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 con reemplazo, aplicado a funciones de prueba en n dimensiones.
A diferencia del Hill Climbing tradicional, este método evalúa un conjunto de vecinos en cada iteración y selecciona el mejor de ellos (First Improvement), lo que permite un recorrido más eficiente del espacio de búsqueda.

2. **Descripción del código**

El código implementa el algoritmo de Hill Climbing de manera general para cualquier número de dimensiones, con parámetros ajustables como:

- Número de iteraciones (max_iter)
- Tamaño del paso de búsqueda (paso)
- Número de vecinos evaluados en cada iteración (num_vecinos)
- Número de reinicios aleatorios (num_reinicios)

Esto permite comparar resultados locales y seleccionar el mejor punto global encontrado.

3. **Componentes del Código**

Función Objetivo

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

```
def hill_climbing_con_reemplazo(funcion_obj, x_inicial, max_iter=1000, paso=0.5, num_vecinos=20):
  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):
    vecinos = [x_actual + np.random.uniform(-paso, paso, size=len(x_actual)) for _ in range(num_vecinos)]
    valores = [funcion_obj(v) for v in vecinos]

    idx_mejor_vecino = np.argmax(valores)
```

```
        mejor_vecino = vecinos[idx_mejor_vecino]
        mejor_valor_vecino = valores[idx_mejor_vecino]

        if mejor_valor_vecino > mejor_valor:
            x_actual = mejor_vecino
            mejor_x = x_actual.copy()
            mejor_valor = mejor_valor_vecino
            historia.append(x_actual.copy())
        else:
            break

    return mejor_x, mejor_valor, np.array(historia)
```

funcion_obj: función objetivo a maximizar o minimizar.

x_inicial: punto de inicio aleatorio.

num_vecinos: número de vecinos evaluados en cada iteración (First Improvement).

max_iter: número máximo de iteraciones.

paso: tamaño del paso de búsqueda.

Se ejecuta el algoritmo varias veces con puntos iniciales distintos y se guarda el mejor resultado global.

for i in range(num_reinicios):

En esta línea

Resultados Obtenidos

Primero vamos con ascendente y descendente en dimensión 2

Al igual que el programa anterior. Para hacerlo descendente y ascendente solo hay que cambiar estas líneas

Minimización

idx_mejor_vecino=np.argmin(valores)

if mejor_valor_vecino < mejor_valor:

mejor_global_valor = np.inf

if valor_optimo < mejor_global_valor:

Maximización

idx_mejor_vecino=np.argmax(valores)

if mejor_valor_vecino > mejor_valor:

if valor_optimo > mejor_global_valor:

mejor_global_valor = -np.inf

Minimización 2D

Reinicio 1/5 → Mejor valor local: -266748.7740

Reinicio 2/5 → Mejor valor local: -269869.0114

Reinicio 3/5 → Mejor valor local: -262756.8316

Reinicio 4/5 → Mejor valor local: -276345.3108

Reinicio 5/5 → Mejor valor local: -266065.0577

Mejor punto global encontrado: [-367.92158216 375.5121603]

Mejor valor global: -276345.311

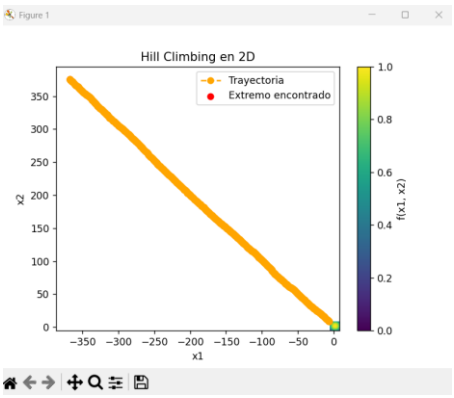

Imagen 1. Descendente en 2D

Maximización 2D

Reinicio 1/5 → Mejor valor local: 7.9919

Reinicio 2/5 → Mejor valor local: 7.9994

Reinicio 3/5 → Mejor valor local: 7.9911

Reinicio 4/5 → Mejor valor local: 7.9981

Reinicio 5/5 → Mejor valor local: 7.9999

Mejor punto global encontrado: [1.99090094 2.0034978]

Mejor valor global: 8.000

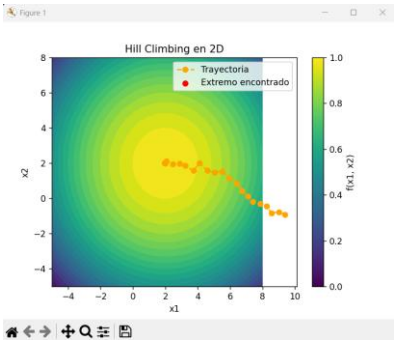

Imagen 2. Ascendente en 2D

Minimización 3D

Reinicio 1/5 → Mejor valor local: -272987.8190

Reinicio 2/5 → Mejor valor local: -253735.9611

Reinicio 3/5 → Mejor valor local: -262397.7273

Reinicio 4/5 → Mejor valor local: -274101.0929

Reinicio 5/5 → Mejor valor local: -271948.1571

Mejor punto global encontrado: [-369.63070782 370.78138495]

Mejor valor global: -274101.093

Maximización 3D

Reinicio 1/5 → Mejor valor local: 11.9728

Reinicio 2/5 → Mejor valor local: 11.9828

Reinicio 3/5 → Mejor valor local: 11.9853

Reinicio 4/5 → Mejor valor local: 11.9817

Reinicio 5/5 → Mejor valor local: 11.9794

Mejor punto global encontrado: [1.93604478 1.98886263 2.1025407]

Mejor valor global: 11.985

4. **Ventajas del Método**

Evaluación eficiente de vecinos (First Improvement).
Mayor probabilidad de escapar de óptimos locales gracias a los reinicios aleatorios.
Fácil de implementar y generalizable a n dimensiones.

5. **Limitaciones**

No garantiza alcanzar el óptimo global en funciones con múltiples máximos o mínimos.
Depende de la selección de parámetros (paso, num_vecinos, max_iter).

6. **Conclusiones**

El algoritmo de Hill Climbing con reemplazo representa una mejora sobre la versión determinista, al incorporar aleatoriedad en la exploración.
Permite explorar el espacio de búsqueda de manera más eficiente y al tener reinicios aleatorios aumenta la probabilidad de encontrar valores cercanos al óptimo global, siendo útil para funciones n-dimensionales, este código es la última variación del código de Hill climbing, y añade unas líneas más para los reinicios que no estaba en el código anterior.

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_con_reemplazo(funcion_obj, x_inicial, max_iter=1000,
paso=0.5,num_vecinos=20):
  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):
    vecinos = [x_actual + np.random.uniform(-paso, paso, size=len(x_actual)) for _ in
```

```
range(num_vecinos)]
      valores=[funcion_obj(v) for v in vecinos]

      idx_mejor_vecino=np.argmax(valores)
      mejor_vecino=vecinos[idx_mejor_vecino]
      mejor_valor_vecino=valores[idx_mejor_vecino]

      if mejor_valor_vecino > mejor_valor:
         x_actual = mejor_vecino
         mejor_x=x_actual.copy()
         mejor_valor = mejor_valor_vecino
         historia.append(x_actual.copy())
      else:
         break


   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
num_vecinos = 20
num_reinicios=5

mejor_global_x = None
mejor_global_valor = -np.inf
mejor_historia = None

for i in range(num_reinicios):
   x_inicial=np.random.uniform(-10,10,size=n_dim)
   x_optimo, valor_optimo, historia = hill_climbing_con_reemplazo(funcion, x_inicial, max_iter,
paso,num_vecinos)
   print(f"Reinicio {i + 1}/{num_reinicios} → Mejor valor local: {valor_optimo:.4f}")

   if valor_optimo > mejor_global_valor:
      mejor_global_valor = valor_optimo
      mejor_global_x = x_optimo
      mejor_historia = historia

print(f"Mejor punto global encontrado: {mejor_global_x}")
print(f"Mejor valor global: {mejor_global_valor:.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(mejor_historia[:, 0], mejor_historia[:, 1], 'o--', color='orange', label='Trayectoria')
    plt.scatter(mejor_global_x[0], mejor_global_x[1], color='red', label='Extremo encontrado')
    plt.title("Hill Climbing en 2D")
    plt.xlabel("x1")
    plt.ylabel("x2")
    plt.legend()
    plt.colorbar(label='f(x1, x2)')
    plt.show()
```