INDEX

Name Shashank-SP

Std.: Div. Sub. BIS-Lab

Telephone No. E-mail ID. Roll No. 1BM22CS256

Blood Group. Birth Day.

Sr.No.	Title	Page No.	Sign./Remarks
01	Genetic Algorithm for optimization problem		9
02	particles form detection		
03	Ant Colony optimization for the travelling Saleman problem.		
04	cuckoo search		9
05	Grey-wolf optimizer.		10
06	Parallel cellular Algorithm		10
07	optimization via gene Expression Algorithm.		10

Lab-01

Genetic Algorithm

Initialize population with random solutions.
evaluate the fitness of each individual in the population.
while (stopping criteria not met):
Select parents from the population based on fitness
Apply crossover to create offspring.
Apply mutation to offspring with a certain probability
Evaluate the fitness of each offspring.
Select individuals for the next generation from current population and offspring.
update population with new generation
Return the best solution found.

Initialize population: create an initial set of random solutions.

Evaluate fitness: calculate how good each solution is according to a fitness function.

Selection: choose parents based on fitness. (using, roulette wheel selection).

Crossover: combine parents to create new offspring, inheriting traits from both.

Mutation: Apply small random changes to offspring to maintain diversity.

Fitness evaluation - calculate fitness for each offspring.

Survivor selection: choose individuals for the next generation, often based on fitness.

Repeat: continue until a termination condition is met.

Result: the best solution found over generations.

Lab-02

Particle Sworm Detection

1) Define problem

$$f(x) = A \cdot n + \sum_{i=1}^{n} [x_i^2 - A \cdot \cos(2\pi x_i)]$$

where $A = 10$, n = dimensionality of input x

x_i is value ith variable in input vector.

Goal: to minimize the function.

2) Initialize parameters.

n -> no. of vertices

w -> Initial weight

C1 -> Cognitive coefficient (controls particles own best position)

C2 -> Social coeff (best known position by sworn)

3) Initialize Particles

x -> position

y -> velocity

personal best position (pi)

personal best fitness (of (pi))

4) Evaluate fitness

Rastrigin function:

fitness for current position; score = f(x_i);

if score < personal_best_score[i];

update personal_best_position[i] = x_i.

update personal_best_store[i] to score.

5) update velocities & positions

generate random values, r1, r2, b/w 0 & 1.

update velocity (w * v_i(t)) + (1 * r1), (i] - pos[i]) + (2 * r2 * (global_best_pos - pos[i]))

update position.

6) Iterate

for each iteration t (t = 1 to num_iteration);

For each particle i (i = 1 to num_particles);

Calculate fitness score @ new position

~~7) output~~

output best solution

After, return global best position & score

Global minimum occurs @ $x_i = 0$. (all $x_i \to 0$)

i.e $An + [10 - A(n)] = A_n - A_n = 0$

Lab-03 14/11/24

Ant - colony optimization for the Traveling salesman problem

Pseudocode:

Initialize pheromone levels on all edges (I_i,j)

For each ant:

Randomly select a starting city.

For each step in the tour:

Choose the next city based on the pheromone level and distance heuristic

Add the city to the ant's tour

End for

For

while stopping criteria not met (eg. max iterations or convergence):

For each ant:

Evaluate the quality of the solution / tour length)

End for

For each edge (i, j);

.update pheromone on edge (i, j) based on ant tours:

I.ij ← (1-p) * I-ij + ΔI-ij

where ΔI-ij is the pheromone deposited by ants:

ΔI.ij = Σ (1/L.K) where LK is the tour length of ant K

End for.

Apply pheromone Evaporation:

I-ij ← (1-p) * I.ij for all edges (i,j)

End while.

Return the best soln found

Initialization

28/11/24

LAB-04 21/11/24

Cuckoo Search

1, Initialization

* Set parameters n, p_a and max_iter
* Randomly initialize a population of n nests (solutions) x_i for $i = 1, 2, \ldots, n$,
* Evaluate the fitness $f(x_i)$ for each nest using the objective function,

2, Generate new Solutions

* for each nest, generate a new solution x_{new} using levy flights, $x_{new} = x_{current} + \alpha \cdot levy(\lambda)$ where α is a step size scaling factor, and $levy(\lambda)$ is the step drawn from a Levy distribution.
* Evaluate $f(x_{new})$
* Replace x_i with x_{new} if $f(x_{new})$ is better.

3, discovery of Alien Eggs.

* Randomly choose a fraction p_a of the worst nests (based on fitness)
* Replace these nests with new random solution

4, update Best Solution

* Track the best solution x^* with the highest fitness

5, iterate

* Repeat steps 2-4 until the max no. of iterations max_iter is reached or a convergence criterion is satisfied.

6, output the best solution.

* Return x^* the sol'n with best fitness.

Pseudo code

Input: objective function $f(x)$, parameters n, pa, max_iter

output: Best solution x^*,

1, initialize a nests x_i randomly
2, Evaluate fitness $f(x_i)$ for each nest
3, while t < max_iter do,
a, generate new solutions using Levy flights for each nest i:
$x_{new} = x_{current} + \alpha \times levy(\lambda)$
if $f(x_{new}) > f(x_i)$:
$x_i = x_{new}$

b, Replace a fraction pa of the worst nests with new random solutions

c, Keep track of the best soln x^*

Hi, end wile

5, Return x^*

21/11/24

Lab-05

21/11/24

"Grey wolf optimizer"

~~Step~~

"Algorithm"

Step-01 initialize the problem

- Define the optimization problem (objective function $f(x)$)
- Set the population size n, (number of wolves) and maximum number of iterations T.

Step-02: initialize the parameters.

- Randomly initialize the position of wolves in the search space
- Assign initial positions to alpa (α), beta (β) and delta (δ) wolves based on fitness evaluations.

Step-03: update the positions

- For each wolf in the pack calculate its distance D from α, β and δ wolves using

$$D = |C \cdot X_{best} - X|$$

- update the position of each wolf based on the weighted influence of α, β and δ

$$x = \frac{x_1 + x_2 + x_3}{3}$$

where x_1, x_2, x_3 are updated positions using $\alpha, \beta, \& \delta$ wolves.

Step-04: Adjust exploration and exploitation.

- update the coefficient vectors A & C

$$A = 2a \cdot r_1 - a, \quad C = 2 \cdot r_2$$

where

- a decreases linearly from 2 to 0 over iterations to control exploration / exploitation.
- r_1, r_2 are random numbers in $[0, 1]$.

Step-05: Evaluate fitness.

- Evaluate the fitness of each wolf using the objective function $f(x)$.
- update $\alpha, \beta, \& \delta$ wolves if better solutions are found.

Step-06:

- if the maximum number of iterations is reached or the solution has converged, return the alpha wolf (α) as the best soln.
- otherwise Go back to Step 3

pseudo-code

1. initialize the population (positions of wolves) randomly within the search space.

2. Evaluate the fitness of each wolf & identify:

Alpha wolf (best soln so far).

Beta wolf (second best soln)

Delta wolf (third - best soln).

while ($t < 1$);

a. for each wolf in the population

i. Calculate the distance D from alpha, beta, and delta.

D_alpha = |C_alpha * x_alpha - x|

D_beta = |C_beta * x_beta - x|

D_delta = |C_delta * x_delta - x|

ii). update the position of the wolf:

x_new = (x_alpha - A_Alpha * D_alpha x

x_beta - A_beta * D_beta *

x_delta - A_delta * D_delta) / 3

b) update the values of coefficient vectors A & C;

A = 2a * r1 - a

C = 2 * r2

decreases linearly from 2 to 0.
and r1 and r2 are random in [0,1].

c. Evaluate the fitness of all wolves

d. update alpha, beta, & delta wolves based on fitness.

Increment t by 1

5. output the alpha wolf as the best soln.

28/11/24

6) Parallel cellular Algorithm and 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, dj] != [0, 0]:
    ni = (i+di) mod grid_size
    nj = (j+dj) mod grid_size
    neighbors.append([ni, nj])
return neighbors

Function update_rule(current, neighbors_val,
for each dimension d in solution
new_solution[d] = neighbor_val[d] + RANDOM
(-perturbation_range, perturbation_range)
return new_solution.

function parallel_cellular_Algorithm(grid_size,
                max_iter, solution)
initialize grid with random soln in [lb, ub]
initialize fitness values, cells using objective_function
set global best_sol = none
set global best_fitness infinity
```

t21 to max_iter:

create an empty new_grid for updated sol

for i ← 0 to grid_size-1;

for j ← 0 to grid_size-1;

neighbors: get_neighbors (grid, i, j, grid)

best_neighbor_grid[i][j]

best_fitness = fitness[i][j]

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_rate [grid[i][j], best_neighbor,

return nation_range=0.1)

new_fit = objective_function (new_sol)

if new_fit < fitness[i][j]

new_grid[i][j] ← new_sol

fitness[i][j] = new_fit

else

new_grid[i][j] ← grid[i][j]

if fitness[i][j] < global_best_fitness

global_best_sol = grid[i][j]

global_best_fitness = fitness[i][j]

grid2 new_grid

return global_best_sol, global_best_fitness

Applications

Simulation of natural phenomena.

Weather simulation.

Forest fire modelling.

Road traffic modelling

7)

Optimization via Gene Expression Algorithm

```
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 i: fitness[i])
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, child2 = parent1, parent2
return child1, child2

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

function gene_exp_edge(population) else,
    gene, len,
mutation_rate, mutation, lb, ub):
population[random.uniform(lb, ub)
for _ in range(gene)]
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,
```

```
offspring = []
for i in range (0, len (selected, pop[]. 2))
    parent 1 = selected - pop1[i]
    parent 2 = selected - pop1[(i+1) % len
                (selected, pop1)]
    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
```