GENERAL SIR JOHN KOTELAWALA DEFENCE UNIVERSITY

Nature Inspired Computing

CS – 3192

Reflective Journal

Name : G.S.M.Jayasoriya

Reg No. : D / BCS / 22 / 0015

TABLE OF CONTENTS

INTRODUCTION

Nature-Inspired Computing (NIC) is a field of computer science where we create algorithms and systems based on how nature works. It takes ideas from things like how animals behave, how evolution happens, or how physical processes work, to solve complex problems efficiently.

Nature-Inspired Optimization Algorithms (NIOS) are algorithms designed to find the best solution to a problem by mimicking natural processes or behaviors. They are particularly useful for solving complex optimization problems where traditional methods might fail. In this journal, let's explore how NIOAs are applied in various industries, reflecting on their impact and potential to drive innovation.

This journal focuses on real world applications of four widely used Nature-Inspired Optimization Algorithms:

- Genetic Algorithm (GA)
- Simulated Annealing (SA)
- Particle Swarm Optimization (PSO)
- Ant Colony Optimization (ACO)

These algorithms have been successfully applied in real-world problems, such as optimizing transportation routes, improving resource allocation, designing efficient networks, and scheduling complex tasks. Their ability to find near-optimal solutions in challenging scenarios highlights their importance and relevance in modern problem-solving.

GENETIC ALGORITHM (GA)

Genetic Algorithm (GA) is a heuristic optimization technique inspired by the process of natural selection in biological evolution. It is part of a class of evolutionary algorithms that solve complex problems by mimicking the principles of survival of the fittest. As work by generating a population of candidate solutions and evolving them over generations to find the optimal or near-optimal solution.

The core steps of GA include:

- ***Initialization:*** Generate an initial population of solutions randomly.
- ***Evaluation:*** Assess each solution using a fitness function.
- ***Selection:*** Choose the best solutions for reproduction.
- ***Crossover:*** Combine pairs of solutions to create offspring.
- ***Mutation:*** Introduce small random changes to offspring for diversity.
- ***Termination:*** Repeat the process until a stopping criterion is met

Selection is a crucial step in GAs that determines which individuals from the current population will contribute to the next generation. Common selection methods include:

- Roulette Wheel Selection
- Tournament Selection
- Rank-Based Selection

PROBLEM STATEMENT

"Optimizing Protein Folding by Stabilizing 3D Structures"

Protein folding is a complex biological process where a protein's amino acid sequence determines its three-dimensional structure, which is critical for its function. Misfolded proteins can lead to severe diseases like Alzheimer's, Parkinson's, and cystic fibrosis. The problem involves ***predicting the most stable 3D structure of a protein*** given its amino acid sequence, ***minimizing its energy state.***

This is a challenging optimization problem due to the vast number of possible conformations. Genetic Algorithms (GAs) are well-suited for solving this problem because they can efficiently explore large solution spaces and converge towards optimal or near-optimal solutions.

PROPOSED SOLUTION

This code segment imports essential Python libraries for performing various tasks. ***Bio.SeqIO*** is a Biopython module for reading and writing biological sequence data, such as protein or nucleotide sequences, in formats like FASTA or GenBank.

```
import numpy as np
import random
import matplotlib.pyplot as plt
import subprocess
from Bio import SeqIO
```

This is the energy function (***fitness function***) placeholder is used because implementing a real energy function for proteins is complex and typically involves modeling molecular interactions, bond energies, steric clashes, and other factors. To use a real energy function, you would need to integrate with specialized tools like ***PyRosetta***, ***OpenMM***, which are designed for molecular modeling and energy calculations.

```
# Example energy function using simple placeholder
def calculate_energy(sequence, angles):
    # Placeholder for energy calculation
    return sum([np.sin(angle)**2 for angle in angles])
```

A diverse initial population of potential protein conformations is generated randomly to ensure broad exploration of the solution space. Each solution (***chromosome***) is represented as a sequence of torsion angles defining the protein structure.

```
# Initialize population
def initialize_population(size, sequence_length):
    return [np.random.uniform(-180, 180, sequence_length) for _ in
range(size)]
```

The fitness evaluation function evaluates the energy of the protein structure using energy function. Lower energy corresponds to more stable and biologically relevant conformations.

```
# Evaluate fitness
def evaluate_population(population, sequence):
    return [calculate_energy(sequence, individual) for individual in
population]
```

Tournament Selection ensures robust diversity by selecting the best individual from randomly chosen subsets of the population, enhancing convergence towards stable protein conformations.

```python
# Selection (Tournament Selection)
def tournament_selection(population, fitness, k=3):
    selected = []
    for _ in range(len(population)):
        # Randomly choose k individuals and pick the best
        indices = random.sample(range(len(population)), k)
        selected.append(min(indices, key=lambda i: fitness[i]))
    return [population[i] for i in selected]
```

Crossover applies a ***single-point crossover mechanism***, combining segments of two parent solutions to generate offspring, which promotes genetic diversity and explores new folding configurations.

```python
# Crossover (Single-point crossover)
def crossover(parent1, parent2):
    point = random.randint(1, len(parent1) - 1)
    child1 = np.concatenate([parent1[:point], parent2[point:]])
    child2 = np.concatenate([parent2[:point], parent1[point:]])
    return child1, child2
```

Mutation introduces small ***random changes*** to the torsion angles of individuals, bounded within a valid range (-180° to 180°) to maintain realistic protein structures. This step prevents premature convergence and allows the algorithm to explore unexplored areas of the conformational space, contributing to the discovery of the most stable and energy-efficient protein structure.

```python
# Mutation (Step 6: Small random changes)
def mutate(individual, mutation_rate=0.1):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] += np.random.uniform(-10, 10)
            individual[i] = np.clip(individual[i], -180, 180)  # Keep 
angles within valid range
    return individual
```

This function processes a FASTA file containing protein or nucleotide sequences and identifies specific chain identifiers (e.g., "A", "B", "C", "D"). These chain identifiers typically represent different segments or domains of a protein molecule. I used ***Hemoglobin sequence*** file downloaded from the ***RCSB PDB website.***

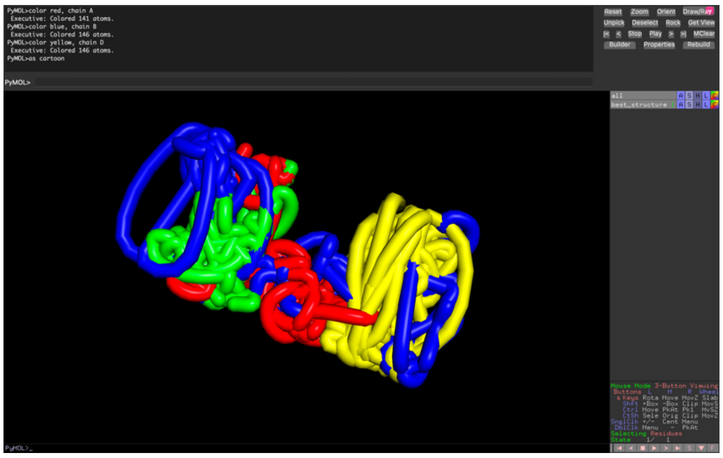
```
def load_chains_from_fasta(file_path):
    chains = {}
    with open(file_path, "r") as f:
        for record in SeqIO.parse(f, "fasta"):
            chain_ids = record.description.split("|")[1].strip()  #
Extract chain info (e.g., "Chains A, C")
            sequence = str(record.seq)
            if "A" in chain_ids:
                chains["A"] = sequence
            if "B" in chain_ids:
                chains["B"] = sequence
            if "C" in chain_ids:
                chains["C"] = sequence
            if "D" in chain_ids:
                chains["D"] = sequence
    return chains
```

This function generates a 3D protein structure file in ***PDB (Protein Data Bank)*** format using provided amino acid sequences and torsion angles.

```
def save_structure_to_pdb(sequences, angles, filename="output.pdb"):
    # Code segment is in the Notebook
    print(f"Structure saved to {filename}")
```

Visualize 3D protein structure saved in a PDB file using ***PyMOL*** molecular visualization tool.

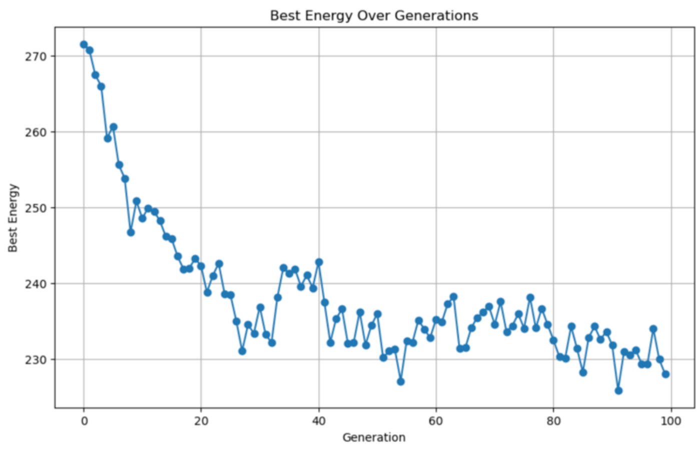
```
# Visualize structure with PyMOL
def visualize_with_pymol(filename="output.pdb"):
    try:
        subprocess.run(["pymol", filename])
    except FileNotFoundError:
        print("PyMOL is not installed or not found in your PATH.")
```



```
PyMOL>color red, chain A
 Executive: Colored 141 atoms.
PyMOL>color blue, chain B
 Executive: Colored 146 atoms.
PyMOL>color yellow, chain D
 Executive: Colored 146 atoms.
PyMOL>as cartoon
PyMOL>
```

This visualization represents the ***predicted folding*** (Best Structure) and organization of hemoglobin's chains, derived from the Genetic Algorithm's output. The different chains of hemoglobin are represented with distinct colors for clarity (e.g.- Chain A is colored red.)

```python
# Visualization of energy over generations
def plot_progress(progress):
    # Code part is in the Notebook
    plt.show()
```

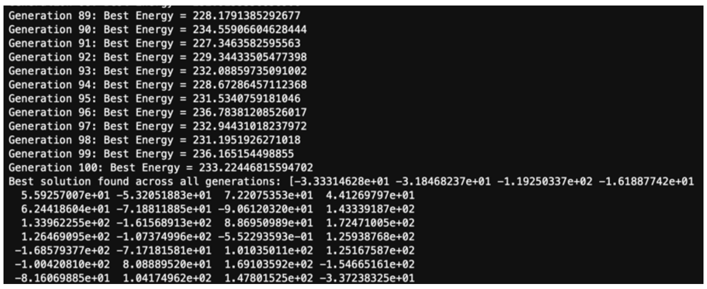


This graph shows the ***best energy (lowest energy)*** achieved in each generation of the Genetic Algorithm for the protein folding problem. Initially, the energy is high due to random conformations, but it steadily decreases as the algorithm progresses, indicating improved stability in the protein structure. Around 20-30 generations, the energy stabilizes due to genetic operations like mutation and crossover. The final low energy values demonstrate the algorithm's success in optimizing the protein's 3D structure towards a ***more stable conformation***.

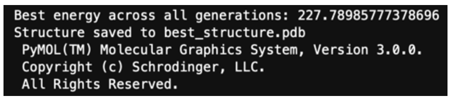
This is the main loop of the Genetic Algorithm (GA) designed to optimize the 3D structure of Hemoglobin using protein sequences from a FASTA file.

```python
if __name__ == "__main__":
    population_size = 20
    generations = 100
    mutation_rate = 0.1

    # Load real protein sequences from FASTA
    fasta_file = "Hemoglobin.fasta"  # Replace with actual FASTA file path
    sequences = load_chains_from_fasta(fasta_file)
    print(f"Loaded sequences for chains: {list(sequences.keys())}")

    # Combine all sequences into one for GA (used for optimization)
    combined_sequence = "".join(sequences.values())

    # Combine lengths of all chains for population initialization
    total_sequence_length = sum(len(seq) for seq in sequences.values())
    population = initialize_population(population_size,
total_sequence_length)

    progress = []  # Track best energy over generations

    # Initialize global best trackers
    global_best_fitness = float("inf")
    global_best_individual = None

    for generation in range(generations):
        # Evaluate population
        fitness = evaluate_population(population, combined_sequence)

        # Check for the global best solution
        if min(fitness) < global_best_fitness:
            global_best_fitness = min(fitness)
            global_best_individual = population[np.argmin(fitness)]

        # Selection
        selected_population = tournament_selection(population, fitness)

        # Crossover
        new_population = []
        for i in range(0, len(selected_population), 2):
            if i + 1 < len(selected_population):
                child1, child2 = crossover(selected_population[i],
selected_population[i + 1])
                new_population.extend([child1, child2])
```

```python
        # Mutation
        new_population = [mutate(individual, mutation_rate) for individual
in new_population]

        # Replace old population with new one
        population = new_population

        # Evaluate and report best fitness
        fitness = evaluate_population(population, combined_sequence)  #
Re-evaluate after mutation
        best_fitness = min(fitness)
        progress.append(best_fitness)
        print(f"Generation {generation + 1}: Best Energy =
{best_fitness}")

    # Final output
    print("Best solution found across all generations:",
global_best_individual)
    print("Best energy across all generations:", global_best_fitness)

    # Save best structure to PDB
    save_structure_to_pdb(sequences, global_best_individual,
filename="best_structure.pdb")

    # Visualize with PyMOL
    visualize_with_pymol("best_structure.pdb")

    plot_progress(progress)
```



```
Generation 89: Best Energy = 228.1791385292677
Generation 90: Best Energy = 234.55906604628444
Generation 91: Best Energy = 227.3463582595563
Generation 92: Best Energy = 229.34433505477398
Generation 93: Best Energy = 232.08859735091002
Generation 94: Best Energy = 228.67286457112368
Generation 95: Best Energy = 231.5340759181046
Generation 96: Best Energy = 236.78381208526017
Generation 97: Best Energy = 232.94431018237972
Generation 98: Best Energy = 231.1951926271018
Generation 99: Best Energy = 236.165154498855
Generation 100: Best Energy = 233.22446815594702
Best solution found across all generations: [-3.33314628e+01 -3.18468237e+01 -1.19250337e+02 -1.61887742e+01
  5.59257007e+01 -5.32051883e+01  7.22075353e+01  4.41269797e+01
  6.24418604e+01 -7.18811885e+01 -9.06120320e+01  1.43339187e+02
  1.33962255e+02 -1.61568913e+02  8.86950989e+01  1.72471005e+02
  1.26469095e+02 -1.07374996e+02 -5.52293593e-01  1.25938768e+02
 -1.68579377e+02 -7.17181581e+01  1.01035011e+02  1.25167587e+02
 -1.00420810e+02  8.08889520e+01  1.69103592e+02 -1.54665161e+02
 -8.16069885e+01  1.04174962e+02  1.47801525e+02 -3.37238325e+01
```



```
Best energy across all generations: 227.78985777378696
Structure saved to best_structure.pdb
 PyMOL(TM) Molecular Graphics System, Version 3.0.0.
 Copyright (c) Schrodinger, LLC.
 All Rights Reserved.
```

These outputs highlight the successful execution of the Genetic Algorithm, demonstrating its ability to minimize the protein's energy and find an optimized structure. The array at the end represents the torsion angles of the protein's best 3D structure. The ***best energy is 227.78***, which represents the most stable conformation found by the algorithm.

SIMULATED ANNEALING (SA)

Simulated Annealing (SA) is an optimization algorithm inspired by the annealing process in metallurgy, where materials are gradually cooled to achieve a stable, low-energy state. It uses the ***Metropolis criterion*** to decide whether to accept a new solution, even if it is worse than the current one, allowing the algorithm to escape local optima. The balance between ***exploration*** and ***exploitation*** is maintained through a ***gradual cooldown*** of the temperature parameter, which reduces the likelihood of accepting worse solutions over time. This makes SA effective for solving complex optimization problems.

PROBLEM STATEMENT

"Efficient Cloud Resource Allocation with Multi-Region Management"

Cloud resource optimization is a critical process for minimizing operational costs while meeting user demands efficiently. This involves determining the ***optimal assignment of cloud users*** to Amazon cloud regions, minimizing latency, reducing operational expenses for maintaining data centers, and avoiding capacity overloads. The solution must determine the optimal allocation of users to cloud regions while ensuring efficiency, cost-effectiveness, and compliance with service-level agreements.

PROPOSED SOLUTION

```python
import folium # Works with geographic data
import random
import math
import pandas as pd
```

This part loads a dataset of AWS cloud regions stored in a CSV file, The dataset, which you obtained from Kaggle, contains details about AWS cloud infrastructure locations.

```python
# Load AWS Cloud Regions Dataset
dataset_path = "aws_cloud_locations.csv"  # Update with the correct path
df = pd.read_csv(dataset_path)
```

```python
# Extract cloud region locations (latitude and longitude)
cloud_regions = list(zip(df["Lat"], df["Long"]))  # Convert to tuples
```

Assigns random capacities (between 300 and 600) and operational costs (between 50 and 150) to cloud regions, simulating their resource and cost characteristics. Additionally, it generates random user locations worldwide.

```
# Assign random capacities and operational costs for the regions
region_capacities = [random.randint(300, 600) for _ in
range(len(cloud_regions))]
region_operational_costs = [random.randint(50, 150) for _ in
range(len(cloud_regions))]
```

```
# Randomly generate user locations worldwide (latitude, longitude)
NUM_USERS = 50
users = [(random.uniform(-60, 60), random.uniform(-180, 180)) for _ in
range(NUM_USERS)]
```

Defines the key parameters for the algorithm. These parameters guide the optimization process, ***balancing exploration*** of the solution space and ***convergence to an optimal solution.***

```
# Parameters for Simulated Annealing
INITIAL_TEMP = 1000
FINAL_TEMP = 1
ALPHA = 0.95
MAX_ITER = 100
```

This function calculates the ***total cost for assigning users to AWS cloud regions*** based on latency, operational costs, and overcapacity penalties. For each user, the function adds the latency (calculated as the distance to the assigned region) to the total cost and tracks the load on each region. It then incorporates the fixed operational costs for each region and applies a ***penalty if a region exceeds its capacity.***

```
# Cost function
def calculate_advanced_cost(region_locations, assignments):
    total_cost = 0
    region_loads = [0] * len(region_locations)

    for user_idx, region_idx in enumerate(assignments):
        latency = haversine_distance(users[user_idx],
region_locations[region_idx])  # Latency as distance
        total_cost += latency  # Add latency as cost
        region_loads[region_idx] += 1  # Increment load on the assigned
region
```

```
    # Add operational costs and overcapacity penalties
    for region_idx, load in enumerate(region_loads):
        total_cost += region_operational_costs[region_idx]  # Fixed
operational cost
        if load > region_capacities[region_idx]:  # Overcapacity penalty
            total_cost += (load - region_capacities[region_idx]) * 50  #
Penalty per extra user

    return total_cost
```

This function calculates the great-circle distance between two points on the Earth's surface, given their latitude and longitude coordinates. It uses the ***Haversine formula***, which accounts for the curvature of the Earth, to compute the shortest distance between two points.

```
# Haversine distance for real-world coordinates
def haversine_distance(coord1, coord2):
    R = 6371  # Earth radius in km
    # Code Segment in the Notebook
    return distance
```

```
# Generate a random initial solution
def initialize_solution():
    assignments = [random.randint(0, len(cloud_regions) - 1) for _ in
range(NUM_USERS)]
    return assignments
```

Creates a ***neighboring solution*** by making a small random change to the current solution. It randomly selects one user and reassigns that user to a new, randomly selected cloud region. This generates a new solution that is similar to the current one, which is essential for ***exploration of the solution space.***

```
# Generate a neighboring solution
def generate_neighbor(current_solution):
    neighbor = current_solution[:]
    user_to_reassign = random.randint(0, NUM_USERS - 1)
    neighbor[user_to_reassign] = random.randint(0, len(cloud_regions) - 1)
    return neighbor
# Visualize the solution on a map
def plot_solution_on_map(users, cloud_regions, solution, region_names):
      # Code Segment in the Notebook
      return cloud_map
```

This is the main function that implements the Simulated Annealing algorithm to find the best solution for assigning users to cloud regions while minimizing costs. It starts with an initial random solution and evaluates its cost.

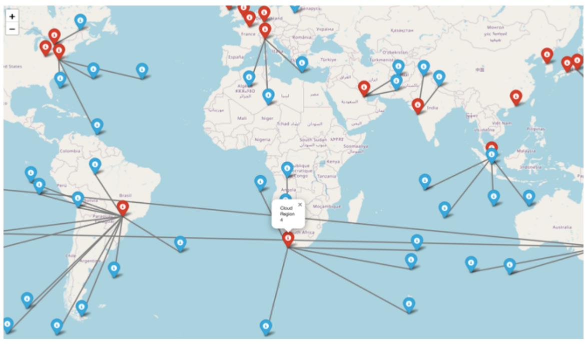
Then, it iteratively generates neighboring solutions and accepts them if they improve the cost or with a certain probability ***(Metropolis criterion)*** to ***avoid getting stuck in local minima*** (from exploration to exploitation).

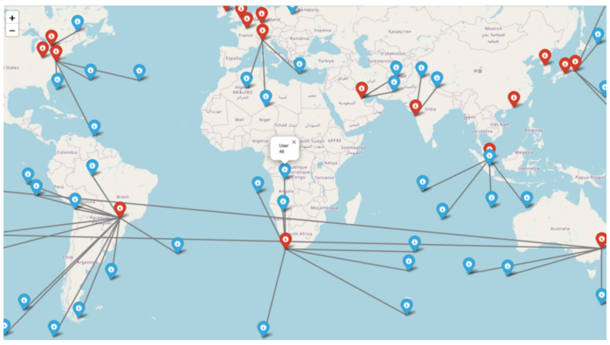
The algorithm ***gradually lowers*** the ***"temperature,"*** which controls the likelihood of accepting worse solutions as it runs, allowing it to explore different possibilities and converge towards the ***optimal solution*** over time. The best solution found during the process is returned along with its cost.

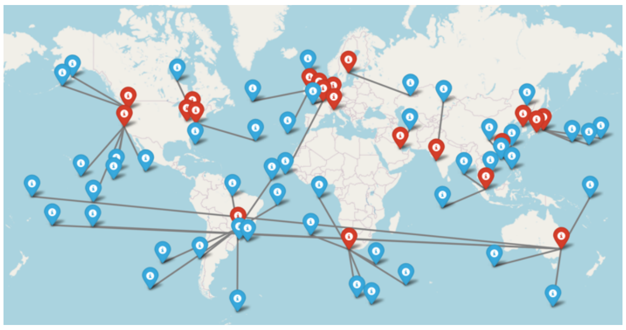
```
# Simulated Annealing Algorithm
def simulated_annealing(users, cloud_regions):
    current_solution = initialize_solution()
    current_cost = calculate_advanced_cost(cloud_regions, 
current_solution)
    best_solution = current_solution
    best_cost = current_cost

    temperature = INITIAL_TEMP

    while temperature > FINAL_TEMP:
        for _ in range(MAX_ITER):
            neighbor_solution = generate_neighbor(current_solution)
            neighbor_cost = calculate_advanced_cost(cloud_regions, 
neighbor_solution)

            # Metropolis Criterion
            delta = neighbor_cost - current_cost
            if neighbor_cost < current_cost or random.random() < 
math.exp(-delta / temperature):
                current_solution = neighbor_solution
                current_cost = neighbor_cost

            if current_cost < best_cost:
                best_solution = current_solution
                best_cost = current_cost

        temperature *= ALPHA

    return best_solution, best_cost
```

```
best_solution, best_cost = simulated_annealing(users, cloud_regions)
```

```
#Generate and show the map
region_names = df["Code"].tolist()  # Assuming the column "Code" contains 
region names

# Generate and show the map with region names
cloud_map = plot_solution_on_map(users, cloud_regions, best_solution, 
region_names)
cloud_map.save("cloud_allocation_map_with_names.html")
print("Map saved as 'cloud_allocation_map.html'. Open it in a browser.")
```







These outputs represent the results of applying ***SA*** to optimize the assignment of cloud users to Amazon cloud regions, visualized through interactive maps. The map highlights specific regions in red color points and users in blue color points. Arrows in the map represent the ***optimistic allocation of cloud users to specific cloud regions*** based on the algorithm.

PARTICLE SWARM OPTIMIZATION (PSO)

Swarm intelligence is a concept inspired by the collective behavior of social organisms, like birds, bees, and ants, working together to solve problems without centralized control. It relies on simple ***agents*** interacting locally with one another and their environment to achieve complex, global objectives. This ***decentralized approach*** enables adaptability, robustness, and efficient problem-solving in dynamic environments.

Algorithms using swarm intelligence concepts include:

- Ant Colony Optimization (ACO)
- Particle Swarm Optimization (PSO)
- Bee Algorithm
- Firefly Algorithm
- Bat Algorithm

The ***Particle Swarm Optimization (PSO)*** algorithm is a nature-inspired optimization technique based on the behavior of bird flocks or fish schools. Each "particle" represents a potential solution and moves through the solution space by adjusting its position based on its ***own best solution*** and ***best-found solution of group.*** PSO balances **exploration** and **exploitation,** and explore the space to ***avoid getting stuck in suboptimal solutions*** while exploiting known good areas to converge on the best solution.

PROBLEM STATEMENT

"Optimizing Crowd Simulations for Realistic Movie Scenes"

Creating realistic crowd simulations for large-scale scenes in movies, such as battle sequences or cityscapes, requires precise coordination of thousands of agents. Each agent must navigate dynamically through obstacles while maintaining realistic interactions, formations, and paths to meet the cinematic vision. The challenge lies in achieving behaviors such as avoiding collisions, preserving formations, and responding to dynamic goals(e.g., target points, obstacles, or enemy interactions).

Particle Swarm Optimization (PSO) is particularly suited for this application as it enables decentralized decision-making, where each agent can dynamically adapt to its environment based on both individual and group objectives.

PROPOSED SOLUTION

```
NUM_AGENTS = 50
ITERATIONS = 200
BOUNDARY = 100
GOAL = np.array([90, 90])  # Goal position
OBSTACLE_COUNT = 5
```

- ***W (Inertia Weight) :*** Determines how much an agent's current velocity influences its next movement, helping maintain momentum.
- ***C1 (Cognitive Component) :*** Controls how strongly an agent is attracted toward its personal best position, encouraging individual exploration.
- ***C2 (Social Component) :*** Governs how strongly an agent is attracted toward the global best position, promoting group coordination.

```
# PSO parameters
W = 0.5         # Inertia weight
C1 = 1.5        # Cognitive component
C2 = 1.5        # Social component
VELOCITY_LIMIT = 3
```

- ***COHESION_WEIGHT :*** Controls how strongly agents are attracted to the center of the group to maintain unity.
- ***SEPARATION_WEIGHT :*** Determines how much agents avoid crowding or colliding with nearby agents.
- ***ALIGNMENT_WEIGHT :*** Governs the tendency of agents to align their movement direction with the group.
- ***GOAL_ATTRACTION_WEIGHT :*** Specifies the strength of attraction toward the target position
- ***AVOID_OBSTACLE_WEIGHT :*** Defines the priority of avoiding obstacles in the agents' path.

```
# Agent behavior weights
COHESION_WEIGHT = 1.0
SEPARATION_WEIGHT = 2.0
ALIGNMENT_WEIGHT = 1.0
```

```
GOAL_ATTRACTION_WEIGHT = 2.5
AVOID_OBSTACLE_WEIGHT = 3.0
```

This section initializes the agents and their properties. Agents randomly generates the starting positions for all agents within the simulation boundary. velocities assigns each agent a random initial velocity, limited by the specified maximum velocity.

```
# Initialize agents
agents = np.random.rand(NUM_AGENTS, 2) * BOUNDARY  # Agent positions
velocities = np.random.rand(NUM_AGENTS, 2) * VELOCITY_LIMIT # velocities

personal_best_positions = np.copy(agents)
global_best_position = GOAL
```

```
# Generate obstacles
obstacles = np.random.rand(OBSTACLE_COUNT, 2) * BOUNDARY
OBSTACLE_RADIUS = 6
```

```
def cohesion(agents):
    center_of_mass = np.mean(agents, axis=0)
    return center_of_mass - agents
```

```
def separation(agents):
    # Coding parts in the Notebook
    return force
```

```
def alignment(agents, velocities):
    avg_velocity = np.mean(velocities, axis=0)
    return avg_velocity - velocities
```

```
def avoid_obstacles(agents, obstacles):
    # Coding parts in the Notebook
    return avoidance
```

```
def attraction_to_goal(agents, goal):
    return goal - agents
```

Calculates the new velocity for each agent by combining three components : ***inertia* , *cognitive*** and ***social.***

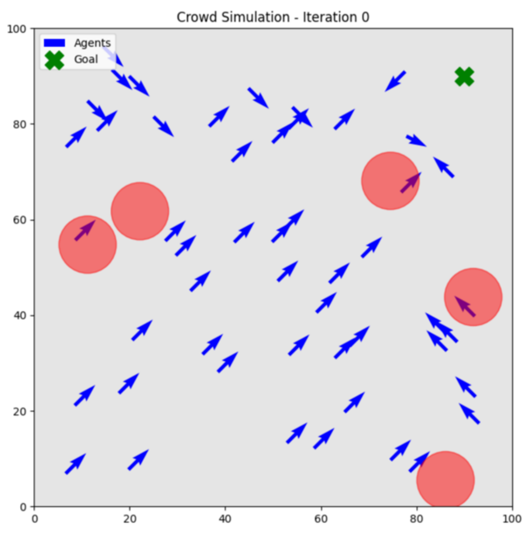
```
def update_velocity(vel, pos, personal_best, global_best):
    inertia = W * vel
    cognitive = C1 * np.random.rand(*vel.shape) * (personal_best - pos)
    social = C2 * np.random.rand(*vel.shape) * (global_best - pos)
    new_velocity = inertia + cognitive + social
    return np.clip(new_velocity, -VELOCITY_LIMIT, VELOCITY_LIMIT)
```

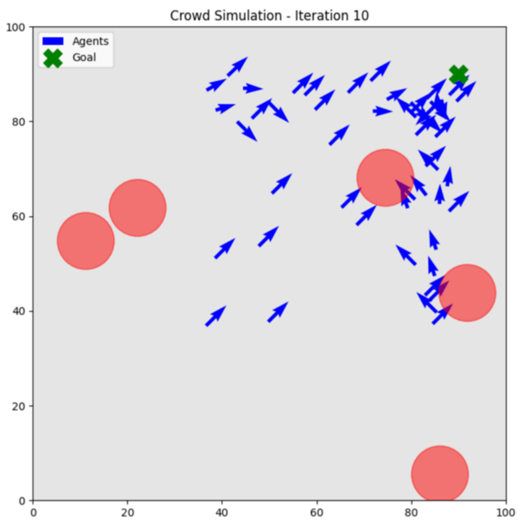
Updates each agent's position based on its new velocity, ensuring it remains within the simulation boundaries.

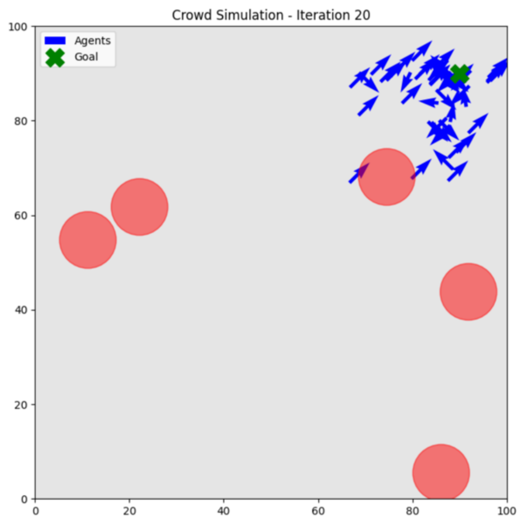
```
def update_positions(pos, vel):
    new_positions = pos + vel
    return np.clip(new_positions, 0, BOUNDARY)
```

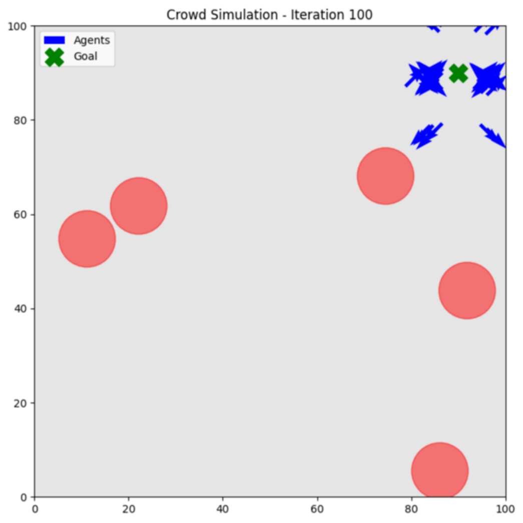
```
# Visualization function
def plot_simulation(agents, velocities, obstacles, goal, iteration):
    # Coding parts in the Notebook
    # Plot arrows for agents
    # Plot obstacles
    # Plot goal
```

```
# PSO loop
for iteration in range(ITERATIONS):
    cohesion_force = COHESION_WEIGHT * cohesion(agents)
    separation_force = SEPARATION_WEIGHT * separation(agents)
    alignment_force = ALIGNMENT_WEIGHT * alignment(agents, velocities)
    goal_force = GOAL_ATTRACTION_WEIGHT * attraction_to_goal(agents, GOAL)
    obstacle_force = AVOID_OBSTACLE_WEIGHT * avoid_obstacles(agents,
obstacles)

    velocities += cohesion_force + separation_force + alignment_force +
goal_force + obstacle_force
    velocities = np.clip(velocities, -VELOCITY_LIMIT, VELOCITY_LIMIT)

    agents = update_positions(agents, velocities)

    for i, agent in enumerate(agents):
        if np.linalg.norm(agent - GOAL) <
np.linalg.norm(personal_best_positions[i] - GOAL):
            personal_best_positions[i] = agent
    if iteration % 10 == 0:
        plot_simulation(agents, velocities, obstacles, GOAL, iteration)
```









The simulation output demonstrates the ***balance between exploration and exploitation*** in crowd behavior as agents navigate toward a goal while avoiding obstacles. In the initial iterations (Iteration 0), agents explore the space with random initial positions and velocities, spreading out across the area. As the iterations progress, agents begin to exploit the learned information by aligning their movement toward the goal. In the ***final iterations***, most agents successfully ***converge at the goal***. It Indicates successful simulation of crowd behavior.

ANT COLONY OPTIMIZATION (ACO)

Ant Colony Optimization (ACO) is a bio-inspired algorithm that simulates the ***foraging behavior*** of ants to solve optimization problems. In nature, ants find the shortest path to food by leaving pheromones on their routes; other ants are attracted to stronger pheromone trails, ***reinforcing*** efficient paths over time. ACO mimics this by using artificial ants that explore potential solutions and deposit virtual pheromones based on their quality, encouraging others to follow better paths.

PROBLEM STATEMENT

"Optimizing Exam Timetables with considering multiple constraints"

Efficiently scheduling exams for educational institutions requires balancing multiple constraints, such as ***avoiding scheduling conflicts***, ***utilizing limited resources*** (exam halls and examiners), and accommodating a growing number of exams and students. Each exam must be assigned a suitable timeslot, exam hall, and examiner while adhering to ***constraints*** such as avoiding overlaps and optimizing resource allocation.

Ant Colony Optimization (ACO) is particularly suited for this application as it leverages the collective intelligence of ants to explore and identify optimal solutions in complex search spaces. ACO dynamically adapts to scheduling constraints and ***optimizes timetables*** to minimize conflicts and resource inefficiencies, ***ensuring a fair and effective exam schedule.***

PROPOSED SOLUTION

These variables represent the input data for the optimization problem, where the goal is to assign exams to timeslots, exam halls, and examiners while ensuring efficiency and avoiding conflicts.

```
exams = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
timeslots = ['9:00 AM to 11:00 AM', '12:00 PM to 2:00 PM']
exam_halls = ['Exam Hall 1', 'Exam Hall 2', 'Exam Hall 3']
examiners = ['Examiner 1', 'Examiner 2', 'Examiner 3', 'Examiner 4',
'Examiner 5']
students = 100
```

Defines the parameters for the Ant Colony Optimization (ACO) algorithm. It sets the number of ants to 10, representing the ***agents exploring solutions.*** The parameter alpha controls the importance of pheromone trails, while beta emphasizes the influence of heuristic information. The ***evaporation_rate*** (0.5) specifies ***how quickly pheromone trails fade over time***, preventing ***premature convergence***. (control the Exploration and Exploitation)

```
# Parameters for ACO
num_ants = 10
num_iterations = 100
alpha = 1  # Pheromone importance
beta = 2   # Heuristic importance
evaporation_rate = 0.5
pheromone_constant = 100
```

```
epsilon = 1e-6  # Small constant to prevent division by zero
```

The ***pheromone matrix*** is filled with ones and represents the trail intensity for each combination of exam, timeslot, exam hall, and examiner, guiding the ants' search. The ***heuristic matrix*** is randomly generated and provides additional information to help ants make decisions based on problem-specific criteria.

```
# Initialize pheromone matrix and heuristic information
pheromone = np.ones((len(exams), len(timeslots), len(exam_halls),
len(examiners)))
heuristic = np.random.rand(len(exams), len(timeslots), len(exam_halls),
len(examiners))
```

This function evaluates the quality of a scheduling solution by assigning a cost. It iterates through the schedule, checking if any exam is assigned to the same combination of timeslot, exam hall, and examiner as another exam. ***If such a conflict is found, a penalty of 10 is added*** to the cost. ***Lower costs*** indicating better solutions that ***minimize scheduling overlaps.***

```
# Function to calculate solution cost
def calculate_cost(solution):
    cost = 0
    assigned = set()
    for exam, (timeslot, exam_hall, examiner) in solution.items():
        if (timeslot, exam_hall, examiner) in assigned:
            cost += 10  # Penalize conflicts
        assigned.add((timeslot, exam_hall, examiner))
    return cost
```

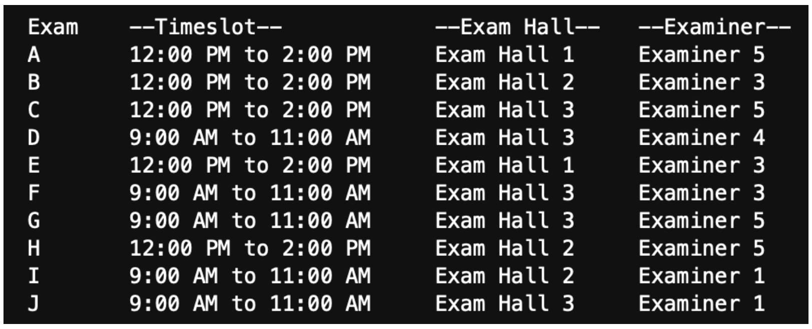
This function simulates the behavior of an ant colony to find the best exam timetable with minimal conflicts. It uses ***multiple "ants" to explore scheduling*** possibilities over several iterations. Each ant builds a solution by assigning exams to timeslots, exam halls, and examiners based on probabilities influenced by pheromone levels and heuristic values. After evaluating solutions using a cost function that penalizes conflicts, pheromone levels are updated to ***reward better schedules*** while allowing evaporation to prevent stagnation.

```
# Ant Colony Optimization algorithm
def ant_colony_optimization():
    global pheromone
    best_solution = None
    best_cost = float('inf')

    for iteration in range(num_iterations):
        solutions = []
        costs = []

        for ant in range(num_ants):
            solution = {}
            for i, exam in enumerate(exams):
                probabilities = []
                for t in range(len(timeslots)):
                    for r in range(len(exam_halls)):
                        for e in range(len(examiners)):
                            prob = (pheromone[i][t][r][e] ** alpha) *
(heuristic[i][t][r][e] ** beta)
                            probabilities.append((prob, t, r, e))
                probabilities = sorted(probabilities, key=lambda x: x[0],
reverse=True)
                chosen = random.choices(probabilities, weights=[p[0] for p
in probabilities], k=1)[0]
                solution[exam] = (timeslots[chosen[1]],
exam_halls[chosen[2]], examiners[chosen[3]])

            cost = calculate_cost(solution)
            solutions.append(solution)
            costs.append(cost)

        # Update pheromones
        pheromone *= (1 - evaporation_rate)
```

```python
        for solution, cost in zip(solutions, costs):
            for exam, (timeslot, exam_hall, examiner) in solution.items():
                i = exams.index(exam)
                t = timeslots.index(timeslot)
                r = exam_halls.index(exam_hall)
                e = examiners.index(examiner)
                pheromone[i][t][r][e] += pheromone_constant / (cost +
epsilon)

        # Track the best solution
        min_cost_idx = np.argmin(costs)
        if costs[min_cost_idx] < best_cost:
            best_cost = costs[min_cost_idx]
            best_solution = solutions[min_cost_idx]

    return best_solution
```

```python
# Run the algorithm
best_timetable = ant_colony_optimization()
```

```python
# Display the result
def display_timetable(timetable):
    print("Exam\t--Timeslot--\t\t--Exam Hall--\t--Examiner--")
    for exam, (timeslot, exam_hall, examiner) in timetable.items():
        print(f"{exam}\t{timeslot}\t{exam_hall}\t{examiner}")

display_timetable(best_timetable)
```



```
Exam    --Timeslot--            --Exam Hall--   --Examiner--
A       12:00 PM to 2:00 PM     Exam Hall 1     Examiner 5
B       12:00 PM to 2:00 PM     Exam Hall 2     Examiner 3
C       12:00 PM to 2:00 PM     Exam Hall 3     Examiner 5
D       9:00 AM to 11:00 AM     Exam Hall 3     Examiner 4
E       12:00 PM to 2:00 PM     Exam Hall 1     Examiner 3
F       9:00 AM to 11:00 AM     Exam Hall 3     Examiner 3
G       9:00 AM to 11:00 AM     Exam Hall 3     Examiner 5
H       12:00 PM to 2:00 PM     Exam Hall 2     Examiner 5
I       9:00 AM to 11:00 AM     Exam Hall 2     Examiner 1
J       9:00 AM to 11:00 AM     Exam Hall 3     Examiner 1
```

The output is a generated exam timetable created by the ACO. The schedule ensures that conflicts are minimized by distributing exams across available timeslots, exam halls, and examiners based on the optimization criteria**.** The algorithm's ability to probabilistically explore and refine solutions while ***balancing global optimization*** and ***local exploitation*** results in a well-structured and conflict-free timetable.

ACCESS THE CODE

Explore the implementation of the real world appications of algorithms using this link.

https://github.com/ShehanJay00/Nature-Inspired-Computing/tree/main/Reflective%20Journal

- END -