_3 = a * (2^*x_3 - 1)$$

$$C_1 = 2^*x_1$$

$$C_2 = 2^*x_2$$

$$C_3 = 2^*x_3$$

b. compute x_1, x_2, x_3

~~$x_1 = \text{alpha_wolf_position}$~~

~~$A_1 + abs(C_1 * \text{alpha_wolf_position} - \text{ith wolf position})$~~

~~$x_2 = \text{beta_wolf_position}$~~

~~$A_2 + abs(C_2 * \text{beta_wolf_position} - \text{ith wolf position})$~~

~~$x_3 = \text{gamma_wolf_position}$~~

AB

A3 * abs (c3 * gamma-wolf-position - ith.wolf.position)

c. Compute new solution and its fitness

$$x_{new} = (x_1 + x_2 + x_3) / 3$$

$$\text{fitness}_{new} = \text{fitness}(\alpha_{new})$$

d. Update the ith-wolf greedily

if ($\text{fitness}_{new} < \text{ith-wolf.fitness}$)

ith-wolf.position = x_{new}

ith-wolf.fitness = fitness_{new}

End-for

compute new alpha, beta and gamma

sort grey wolf population based on fitness values

alpha_wlf = wolf with least fitness value

beta_wlf = wolf with second least fitness value

gamma_wlf = 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 ($\sum x_i^2$ for x_i in x)

Function get_neighbours(grid, i, j, grid-size):

neighbors = []

for di in [-1, 0, 1]:

for dj in [-1, 0, 1]:

if (di != 0 or dj != 0):

$n_i = (i + di) \bmod \text{grid-size}$

$n_j = (j + dj) \bmod \text{grid-size}$

neighbors.append((ni, nj))

return neighbors

function update_rule(curr.sol, neighbor.sol, perturbation)

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, col_dim, dh)

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 soln

for i = 0 to grid-size - 1:

for j = 0 to grid-size - 1:

neighbors = get_neighbours(grid, i, j, grid-size)

best-neighbor = $\text{grid}[i][j]$

best-fitness = $\text{fitness}[i][j]$

for each (n_i, n_j) in neighbors:

if $\text{fitness}[n_i][n_j] < \text{best-fitness}$:

best-fitness $\leftarrow \text{grid}[n_i][n_j]$

best-neighbor $\leftarrow \text{fitness}[n_i][n_j]$

new-sol = update-rule($\text{grid}[i][j]$, best-neighbor,
perturbation-range=0.1)

new-fit = objective-function(new-sol)

if new-fit < fitness[i][j]

new-grid[i][j] \leftarrow new-sol

fitness[i][j] = new-fit

else

new-grid[i][j] \leftarrow grid[i][j]

if $\text{fitness}[i][j] < \text{global-best-fitness}$:

global-best-sol = grid[i][j]

global-best-fitness = fitness[i][j]

grid \leftarrow new-grid

return global-best-sol, global-best-fitness

2) Optimization via Gene Expression Algorithm:

```
function objective_function (sol):
```

```
    return sum ( $x^2$  for  $x$  in sol)
```

```
function gene_exp (gene, seg):
```

```
    return gene-seg #direct mapping
```

```
function selection (population, fitness, selecting_size):
```

```
    sorted_idx = sorted (range (len(fitness)), key=functools.key)
```

```
    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[0:point] + parent2[point:]
```

```
        child2 = parent2[0:point] + parent1[point:]
```

```
    else
```

```
        child1, child2 = 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_elgo (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_expr(ind)) for ind in population]

best_sol = population[0]

best_fit = fitness[0]

for gen in range(1, max_gen + 1):

selected_pop1 = selection(population, fitness, population_size // 2)

offspring = []

for i in range(0, len(selected_pop1), 2):

parent1 = selected_pop1[i]

parent2 = selected_pop1[i+1]

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)

population = selected_pop1 + offspring

fitness = [objective_function(gene_expr(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

88
19/12/24