

**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 3](#_Toc185723306)

[GENETIC ALGORITHM (GA) 4](#_Toc185723307)

***[PROBLEM STATEMENT](#_Toc185723308)* [4](#_Toc185723308)**

[***PROPOSED SOLUTION* 5**](#_Toc185723309)

[SIMULATED ANNEALING (SA) 12](#_Toc185723310)

[***PROBLEM STATEMENT* 12**](#_Toc185723311)

[***PROPOSED SOLUTION* 12**](#_Toc185723312)

# **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.

# 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.

# 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.

# 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.***

# Load multiple chains from a FASTA file

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"):

x, y, z = 0.0, 0.0, 0.0 # Starting coordinates

chains = list(sequences.keys()) # Get chain IDs (e.g., A, B, C, D)

with open(filename, "w") as f:

f.write(f"MODEL\n")

atom\_index = 1

angle\_index = 0

for chain\_id, sequence in sequences.items():

for i, residue in enumerate(sequence):

if angle\_index >= len(angles): # Avoid index errors

break

# Calculate new coordinates based on angles

angle = angles[angle\_index]

x += math.cos(math.radians(angle))

y += math.sin(math.radians(angle))

z += angle / 100.0 # Spread in Z based on angle

# Write atom record for this residue

f.write(

f"ATOM {atom\_index:5d} CA {residue:3s} {chain\_id} {i+1:3d} {x:8.3f} {y:8.3f} {z:8.3f} 1.00 0.00 C\n"

)

atom\_index += 1

angle\_index += 1 # Move to the next angle

f.write("ENDMDL\n")

print(f"Structure saved to {filename}")

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

# Visualize structure with PyMOL

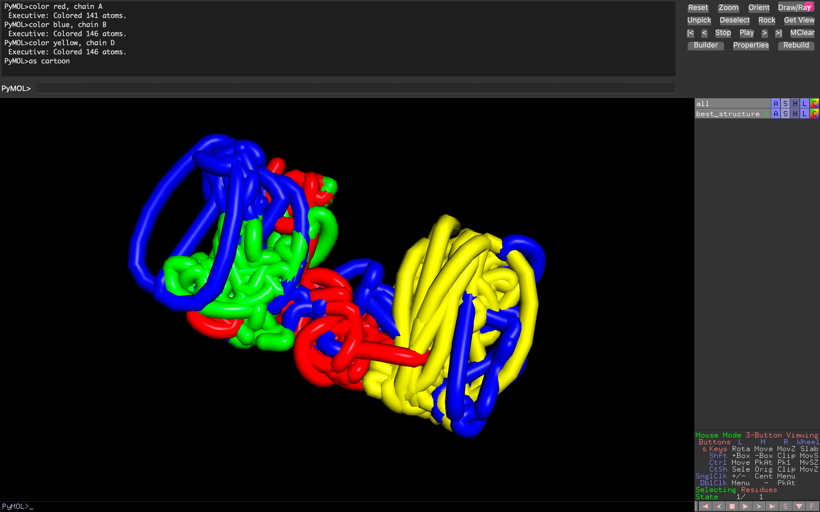
def visualize\_with\_pymol(filename="output.pdb"):

try:

subprocess.run(["pymol", filename])

except FileNotFoundError:

print("PyMOL is not installed or not found in your PATH.")

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.)

# Visualization of energy over generations

def plot\_progress(progress):

plt.figure(figsize=(10, 6))

plt.plot(progress, marker='o')

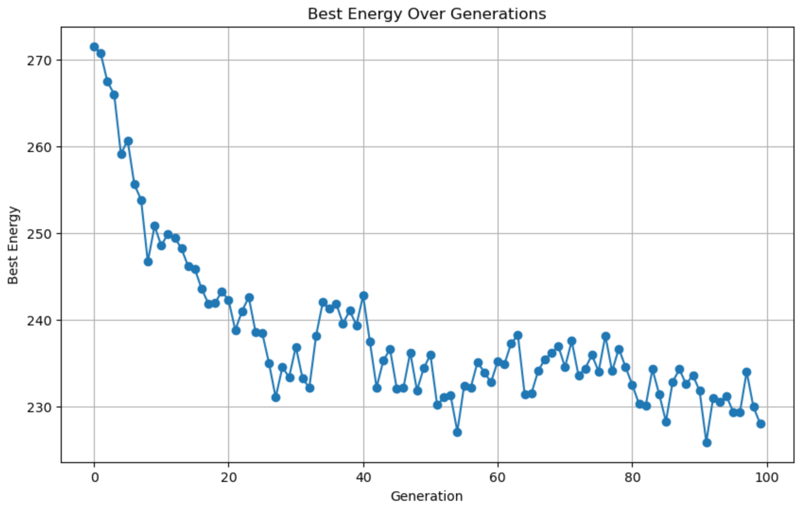
plt.title('Best Energy Over Generations')

plt.xlabel('Generation')

plt.ylabel('Best Energy')

plt.grid()

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.

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])

# 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

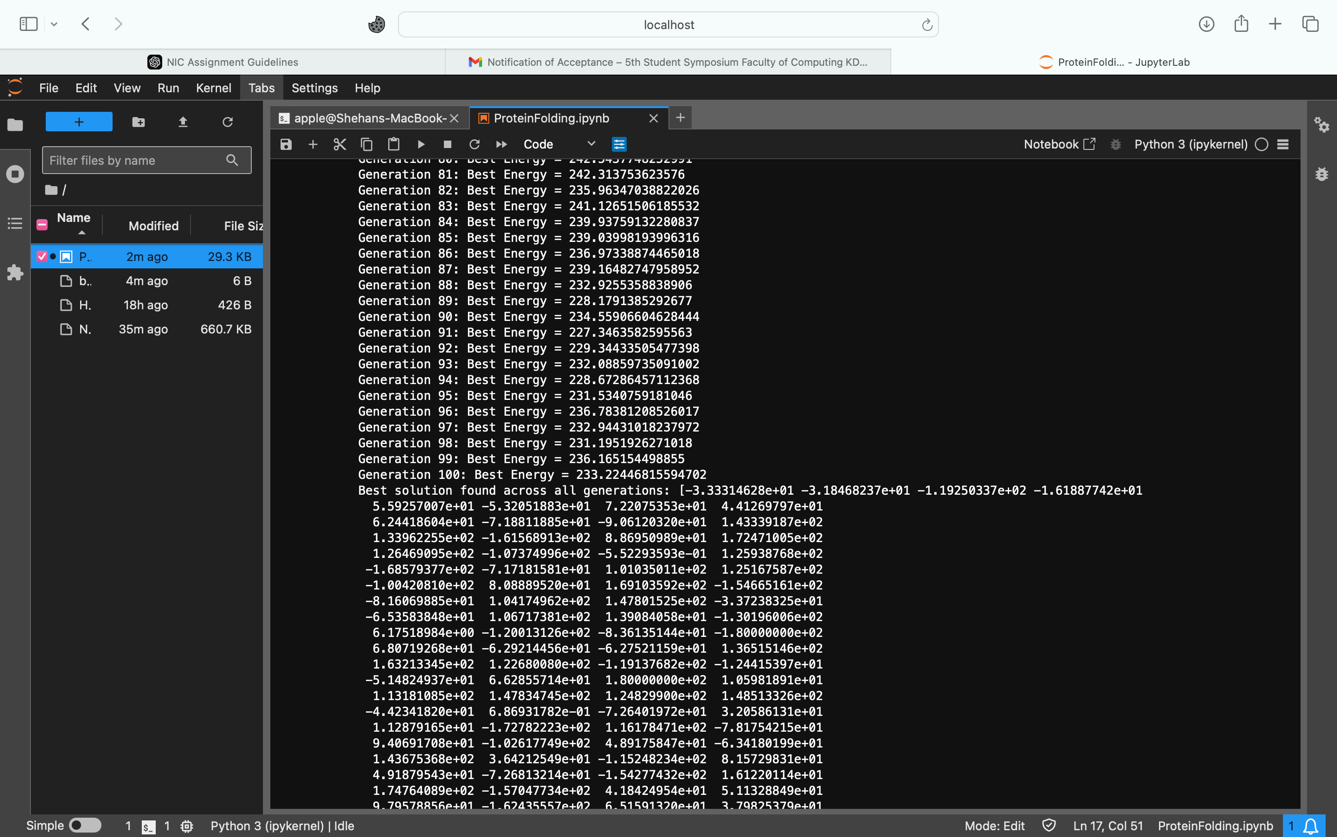
save\_structure\_to\_pdb(sequences, global\_best\_individual, filename="best\_structure.pdb")

# Visualize with PyMOL

visualize\_with\_pymol("best\_structure.pdb")

# Plot the progress

plot\_progress(progress)



A black background with white text

Description automatically generated

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)**

## ***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***