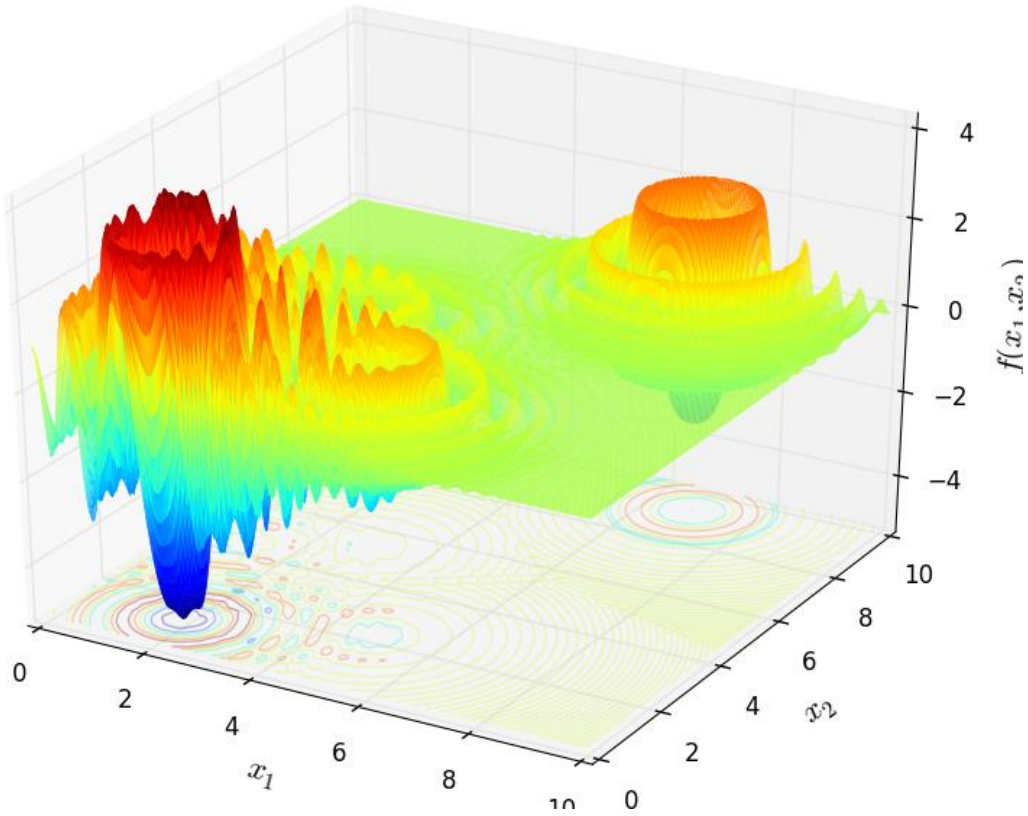
INSTITUTO POLITÉCNICO NACIONAL

ESCOM

9/13/2024

Algoritmo Genético

ALGORITMOS BIOINPIRADOS



Chen Yangfeng

6BV1 PROF. DANIEL MOLINA PÉREZ

Índice

Introducción

Este programa implementa un algoritmo evolutivo para minimizar la función **Langermann**, una función de prueba comúnmente utilizada en la optimización global. Los algoritmos evolutivos se inspiran en los procesos biológicos de selección natural y evolución, y buscan soluciones óptimas mediante la generación de múltiples soluciones candidatas, conocidas como población, y su mejora iterativa a través de operadores como la selección, cruzamiento (crossover) y mutación.

El propósito del programa es encontrar el mínimo global de la función Langermann en un espacio de búsqueda definido, utilizando una combinación de técnicas avanzadas como el cruzamiento SBX (Simulated Binary Crossover) y la mutación polinomial.

Desarrollo del programa

1. El programa está estructurado en varias etapas que representan el flujo de un algoritmo evolutivo básico:
2. Inicialización de la Población: Se genera una población inicial de individuos de manera aleatoria en un espacio de búsqueda definido por los límites [0, 10] en dos dimensiones. Cada individuo representa una posible solución al problema de optimización.
3. Evaluación de la Aptitud: Cada individuo se evalúa utilizando la función Langermann, que recibe como parámetros dos valores (correspondientes a las coordenadas de la solución) y calcula un valor de aptitud. El objetivo del algoritmo es minimizar esta función.
4. Selección de Padres: Se utiliza un esquema de selección de torneo determinista para elegir a los padres que pasarán al proceso de reproducción. En cada torneo, se seleccionan dos individuos y el que tenga mejor aptitud (valor de la función Langermann más bajo) es seleccionado para reproducirse.
5. Cruzamiento SBX: Los padres seleccionados se someten a un proceso de cruzamiento mediante la técnica SBX, que simula el comportamiento de la recombinación genética. Esta técnica crea nuevos individuos (hijos) que heredan características de los padres, introduciendo variabilidad en la población.
6. Mutación Polinomial: Los hijos generados se someten a una mutación polinomial, que altera sus genes (coordenadas) con cierta probabilidad. Esto introduce nuevas soluciones en el espacio de búsqueda y ayuda al algoritmo a explorar diferentes regiones.
7. Sustitución y Elitismo: Después de aplicar la mutación, se realiza la sustitución de la población anterior por la nueva. Se utiliza elitismo para asegurar que el mejor individuo de la población no se pierda durante el proceso de evolución.
8. Iteración. Este ciclo de selección, cruzamiento, mutación y evaluación se repite durante un número predefinido de generaciones. En cada generación, se monitorean el mejor individuo, el peor, la media de las soluciones y la desviación estándar, lo que permite evaluar el progreso del algoritmo.
9. Al final del proceso evolutivo, el algoritmo presenta la mejor solución encontrada, que se espera sea cercana al mínimo global de la función Langermann.

Codificación en Python

```
import numpy as np
import statistics

# Definir la función Langermann
def langermann(x, m=5, a=None, b=None, c=None):
    x1, x2 = x

    # Valores por defecto si no se proporcionan a, b y c
    if a is None:
        a = np.array([3, 5, 2, 1, 7])
    if b is None:
        b = np.array([5, 2, 1, 4, 9])
    if c is None:
        c = np.array([1, 2, 5, 2, 3])

    # Asegurarse de que las listas tienen longitud m
    a = np.array(a[:m])
    b = np.array(b[:m])
    c = np.array(c[:m])

    # Calcular el valor de la función Langermann
    result = 0
    for i in range(m):
        dist = (x1 - a[i]) ** 2 + (x2 - b[i]) ** 2
        result += c[i] * np.exp(-dist / np.pi) * np.cos(np.pi * dist)

    return -result


def create_initial_population(Np, n_var):
    return np.random.uniform(0, 10, (Np, n_var))


# Cruzamiento SBX (Simulated Binary Crossover)
def crossover_SBX(parents, lb, ub, Np, Nvar, Pc, Nc):
    Hijos = np.zeros((Np, Nvar))
    for i in range(0, Np-1, 2):
        if np.random.rand() <= Pc:
            u = np.random.rand()
            hijo1 = np.zeros(Nvar)
            hijo2 = np.zeros(Nvar)
            for j in range(Nvar):
                p1 = parents[i, j]
```

```
                p2 = parents[i+1, j]
                if p1 != p2:
                    beta = 1 + (2 / (p2 - p1)) * min(p1 - lb[j], ub[j] -
p2)
                    alpha = 2 - abs(beta) ** -(Nc + 1)
                    if u <= 1 / alpha:
                        beta_c = (u * alpha) ** (1 / (Nc + 1))
                    else:
                        beta_c = (1 / (2 - u * alpha)) ** (1 / (Nc + 1))
                    hijo1[j] = 0.5 * ((p1 + p2) - beta_c * abs(p2 - p1))
                    hijo2[j] = 0.5 * ((p1 + p2) + beta_c * abs(p2 - p1))
                else:
                    hijo1[j] = p1
                    hijo2[j] = p2
        else:
            hijo1 = parents[i, :]
            hijo2 = parents[i+1, :]
        Hijos[i, :] = hijo1
        Hijos[i+1, :] = hijo2
    return Hijos



def seleccion_torneo(poblacion, aptitud, Np, Nvar):

    padres = np.zeros((Np, Nvar))

    # Crear la matriz de torneos con dos columnas por cada selección
    torneo = np.column_stack((np.random.permutation(Np),
np.random.permutation(Np)))

    # Para cada competidor, determinar el ganador del torneo
    for i in range(Np):
        if aptitud[torneo[i, 0]] < aptitud[torneo[i, 1]]:
            # Pasa el competidor de la izquierda
            padres[i, :] = poblacion[torneo[i, 0], :]
        else:
            # Pasa el competidor de la derecha
            padres[i, :] = poblacion[torneo[i, 1], :]

    return padres


def mutation_polynomial(Hijos, lb, ub, Pm, Nm):
    Np, Nvar = Hijos.shape
```

```
    for i in range(Np):
        for j in range(Nvar):
            if np.random.rand() <= Pm:
                r = np.random.rand()
                delta = min((ub[j] - Hijos[i, j]), (Hijos[i, j] - lb[j]))
/ (ub[j] - lb[j])

                if r <= 0.5:
                    deltaq = -1 + (2*r + (1 - 2*r) * (1 - delta) ** (Nm +
1)) ** (1 / (Nm + 1))
                else:
                    deltaq = 1 - (2*(1-r) + 2*(r - 0.5)*(1 - delta) ** (Nm
+ 1)) ** (1 / (Nm + 1))

                # Mutar el individuo
                Hijos[i, j] = Hijos[i, j] + deltaq * (ub[j] - lb[j])

                # Asegurarse de que el valor mutado está dentro de los
límites
                Hijos[i, j] = np.clip(Hijos[i, j], lb[j], ub[j])

    return Hijos


def calculate_fitness(population, langermann_func, *langermann_params):
    return np.array([langermann_func(individual, *langermann_params) for
individual in population])


def elistismo(hijos, best_idx):
    random_idx = np.random.randint(0, len(hijos))
    hijos[random_idx] = best_idx
    return hijos


def main():

    Np = 200                # numero de poplacion
    num_generation = 200    # Número de generaciones
    n_var = 2               # Número de variables de decisión
    pc = 0.85               # Probabilidad de cruzamiento (Crossover)
    Pm = 0.03               # probabilidad de mutacion
    Nc = 2                  # Parámetro del SBX
    Nm = 100                 # numero de mutacion (indice de distribucion)
```

```
    lb = np.array([0, 0])
    ub = np.array([10, 10])

    population = create_initial_population(Np, n_var) # 1. Generación de
población inicial
    fitness = calculate_fitness(population, langermann) # 2. Evaluación de
población en la FO (cálculo de aptitud)

    for generation in range(num_generation):
        # 3 Selección del miembro de la población de mejor aptitud
        best_idx = np.argmin(fitness)
        best_individual = population[best_idx]

        parents = seleccion_torneo(population, fitness, Np, n_var) # 4.
Selección de padres (Torneo determinista de dos individuos)
        hijos = crossover_SBX(parents, lb, ub, Np, n_var, pc, Nc) # 5. SBX
        hijos = mutation_polynomial(hijos, lb, ub, Pm, Nm) # 6. Mutación
(Polinomial)

        population = elistismo(hijos, best_individual) # 7. Sustitución
(Extintiva con elitismo)

        fitness = calculate_fitness(population, langermann) # 8.
Evaluación de los descendientes en la FO (cálculo de aptitud)

        # Mejor resultado de la generación
        best_idx = np.argmin(fitness)
        best_solution = population[best_idx]

        # promedio de la generacion
        avr_solucion = sum(fitness) / len(fitness)

        # el peor de la generacion
        worst_idx = np.argmax(fitness)
        worst_solution = population[worst_idx]

        # desviacion estandar
        desv_est = statistics.stdev(fitness)

        print(f"Generación {generation}: Mejor solución = {best_solution}
-> Valor = {langermann(best_solution):.9f}. Peor solucion =
{worst_solution} -> Valor = {langermann(worst_solution):.9f}. Solucion
media = {avr_solucion:.9f}. Desv. estandar = {desv_est:.9f}")

    # mejor solucion
```

```
    best_idx = np.argmin(calculate_fitness(population, langermann))
    best_solution = population[best_idx]

    print("\nMejor solución encontrada:", best_solution)
    print("Valor de la función Langermann:", langermann(best_solution))


if __name__ == "__main__":
    main()
    # minimo global = -5.1621259
    # x = [2.00299219, 1.006096]
```

Tabla de resultados

Ejecución	Mejor	Media	Peor	Desv.estandar
1	-5.162126159	-5.153109654	-4.232589195	0.076963108
2	-5.162126160	-5.152231785	-4.498057844	0.061085938
3	-5.162126160	-5.156875134	-4.857750010	0.031608574
4	-5.162126160	-5.150539827	-4.390180152	0.071652526
5	-5.162126160	-5.152187618	-4.767777174	0.047536407
6	-5.162126160	-5.147336160	-4.664998672	0.063719294
7	-5.162126160	-5.150214820	-4.351922664	0.077682088
8	-5.162126160	-5.156313782	-4.857548451	0.036362269
9	-5.162126160	-5.156941488	-4.790740131	0.037234186
10	-5.162126160	-5.152047618	-4.774492635	0.050762952

Utilizando los siguientes parámetros:

```
Np = 200                # numero de poblacion
num_generation = 200    # Número de generaciones
n_var = 2               # Número de variables de decisión
pc = 0.85               # Probabilidad de cruzamiento (Crossover)
Pm = 0.03               # probabilidad de mutacion
Nc = 2                  # Parámetro del SBX
Nm = 100                # numero de mutacion (indice de distribucion)
lb = np.array([0, 0])
ub = np.array([10, 10])
```

Conclusión

El programa ilustra cómo los algoritmos evolutivos pueden ser aplicados para resolver problemas de optimización global no lineales, como la minimización de la función Langermann. La combinación de métodos de selección de torneo, cruzamiento SBX y mutación polinomial proporciona un enfoque robusto para explorar eficientemente el espacio de búsqueda.

A lo largo de múltiples generaciones, el algoritmo busca mejorar la calidad de las soluciones y aproximarse al mínimo global, presentando los mejores resultados encontrados en cada generación y asegurando que el mejor individuo no se pierda. Este enfoque permite encontrar soluciones óptimas para problemas complejos donde los métodos tradicionales de optimización pueden fallar debido a la presencia de múltiples óptimos locales.

El algoritmo tiene potencial de ser adaptado a otros problemas de optimización, siempre que se ajusten los parámetros y las funciones objetivo correspondientes

Referencias

N-D Test Functions L — AMPGO 0.1.0 documentation. (n.d.). https://infinity77.net/global_optimization/test_functions_nd_L.html