FREQUENCY BASED CODON

# Define the codon usage table for E. coli (actual data):

codon\_usage\_ecoli = {

'TTT': 0.0124, 'TTC': 0.0222, # Phenylalanine (Phe)

'TTA': 0.0041, 'TTG': 0.0031, # Leucine (Leu)

'CTT': 0.0108, 'CTC': 0.0153, 'CTA': 0.0069, 'CTG': 0.0154, # Leucine (Leu)

'ATT': 0.0244, 'ATC': 0.0296, 'ATA': 0.0056, # Isoleucine (Ile)

'ATG': 0.0186, # Methionine (Met)

'GTT': 0.0196, 'GTC': 0.0205, 'GTA': 0.0132, 'GTG': 0.0155, # Valine (Val)

'TCT': 0.0197, 'TCC': 0.0182, 'TCA': 0.0244, 'TCG': 0.0109, # Serine (Ser)

'CCT': 0.0142, 'CCC': 0.0152, 'CCA': 0.0146, 'CCG': 0.0134, # Proline (Pro)

'ACT': 0.0167, 'ACC': 0.0176, 'ACA': 0.0213, 'ACG': 0.0124, # Threonine (Thr)

'GCT': 0.0235, 'GCC': 0.0311, 'GCA': 0.0211, 'GCG': 0.0134, # Alanine (Ala)

'TAT': 0.0111, 'TAC': 0.0127, # Tyrosine (Tyr)

'TAA': 0.0004, 'TAG': 0.0004, 'TGA': 0.0007, # Stop codons

'CAT': 0.0144, 'CAC': 0.0183, # Histidine (His)

'CAA': 0.0196, 'CAG': 0.0197, # Glutamine (Gln)

'AAT': 0.0160, 'AAC': 0.0299, # Asparagine (Asn)

'AAA': 0.0177, 'AAG': 0.0203, # Lysine (Lys)

'GAT': 0.0140, 'GAC': 0.0225, # Aspartic Acid (Asp)

'GAA': 0.0217, 'GAG': 0.0299, # Glutamic Acid (Glu)

'TGT': 0.0075, 'TGC': 0.0145, # Cysteine (Cys)

'TGG': 0.0132, # Tryptophan (Trp)

'CGT': 0.0036, 'CGC': 0.0046, 'CGA': 0.0053, 'CGG': 0.0090, # Arginine (Arg)

'AGT': 0.0122, 'AGC': 0.0144, # Serine (Ser)

'AGA': 0.0047, 'AGG': 0.0062, # Arginine (Arg)

'GGT': 0.0127, 'GGC': 0.0221, 'GGA': 0.0150, 'GGG': 0.0074, # Glycine (Gly)

}

# Define the input sequence to be optimized:

input\_sequence = "CGTAGTTACCATGGAGCAGT"

# Function to perform frequency-based codon optimization:

def optimize\_codons(input\_sequence, codon\_usage):

optimized\_sequence = ""

for i in range(0, len(input\_sequence), 3):

codon = input\_sequence[i:i+3]

# Check if the codon exists in the codon usage table

if codon in codon\_usage:

optimized\_sequence += codon

else:

# Find the most frequent codon for the amino acid

amino\_acid = "X" # Placeholder for unknown codons

if i < len(input\_sequence) - 3:

next\_amino\_acid = input\_sequence[i+3:i+3+3]

if next\_amino\_acid in codon\_usage:

amino\_acid = next\_amino\_acid

# Find the most frequent codon for the amino acid

best\_codon = max(codon\_usage, key=lambda x: codon\_usage[x] if x.endswith(amino\_acid) else 0)

optimized\_sequence += best\_codon

return optimized\_sequence

# Perform codon optimization

optimized\_sequence = optimize\_codons(input\_sequence, codon\_usage\_ecoli)

# Print the original and optimized sequences

print("Original Sequence: ", input\_sequence)

print("Optimized Sequence: ", optimized\_sequence)

CODON OPTIMIZATION INDEX

from collections import Counter

from math import log

import random

def calculate\_cai(sequence, codon\_usage):

codon\_counts = Counter()

total\_codons = 0

# Calculate codon counts

for i in range(0, len(sequence), 3):

codon = sequence[i:i+3]

codon\_counts[codon] += 1

total\_codons += 1

cai = 1.0

# Calculate CAI value

for codon, count in codon\_counts.items():

if codon in codon\_usage:

cai \*= (codon\_usage[codon] \*\* count)

cai = cai \*\* (1 / total\_codons)

return cai

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# Replace this with your target DNA sequence

target\_sequence = "ATGGCACCTCCTGCATCCTGGTGGTCTGTGGGCGG"

# Replace this with the codon usage table for your host organism

codon\_usage = {

"TTT": 0.17, "TTC": 0.33, "TTA": 0.06, "TTG": 0.08

# Add more codons and their frequencies as needed

}

cai\_value = calculate\_cai(target\_sequence, codon\_usage)

print(f"CAI value: {cai\_value:.3f}")

def calculate\_cai(sequence, codon\_usage):

codon\_counts = {}

total\_codons = 0

for i in range(0, len(sequence), 3):

codon = sequence[i:i + 3]

codon\_counts[codon] = codon\_counts.get(codon, 0) + 1

total\_codons += 1

cai = 1.0

for codon, count in codon\_counts.items():

if codon in codon\_usage:

cai \*= (codon\_usage[codon] \*\* count)

cai = cai \*\* (1 / total\_codons)

return cai

def optimize\_codon\_sequence(target\_cai, current\_sequence, codon\_usage, max\_iterations=1000, mutation\_rate=0.1):

current\_cai = calculate\_cai(current\_sequence, codon\_usage)

best\_sequence = current\_sequence

best\_cai = current\_cai

for \_ in range(max\_iterations):

position\_to\_mutate = random.randint(0, len(current\_sequence) - 3)

new\_sequence = list(current\_sequence)

new\_codon = random.choice(list(codon\_usage.keys()))

new\_sequence[position\_to\_mutate:position\_to\_mutate+3] = list(new\_codon)

new\_sequence = ''.join(new\_sequence)

new\_cai = calculate\_cai(new\_sequence, codon\_usage)

if abs(new\_cai - target\_cai) < abs(best\_cai - target\_cai):

best\_sequence = new\_sequence

best\_cai = new\_cai

current\_sequence = best\_sequence

current\_cai = best\_cai

return best\_sequence

if \_\_name\_\_ == "\_\_main\_\_":

# Replace with your actual target CAI value and codon usage data

target\_cai = 0.8

codon\_usage = {

'TTT': 0.0124, 'TTC': 0.0222, # Phenylalanine (Phe)

'TTA': 0.0041, 'TTG': 0.0031, # Leucine (Leu)

'CTT': 0.0108, 'CTC': 0.0153, 'CTA': 0.0069, 'CTG': 0.0154, # Leucine (Leu)

'ATT': 0.0244, 'ATC': 0.0296, 'ATA': 0.0056, # Isoleucine (Ile)

'ATG': 0.0186, # Methionine (Met)

'GTT': 0.0196, 'GTC': 0.0205, 'GTA': 0.0132, 'GTG': 0.0155, # Valine (Val)

'TCT': 0.0197, 'TCC': 0.0182, 'TCA': 0.0244, 'TCG': 0.0109, # Serine (Ser)

'CCT': 0.0142, 'CCC': 0.0152, 'CCA': 0.0146, 'CCG': 0.0134, # Proline (Pro)

'ACT': 0.0167, 'ACC': 0.0176, 'ACA': 0.0213, 'ACG': 0.0124, # Threonine (Thr)

'GCT': 0.0235, 'GCC': 0.0311, 'GCA': 0.0211, 'GCG': 0.0134, # Alanine (Ala)

'TAT': 0.0111, 'TAC': 0.0127, # Tyrosine (Tyr)

'TAA': 0.0004, 'TAG': 0.0004, 'TGA': 0.0007, # Stop codons

'CAT': 0.0144, 'CAC': 0.0183, # Histidine (His)

'CAA': 0.0196, 'CAG': 0.0197, # Glutamine (Gln)

'AAT': 0.0160, 'AAC': 0.0299, # Asparagine (Asn)

'AAA': 0.0177, 'AAG': 0.0203, # Lysine (Lys)

'GAT': 0.0140, 'GAC': 0.0225, # Aspartic Acid (Asp)

'GAA': 0.0217, 'GAG': 0.0299, # Glutamic Acid (Glu)

'TGT': 0.0075, 'TGC': 0.0145, # Cysteine (Cys)

'TGG': 0.0132, # Tryptophan (Trp)

'CGT': 0.0036, 'CGC': 0.0046, 'CGA': 0.0053, 'CGG': 0.0090, # Arginine (Arg)

'AGT': 0.0122, 'AGC': 0.0144, # Serine (Ser)

'AGA': 0.0047, 'AGG': 0.0062, # Arginine (Arg)

'GGT': 0.0127, 'GGC': 0.0221, 'GGA': 0.0150, 'GGG': 0.0074, # Glycine (Gly)

}

# Replace this with your initial DNA sequence

current\_sequence = "CGTAGTTACCATGGAGCAGT"

optimized\_sequence = optimize\_codon\_sequence(target\_cai, current\_sequence, codon\_usage)

print(f"Optimized Sequence: {optimized\_sequence}")

print(f"Optimized CAI: {calculate\_cai(optimized\_sequence, codon\_usage):.3f}")

GENERIC ALGORITHM

import random

# Define the target protein sequence

target\_protein = "MSTHDTSLKTTE" # Replace with your actual protein sequence

# Define the codon usage table for the host organism (example)

host\_codon\_table = {

"M": ["ATG"],

"S": ["TCA", "TCT", "TCC", "TCG", "AGT", "AGC"],

"T": ["ACA", "ACT", "ACC", "ACG"],

"H": ["CAT", "CAC"],

"D": ["GAT", "GAC"],

"K": ["AAA", "AAG"],

"E": ["GAA", "GAG"],

"L": ["TTA", "TTG", "CTT", "CTC", "CTA", "CTG"],

}

# Define the genetic algorithm parameters

population\_size = 50

mutation\_rate = 0.01

generations = 100

# Initialize the population with random codon sequences

def generate\_random\_sequence(target\_protein):

return [random.choice(host\_codon\_table[aa]) for aa in target\_protein]

population = [generate\_random\_sequence(target\_protein) for \_ in range(population\_size)]

# Define the fitness function

def calculate\_fitness(sequence, target\_protein):

# Calculate a fitness score based on how well the sequence matches the target protein

return sum(codon == target\_codon for codon, target\_codon in zip(sequence, target\_protein))

# Genetic algorithm loop

for generation in range(generations):

# Calculate fitness scores for the entire population

fitness\_scores = [calculate\_fitness(seq, target\_protein) for seq in population]

# Select the top-performing sequences to be parents

parents = [population[i] for i in sorted(range(len(fitness\_scores)), key=lambda x: fitness\_scores[x], reverse=True)[:10]]

# Create the next generation

new\_population = parents[:]

while len(new\_population) < population\_size:

parent1 = random.choice(parents)

parent2 = random.choice(parents)

crossover\_point = random.randint(1, len(target\_protein) - 1)

child = parent1[:crossover\_point] + parent2[crossover\_point:]

# Apply mutation

if random.random() < mutation\_rate:

mutation\_position = random.randint(0, len(target\_protein) - 1)

new\_codon = random.choice(host\_codon\_table[target\_protein[mutation\_position]])

child = child[:mutation\_position] + [new\_codon] + child[mutation\_position+1:]

new\_population.append(child)

population = new\_population

# Select the best sequence from the final generation

best\_sequence = max(population, key=lambda seq: calculate\_fitness(seq, target\_protein))

print("Best codon-optimized sequence:", best\_sequence)

ITERATIVE REFINEMENT

import random

# Define the target protein sequence

target\_protein = "CGTAGTTACCATGGAGCAGT" # Replace with your actual protein sequence

# Define the codon usage table for the host organism, including A, C, and G nucleotides

host\_codon\_table = {

"M": ["ATG"],

"S": ["TCA", "TCT", "TCC", "TCG", "AGT", "AGC"],

"T": ["ACA", "ACT", "ACC", "ACG"],

"H": ["CAT", "CAC"],

"D": ["GAT", "GAC"],

"K": ["AAA", "AAG"],

"E": ["GAA", "GAG"],

"L": ["TTA", "TTG", "CTT", "CTC", "CTA", "CTG"],

"A": ["GCA", "GCC", "GCG", "GCT"],

"C": ["TGC", "TGT"],

"G": ["GGG", "GGC", "GGA", "GGT"],

}

# Define the genetic algorithm parameters

population\_size = 50

mutation\_rate = 0.01

generations = 10 # Number of refinement iterations

# Initialize the population with random codon sequences

def generate\_random\_sequence(target\_protein):

return [random.choice(host\_codon\_table[aa]) for aa in target\_protein]

population = [generate\_random\_sequence(target\_protein) for \_ in range(population\_size)]

# Define the fitness function

def calculate\_fitness(sequence, target\_protein):

# Calculate a fitness score based on how well the sequence matches the target protein

return sum(codon == target\_codon for codon, target\_codon in zip(sequence, target\_protein))

# Iterative refinement loop

for \_ in range(generations):

# Calculate fitness scores for the entire population

fitness\_scores = [calculate\_fitness(seq, target\_protein) for seq in population]

# Select the top-performing sequences to be parents

parents = [population[i] for i in sorted(range(len(fitness\_scores)), key=lambda x: fitness\_scores[x], reverse=True)[:10]]

# Create the next generation

new\_population = parents[:]

while len(new\_population) < population\_size:

parent1 = random.choice(parents)

parent2 = random.choice(parents)

crossover\_point = random.randint(1, len(target\_protein) - 1)

child = parent1[:crossover\_point] + parent2[crossover\_point:]

# Apply mutation

if random.random() < mutation\_rate:

mutation\_position = random.randint(0, len(target\_protein) - 1)

new\_codon = random.choice(host\_codon\_table[target\_protein[mutation\_position]])

child = child[:mutation\_position] + [new\_codon] + child[mutation\_position+1:]

new\_population.append(child)

population = new\_population

# Select the best sequence from the final generation

best\_sequence = max(population, key=lambda seq: calculate\_fitness(seq, target\_protein))

#print("Best codon-optimized sequence:", best\_sequence)

# Select the best sequence from the final generation

# Convert the list of codons to a single string

optimized\_sequence\_str = "".join(best\_sequence)

print("Best codon-optimized sequence:", optimized\_sequence\_str)

ONE - TO - ONE CODON

# Define the target protein sequence

target\_protein = "MSTHDTSLKTTE" # Replace with your actual protein sequence

# Define the codon usage table for the host organism (example)

host\_codon\_table = {

"M": ["ATG"],

"S": ["TCA", "TCT", "TCC", "TCG", "AGT", "AGC"],

"T": ["ACA", "ACT", "ACC", "ACG"],

"H": ["CAT", "CAC"],

"D": ["GAT", "GAC"],

"K": ["AAA", "AAG"],

"E": ["GAA", "GAG"],

"L": ["TTA", "TTG", "CTT", "CTC", "CTA", "CTG"],

}

# Function to perform one-to-one codon replacement

def one\_to\_one\_codon\_optimization(target\_protein, host\_codon\_table):

optimized\_sequence = []

for aa in target\_protein:

if aa in host\_codon\_table:

codon\_choices = host\_codon\_table[aa]

optimized\_sequence.append(codon\_choices[0]) # Select the most frequently used codon

return optimized\_sequence

# Perform one-to-one codon replacement

optimized\_sequence = one\_to\_one\_codon\_optimization(target\_protein, host\_codon\_table)

print("Codon-optimized sequence:", optimized\_sequence)