Josue Santana Robledo Corona 325073061

Maestría en Ciencias de la Robótica e Inteligencia Artificial

Algoritmos Bio-Inspirados

Centro Universitario de Ciencias Exactas e Ingeniería.

Mtro. Carlos Alberto López Franco

1. **Introducción**

Este reporte presenta la implementación del algoritmo de búsqueda local Hill Climbing, aplicado a funciones de prueba en n dimensiones.

2. **Descripción del código**

El código desarrollado implementa el algoritmo de Hill Climbing de manera generalizada para cualquier número de dimensiones (n).
Permite optimizar funciones matemáticas definidas por el usuario, ajustando parámetros como el número de iteraciones y el tamaño del paso de búsqueda.

3. **Componentes del Código**

Funciones de Prueba Implementadas

```
def funcion(x):
  return -np.sum(x**2) + 4*np.sum(x)
```

Aquí es donde ocurre la verdadera implementación del algoritmo

```
def hill_climbing(funcion, dimensiones=3, iteraciones=1000, paso=0.1, rango=(-5, 5)):
  actual = np.random.uniform(rango[0], rango[1], dimensiones)
  valor_actual = funcion(actual)

  for _ in range(iteraciones):
    candidato = actual + np.random.uniform(-paso, paso, dimensiones)
    valor_candidato = funcion(candidato)

    if valor_candidato < valor_actual:
      actual = candidato
      valor_actual = valor_candidato

  return actual, valor_actual
```

funcion: recibe la función objetivo a minimizar.

dimensiones: permite trabajar en espacios n-dimensionales.

paso: controla la magnitud del movimiento aleatorio.

iteraciones: define el número de evaluaciones.

rango: establece el dominio de búsqueda inicial.

Resultados Obtenidos

Primero vamos con ascendente y descendente en dimensión 2

Para hacerlo descendente y ascendente solo hay que cambiar estas líneas

Minimización

```
if valor_nuevo > valor_actual:
   x_actual = x_nuevo
   historia.append(x_actual.copy())

if valor_nuevo > mejor_valor:
   mejor_x = x_nuevo
   mejor_valor = valor_nuevo
```

Maximización

```
    if valor_candidato < valor_actual:
        actual = candidato
        valor_actual = valor_candidato

        if valor_actual < mejor_valor:
            mejor = actual.copy()
            mejor_valor = valor_actual
```

Minimización 2D

Punto inicial: x = [4.61160935 8.70960113], f(x) = -43.839

Resultado optimizado: x = [2.01462893 2.00595457], f(x) = 8.000

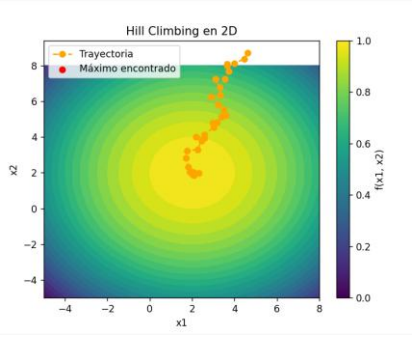

Imagen 1. Descendente en 2D

Maximización 2D

Punto inicial: x = [1.44791651 -7.7455844], f(x) = -87.281

Resultado optimizado: x = [-10.13054559 -133.26221384], f(x) = -18435.017

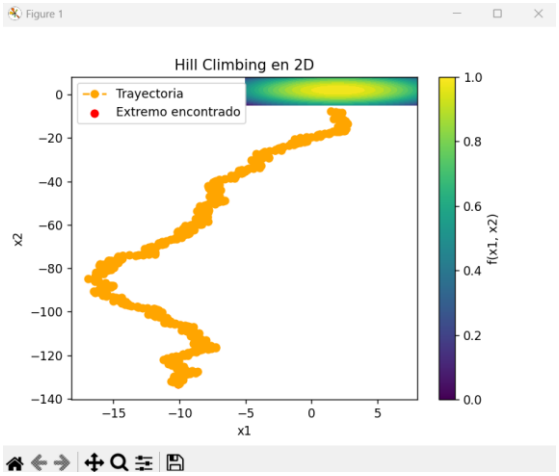

Imagen 2. Ascendente en 2D

Minimización 3D

Punto inicial: x = [2.30756557 -3.66000702 6.77559756], f(x) = -42.937

Resultado optimizado: x = [2.01211083 1.99319293 2.02404431], f(x) = 11.999

Maximización 3D

Punto inicial: x = [-8.33868035 -0.92879614 -9.80921342], f(x) = -242.924

Resultado optimizado: x = [-35.06820764 -20.87640368 -127.81598799], f(x) = -18737.573

4. Ventajas del Método

Simplicidad: fácil de entender e implementar.

Flexibilidad: aplicable a funciones en cualquier número de dimensiones.

5. Limitaciones

Óptimos locales: no garantiza encontrar el mínimo global si la función tiene múltiples mínimos.

6. Conclusiones

La implementación del algoritmo Hill Climbing en **n dimensiones** demuestra de forma clara el funcionamiento básico de las técnicas de búsqueda local.El código llegó en un momento en que se juntarón los algoritmo, y llevó más tiempo del que pensaba, causando un retraso inesperado, a pesar de ello se logró implementar perfectamente funcional, y se aprende un nuevo logaritmo al repertorio.

Código

```
import random
import numpy as np
import matplotlib.pyplot as plt


def funcion(x):
  return -np.sum(x**2) + 4*np.sum(x)

def hill_climbing_ND(funcion_obj, x_inicial, max_iter=1000, paso=0.5):
  x_actual = np.array(x_inicial)
  mejor_x = x_actual.copy()
  mejor_valor = funcion_obj(x_actual)
  historia = [x_actual.copy()]

  for i in range(max_iter):
    x_nuevo = x_actual + np.random.uniform(-paso, paso, size=x_actual.shape)
    valor_actual = funcion_obj(x_actual)
    valor_nuevo = funcion_obj(x_nuevo)

    if valor_nuevo > valor_actual:
      x_actual = x_nuevo
      historia.append(x_actual.copy())

    if valor_nuevo > mejor_valor:
      mejor_x = x_nuevo
      mejor_valor = valor_nuevo

  return mejor_x, mejor_valor, np.array(historia)


n_dim = 2
x_inicial = np.random.uniform(-10, 10, size=n_dim)
max_iter = 1000
paso = 0.5

x_optimo, valor_optimo, historia = hill_climbing_ND(funcion, x_inicial, max_iter, paso)
```

```
print(f"Punto inicial: x = {x_inicial}, f(x) = {funcion(x_inicial):.3f}")
print(f"Resultado optimizado: x = {x_optimo}, f(x) = {valor_optimo:.3f}")

if n_dim == 2:
    x1 = np.linspace(-5, 8, 200)
    x2 = np.linspace(-5, 8, 200)
    X1, X2 = np.meshgrid(x1, x2)
    Z = -X1**2 - X2**2 + 4*X1 + 4*X2

    plt.contourf(X1, X2, Z, levels=30, cmap='viridis')
    plt.plot(historia[:, 0], historia[:, 1], 'o--', color='orange', label='Trayectoria')
    plt.scatter(x_optimo[0], x_optimo[1], color='red', label='Máximo encontrado')
    plt.title("Hill Climbing en 2D")
    plt.xlabel("x1")
    plt.ylabel("x2")
    plt.legend()
    plt.colorbar(label='f(x1, x2)')
    plt.show()
```