Instituto Politécnico Nacional
Escuela Superior de Cómputo
Centro de Investigación en Computación

Ingeniería en Inteligencia Artificial, Metaheuristicas
Fecha: 13 de abril de 2025

Ascensión de colinas

Presenta

Angeles López Erick Jesse[1]

Disponible en:
`https://github.com/JesseAngeles/HillClimbing`

Resumen

Se describe el comportamiento del algoritmo *Hill Climbing* para resolver problemas de optimización local. Se estudian tres versiones del algoritmo, ventajas y desventanas, pseudocodigo y propuestas para resolver problemas en computación: *Knapsack problem*, *Travel Salesman problem* y problemas de optmización del CEC 2017.

Palabras clave: Algoritmo, Hill Climibing, Pseudocódigo, Resultado óptimo

[1] eangelesl1700@alumno.ipn.mx

Índice

1. Hill Climbing

Es un algoritmo de búsqueda local que continuamente se mueve en la dirección que optimice el resultado. Dado un estado inicial, el algoritmo buscara aquel vecino que mejore su posición actual para moverse a el, cuando ninguno de los vecinos ofrece un mejor resultado, entonces se ha encontrado un óptimo local [1].

Este algoritmo sigue el mismo principio de subir una montaña, pues siempre se seguirá la ruta que tenga mayor altitud. Sin embargo, esto no nos garantiza que "escalemos" la montaña mas alta, sino la que tenemos mas cerca (dada nuestra condición inicial).

Debido a esto, se considera a *Hill Climbing* un algoritmo Greedy local, pues dada una condición inicial, unicamente es capaz de encontrar óptimos locales (Pueden ser máximos o mínimos dependiendo del tipo de problema) [1].

Este algoritmo debe de tener los siguientes elementos:

- **Estado inicial**: Solución principal. Puede ser fija o generada de manera aleatoria, la repetición de este algoritmo bajo condiciones aleatorias puede explorar multiples óptimos locales.
- **Vecindad**: Son el conjunto de estados a los cuales se pueden llegar a partir de un estado inicial.
- **Función objetivo**: Función que pondera cada estado para realizar un comparativa. Es necesario definir si es un problema de maximización o minimización, pues sera el criterio para definir si es un opción viable.

Ademas, se proponen tres variaciones del algoritmo *Hill Climbing*:

- **Steepest-Ascent Hill Climbing**: Ahora se analizan todos los vecinos y se mueve a aquel que tenga el mejor rendimiento según la función objetivo.
- **Next-Ascent Hill Climbin**: Analiza las funciones objetivos de todos los vecinos, y se mueve a aquel que tenga la mínima mejora.
- ***Random-Mutation Hill Climbing***: Se mueve a un estado aleatorio mutando un atributo siempre que exista una mejora en la función objetivo.

1.1. Ventajas

- **Optimo local**: Algoritmo con la capacidad de realizar búsquedas en amplitud para encontrar óptimos locales. Tiene variaciones que le permiten encontrar mínimos locales, aumentar el grado de exploración o incluso aumentar las posibilidades de encontrar óptimos globales.

- **Sencillez**: Algoritmo intuitivo de fácil implementación que no requiere estructuras de datos complejas ni almacenamiento constante. Pues la única información que almacena es la de el estado actual y de sus vecinos, no almacena información de la trayectoria realizada.

- **Búsqueda sin información completa**: Si no se conoce todo el espacio de búsqueda, puede ser una buena alternativa para definir y mapear el entorno. Esto puede reducir la complejidad al momento de querer explorar el espacio.

1.2. Desventajas

- **Óptimo global**: Partiendo de un único inicio, *Hill Climbing* no puede encontrar un óptimo local (La mayoría de las veces). Una alternativa para explorar óptimos globales es repetir el mismo algoritmo, desde diferentes condiciones iniciales para encontrar todos los máximos locales y escoger el mejor.

- **Cordilleras y corredores**: Supongamos un terreno donde la respuesta optima esta en dos direcciones (2 ejes), como Hill Climbing actualiza un único elemento del vector a la vez, tendrá que moverse en zig-zag para alcanzar el objetivo. Si los lados de la cordillera son muy pronunciados el algoritmo se ve forzado a realizar movimiento mas pequeños, lo que aumenta la cantidad de tiempo para escalar la cordillera.

- **Mesetas**: Si todas las opciones a las que puede moverse no mejoran ni empeoran entonces se esta en una meseta. El algoritmo no tiene una forma de determinar la próxima dirección que debe de tomar o si es la mejor solución.

1.3. Aplicaciones

- Reconocimiento de placas vehiculares. Permite centrar la atención de un software de visión artificial enfocarse en áreas de alto contraste como pueden ser semáforos, placas de vehículos y letreros [2].

- Partición de circuitos integrados mediante un algoritmo memetico basado en el algoritmo de escalada de colinas reduciendo el tiempo de convergencia, disminuye la comunicación entre sub-circuitos y disminuye el consumo energético [3].

- Nuevo método de optimización para encontrar el gradiente de una función [4].

- Clasificación y división de datos en conjuntos de entrenamientos y pruebas para grandes datasets. Se eliminan instancias redundantes, erróneas o ruidosas [5].

1.4. Resultados de Forrest y Mitchel

Al evaluar diferentes estrategias de ascensión de colinas, los autores demuestran que no existe un algoritmo "óptimo" para todos los casos, sino que cada variación del algoritmo se enfoca en un aspecto diferentes. Mientras que SAHC y NAHC no lograron encontrar el optimo en el tiempo estipulado, RMHC lo logra en un tiempo significativamente mas rápido al realizar menos evaluación de la función de aptitud.

Ciertamente existen algunas limitaciones. Las funciones *Royal Road* son intencionalmente simples que favorece ciertos comportamientos. Estas condiciones ideales pueden discernir los resultados de aplicaciones a problemas reales, con mas ruido, o con mayor complejidad. Ademas, estos algoritmos dependen unicamente de los valores de la función de aptitud, no considera otros aspectos que podrían ser esenciales para mejorar los resultados o la velocidad en que se obtienen.

Estas condiciones ideales permiten al algoritmo de ascensión de colinas tener mejores resultados que los algoritmos genéticos, pero puede que esta ventaja sea exclusiva por la naturaleza del problema que se desea resolver, evaluar diferentes escenarios permitiría explorar diferentes comportamientos ante condiciones volátiles [6].

2. Pseudocódigos

A continuación se describe el comportamiento del algoritmo *Hill Climbing* y sus variaciones. Los algoritmos 2, 3, 4 y 5 unicamente seleccionan el siguiente vecino. Mientras que el algoritmo 1 itera sobre un cierto numero de épocas y se "mueve" a la mejor posición. Se detiene cuando no hay mejora.

Algorithm 1 Hill Climbing

```
current_state = random state in states
for epoch in epochs do
    next_state = HillClimbing(current_state)
    if objective(current_state) == objective(next_state) then
        break
    end if
    current_state = next_state
end for
return current_state
```

2.1. Hill Climbing

Algorithm 2 Simple Hill Climbing
Input: *current_state*, *objective()*

```
for neighbour in current_state do
    if objective(neighbor) > objective(current_state) then
        current_state = neighbor
        break
    end if
end for
return current_state
```

2.2. Steepest-Ascent Hill Climbing (SAHC)

Algorithm 3 Steepest-Ascent Hill Climbing
Input: *current_state*, *objective()*

```
1: current_max = current_state
2: max = objective(current_state)
3: for all neighbour in current_state do
4:     if objective(neighbour) > max then
5:         max = objective(neigbour)
6:         current_max = neighbour
7:     end if
8: end for
9: return current_max
```

2.3. Next-Ascent Hill Climbing (NAHC)

Algorithm 4 Next-Ascent Hill Climbing
Input: *current_state*, *objective()*

```
1: current_min = current_state
2: min = ∞
3: for all neighbour in current_state do
4:     if objective(neighbour) > objective(current_state)
5:     and objective(neighbour) < min then
6:         min = objective(neigbour)
7:         current_min = neighbour
8:     end if
9: end for
10: return current_min
```

2.4. Random-Mutation Hill Climbing (RMHC)

Algorithm 5 Random-Mutation Hill Climbing
Input: *current_state*, *objective()*

```
1: new_state = random neighbour in current_state
2: if objective(new_state) > objective(current_state) then
3:     current_state = new_state
4: end if
5: return current_state
```

3. Problemas

3.1. Knapsack problem

Dado un conjunto de n ítems

$$I = \{1, 2, \ldots, n\}$$

Donde cada ítem i tiene un valor $v_i \geq 0$ y un peso $w_i \geq 0$ y dada una mochila con capacidad máxima W, se busca seleccionar un subconjunto de ítems que maximice el valor total sin exceder la capacidad.

Podemos representar los elementos dentro de la mochila como un vector binario:

$$x = (x_1, x_2, \ldots, x_n) \text{ con } x_i \in \{0, 1\}$$

Donde:

- $x_i = 0$ si el ítem no esta en la mochila
- $x_i = 1$ si el ítem si esta en la mochila

Para calcular el valor $v(x)$ y el peso $w(x)$ de la mochila sumamos los valores que si se encuentren dentro de ella:

$$v(x) = \sum_{i=1}^{n} v_i x_i$$
$$w(x) = \sum_{i=1}^{n} w_i x_i$$

El objetivo, es encontrar el mayor $v(x)$ siempre que el peso $w(x)$ no exceda el peso máximo W.

- El conjunto de estados posibles son todas las cadenas binarias de tamaño n:
$$S = \{x \in \{0, 1\}^n\}$$
- El estado inicial es una cadena de tamaño n sin ningún elemento en la mochila:
$$s_0 = \{x \in 0^n\}$$
Otra alternativa es escoger elementos aleatorios, pero existe la posibilidad de que dicha configuración inicial exceda el limite de peso.

- Se busca maximizar el valor de la mochila. La función objetivo suma los valores de los objetos dentro de la mochila. Si el peso de la mochila excede el limite, entonces se le asigna una ganancia negativa.

$$f(x) = \begin{cases} v(x), & \text{si } w(x) \leq W \\ W - w(x), & \text{si } w(x) > W \end{cases}$$

  Se le asigna la diferencia del peso máximo menos el peso actual (Dando un numero negativo). Esto con el objetivo de que, si por alguna razón esa es la mejor solución actual, sepa encontrar una mejor solución disminuyendo esa diferencia.

- Entonces, un estado x_j es un estado final si genera mayor aptitud en comparación de los demás x_i generados y tiene una aptitud no negativa:

$$f(x_j) \geq 0 \wedge f(x_j) \geq f(x_i) \; \forall x_i \in S$$

- La operación que genere los vecinos sera *Bit flip* que intercambia un 0 por un 1 y viceversa en la posición i.

$$B(x_i) = \begin{cases} 1, & \text{si } x_i = 0 \\ 0, & \text{si } x_i = 1 \end{cases}$$

3.2. Travel Salesman Problem (TSP)

Dado un conjunto de n ciudades

$$C = \{1, 2, \ldots, n\}$$

Y una matriz simétrica M que almacena las distancias entre las ciudades, se busca encontrar el camino hamiltoniano con menor distancia a recorrer. Es decir, se busca encontrar el recorrido de ciudades con la menor distancia pasando solo una vez por ciudad.

Podemos representar la trayectoria de las ciudades como un vector de enteros:

$$x = (x_1, x_2, \ldots, x_n) \text{ con } x_i \in [1, n]$$

Donde:

- $x_i = c$ es la ciudad c visitada en la i-ésima posición. Es necesario que cada c sea único en cada ruta x, es decir, que x sea una permutación de C.

Para calcular la distancia, iteramos el vector en orden y consultamos las distancias de cada par en la matriz M:

$$d(x) = \sum_{i=1}^{n} M(x_i, (x_{i+1}) \,\%\, n)$$

El objetivo, es encontrar la ruta x que minimice la distancia $d(x)$ siempre que la ruta no tenga ciudades c repetidas.

- El conjunto de estados posibles son todas las cadenas de enteros de tamaño n que sean una permutación de C:

  $$S = \{x \in [1, n]^n \mid x \text{ es una permutación de } C\}$$

- El estado inicial es una secuencia continua de todas las ciudades visitadas en orden:

  $$s_0 = (c_1, c_2, \ldots, c_n) = (c_i)_{i=1}^{n}$$

  Otra opción es generarlo de manera arbitraria. Se selecciona un numero en el rango al azar y se coloca en la ultima posición de la ruta. Este paso se repite para todos los números restantes (No se puede repetir un numero).

- Se busca minimizar la ruta. La función objetivo suma todas las distancias de la ruta planeada. Si una ciudad se visita mas de una vez, entonces se le asigna una ganancia nula. Dado que queremos minimizar la función, se le asigna infinito.

  $$f(x) = \begin{cases} d(x), & \text{si } \forall c \in C \colon \{c \in x\} \\ \infty, & \text{si } \exists c \in C \colon \{c \notin x\} \end{cases}$$

  Esto significa que:

  - Se le asigna $d(x)$ si todas las ciudades se encuentran en la ruta. Dado que la ruta es del mismo tamaño que el numero de ciudades, si aparecen todas las ciudades, entonces no hay ciudades repetidas.
  - Se le asigna ∞ si existe una ciudad que no aparezca en la ruta. Si una ciudad no aparece en la ruta, significa que al menos una ciudad aparece dos veces, por lo que se repite.

- Entonces, un estado x_j es un estado final si genera una menor aptitud en la comparación de los demás x_i generados:

  $$f(x_j) \leq f(x_i) \; \forall x_i \in S$$

- La operación que genere los vecinos sera *Swap*, ya que asegura unicamente cambiar el orden de los elementos sin tener que repetir ciudades. Esto implica que:

$$x_i = x_j \ \& \ x_j = x_i$$

Nótese que el estado inicial puede ser un estado de aceptación. Si realizamos puras operaciones *Swap*, no estamos añadiendo ni quitando ciudades, sino que unicamente se obtiene una nueva permutación. Por lo que podemos redefinir la función objetivo como:

$$f(x) = d(x)$$

Y el conjunto de estados posibles como cualquier vector de tamaño n que tenga números únicos en rango de $[1, n]$:

$$S = \{x \in \{1, 2, \ldots, n\}^n | \forall x_i \colon \forall x_j \colon x_i \neq x_j\}$$

3.3. Minimizar la función

Obtener los mínimos de la función

$$f(x) = \sum_{i=1}^{D} x_i^2, \ \text{ con } \ -10 \geq x_i \geq 10$$

.

Dado un vector de D números en el rango de $[-10, 10]$, se busca obtener el valor mínimo del sumatoria de sus cuadrados.

- El conjunto de estados posibles son todas las cadenas de enteros en dicho intervalo:

$$S = \{x \in [-10, 10]^n\}$$

- El estado inicial se genera de forma arbitraria como un vector de D números en el rango establecido $[-10, 10]$

- La función objetivo unicamente considera los valores dentro del propio vector:

$$f(x)$$

- Un estado de aceptación x_j es aquel que produzca el menor valor de aptitud en la función comparando con los demás x_i generados:

$$f(x_j) \leq f(x_i) \ \forall x_i \in S$$

- La operación que genere los vecinos puede tener multiples interpretaciones. Para este problema se asume un espacio circular donde -10 es el consecutivo del 10 y que $\forall d_i \in D, d_i \in \mathbb{Z}$. Entonces, los vecinos de d_i son los números consecutivos, es decir d_{i-1} y d_{i+1}.

  La operación sera entonces:

$$d_i = min(f(d_{i-1}), f(d_i), f(d_{i+1}))$$

3.4. Código

3.4.1. Hill Climbing

Se define una clase de *HillClimbing* que permite realizar las cuatro versiones del ascenso de colinas ya vistas en el código 1. Los parametros requeridos son los siguientes:

- *space*: Es es el espacio de soluciones de cada problema.
- *aditional_information*: Información adicional necesaria para calcular la función objetivo, puede de ser de cualquier tipo y es opcional según el problema.
- *objective_function*: Es la función objetivo que recibe como parámetros la información adicional y una configuración del espacio de soluciones.
- *neighbours_function*: Es la función que, dado una solución, obtiene todos los posibles vecinos realizando unicamente un movimiento. Esta función es especifica de cada problema.
- *max_random*: *Hill Climbing* se detiene cuando ya no encuentra una mejora, para evitar esto, se añade un numero máximo de iteraciones en las que se prueba con un vecino aleatorio antes de devolver una respuesta.

Listing 1: Constructor de la clase HillClimbing

```
1  class HillClimbing:
2    def __init__(self,
3      state,
4      aditional_information,
5      objective_function,
6      neighbours_function,
7      max_random = 30):
8    self.state = state
9    self.aditional_information = aditional_information
10   self.objective_function = objective_function
11   self.neighbours_function = neighbours_function
12   self.max_random = max_random
```

El código 2 se encarga de ejecutar el flujo de cualquier versión de *Hill Climbing*. Primero itera sobre las épocas y después llama a alguno de los cuatro métodos. Si mejora la función objetivo, entonces se mueve a dicha configuración. Si la función objetivo se mantiene igual significa que alcanzo un óptimo local, por lo que se detiene.

Listing 2: Función constructora HillClimbing

```
def HillClimbing(self, method, epochs:int, print: bool):
  current_state = self.state
  for _ in range(epochs):
    next_state = method(current_state)
    if (print):
      self.Print(next_state)
    if self.objective_function(self.aditional_information
       , next_state) == self.objective_function(self.
       aditional_information, current_state):
      break
  current_state = next_state

self.state = current_state
```

Los códigos 3, 4, 5 y 6 muestran la implementación de los pseudocodigos *Hill Climbing*, SAHC, NAHC y RMHC.

La función 6 tiene un ciclo adicional ya que busca de forma aleatoria dentro de los vecinos. Este limite se define en el constructor y permite explorar mas de una posibilidad, pues si no lo encuentra el algoritmo se termina. Es el numero de intentos requeridos antes de finalizar la búsqueda.

Listing 3: Función Hill Climbing

```
def SimpleHillClimbing(self, current_state):
  for i in range(len(current_state)):
    neighbours = self.neighbour_function(current_state,i)
    for neighbour in neighbours:
      if self.objective_function(self.
         aditional_information, neighbour) > self.
         objective_function(self.aditional_information,
         current_state):
        current_state = neighbour
        break
  return current_state
```

Listing 4: Función Steeptest Ascent Hill Climbing

```
def SteepestAscentHillClimbing(self, current_state):
  current_max = current_state
  max = self.objective_function(self.
     aditional_information, current_state)
  for i in range(len(current_state)):
    neighbours = self.neighbour_function(current_state,i)
    for neighbour in neighbours:
      if self.objective_function(self.
         aditional_information, neighbour) > max:
```

```
8           max = self.objective_function(self.
               aditional_information, neighbour)
9           current_max = neighbour
10    return current_max
```

Listing 5: Función Next Ascent Hill Climbing

```
1  def NextAscentHillClimbing(self, current_state)->list[any
       ]:
2    current_min = current_state
3    min = float('inf')
4     for i in range(len(current_state)):
5      neighbours = self.neighbour_function(current_state,i)
6      for neighbour in neighbours:
7        neighbour_objective_function = self.
             objective_function(self.aditional_information,
             neighbour)
8        if (neighbour_objective_function > self.
             objective_function(self.aditional_information,
             current_state) and neighbour_objective_function
             < min):
9          min = neighbour_objective_function
10         current_min = neighbour
11   return current_min
```

Listing 6: Función Random-Mutation Hill Climbing

```
1  def RandomMutationhillClimbing(self, current_state)->list
       [any]:
2    for _ in range(self.max_random):
3      random_pos = random.randrange(len(current_state))
4      new_state = self.neighbour_function(current_state,
           random_pos,True)
5
6      if self.objective_function(self.aditional_information
           , new_state) > self.objective_function(self.
           aditional_information, current_state):
7        current_state = new_state
8      break
9    return current_state
```

3.4.2. Objective Function

La función objetivo es una variable que depende del problema que se desea resolver. Esta función requiere de un estado a evaluar e información adicional

que sirve para realizar la evaluación como: distancias entre puntos, valores de cada elemento del estado, elementos anidados, transformaciones de los datos, etc.

El resultado que retorna es el valor objetivo de dicha configuración. Conforme a lo diseñado, la función *Hill Climbing* unicamente maximiza, por lo que si el objetivo es minimizar sera necesario invertir el signo de salida.

3.4.3. Neighbour Function

La función de vecindad recibe como parámetros el estado actual, el indice que se desea permutar y un valor booleano para obtener unicamente un elemento (Exclusivo para *Random Mutation Hill Climbing*).

Supongamos que el estado actual es un vector de tres dimensiones discreto $x = (a, b, c)$. Los vecinos serán todos aquellos que se pueda acceder mediante algún cambio en el vector (sea una unidad mas y una unidad menos): $\{(a_{-1}, b, c), (a_{+1}, b, c), \ldots, (a, b, c_{+1})\}$. Para evitar calcular todos los vecinos, el parámetro del indice indica el elemento del vector del cual se obtendrán los vecinos. Por otro lado, cabe la posibilidad de cada elemento tenga mas de un vecino (como lo es el a_{+1} y a_{-1}), por lo que el valor booleano se utiliza cuando unicamente se desea calcular uno de los vecinos.

3.4.4. Knapsack problem

La función objetivo se define en el código 7. Primero itera sobre todos los elementos que están dentro de la bolsa y suma los pesos como los valores correspondientes. Si el peso es menor al peso máximo entonces devuelve el valor total, en caso contrario devuelve la diferencia de los pesos.

Listing 7: Función objetivo de Knapsack problem

```
def objectiveFunction(aditional_information, state):
  max_weight = aditional_information[-1]
  weight: int = 0
  value: int = 0

  for i in range(len(state)):
    if state[i]:
      weight += aditional_information[i][0]
      value += aditional_information[i][1]

  if weight > max_weight:
    return max_weight - weight

  return value
```

Para obtener los vecinos de un un estado se utiliza la función de [**?**]. Esta función itera sobre cada elemento de la configuración y aplica Bit Flit.

Listing 8: Vecindad de Knapsack problem

```
def neighbourFunction(state:list[any], index:int,
   only_one = False)->list[list[any]]:
  neighbour = state[:]
  neighbour[index] = not neighbour[index]

  return neighbour if only_one else [neighbour]
```

La información adicional requerida son n pares de valores que almacenan los pesos y los valores respectivamente. El numero en la ultima posición representa el peso máximo de la mochila.

Listing 9: Parametros de knapsack problem

```
epochs:int = 100
state:list[bool] = [False, False, False]
aditional_information = [[3, 2], [3, 4], [5, 6], 10]
```

3.4.5. Travel Salesman

En el código 10 se definen la función objetivo. La función itera sobre la posible respuesta y busca las intersecciones en la matriz para obtener la distancia entre dos ciudades. Dado que queremos minimizar el problema, regresamos el numero negativo, entonces entre mayor sea (Mas cercano al cero), mejor resultado obtendrá.

Listing 10: Función objetivo de Travel Salesman problem

```
def objective_function(aditional_information, state):
  distance = 0
  num_cities:int = len(state)

  for i in range(num_cities):
    current_city = state[i]
    next_city = state[(i + 1) % num_cities]
    distance += aditional_information[current_city][
       next_city]

  return -distance
```

La función 11 obtiene la vecindad de un estado. Se consideran como vecindades todas aquellas configuraciones que se pueden alcanzar realizando unicamente un intercambio de elementos.

Listing 11: Vecindad de Travel Salesman Problem

```
def neighboursFunction(state:list[any], index:int,
   only_one = False)->list[list[any]]:
  if only_one:
    neighbour = state[:]
    pos = index
    while pos == index:
    pos = random.randrange(len(state))
    neighbour[pos], neighbour[index] = neighbour[index],
       neighbour[pos]
    return neighbour

  neighbours = []
  for i in range(len(state)):
    if index == i:
      continue

    neighbour = state[:]
    neighbour[i], neighbour[index] = neighbour[index],
       neighbour[i]
```

```
17
18      neighbours.append(neighbour)
19  return neighbours
```

La información adicional es la matriz de adyacencia.

Listing 12: Parametros de Travel Salesman problem

```
1  epochs:int = 100
2  state:list[int] = [0,1,2,3,4]
3  aditional_information = [[0,10,15], [10,0,35],[15,35,0]]
```

3.4.6. Minimizar la función

La función objetivo no utiliza la información adicional, pues son los propios elementos los que permite calcular el valor. En el código 13 itera sobre todos los y los eleva al cuadrado, dado que es un problema de minimización se invierte el signo.

Listing 13: Función objetivo de Función de suma

```
def objectiveFunction(aditional_information, state):
  total_sum:float = 0

  for val in state:
    total_sum += val**2

  return -total_sum
```

La función para obtener los vecinos 14 considera como vecinos todas aquellas configuraciones que pueden ser alcanzadas aumentando o disminuyendo en una unidad un parámetro (Siempre que es en el rango).

Listing 14: Vecinos de Función de suma

```
def neighboursFunction(state:list[any], index:int,
   only_one = False)->list[list[any]]:
  sign = random.choice({-1,1})
  neighbour1 = state[:]
  neighbour1[index] -= 1 * sign
  if neighbour1[index] < -10:
    neighbour1[index] = 10

  if only_one:
    return neighbour1 * sign

  neighbour2 = state[:]
  neighbour2[index] += 1
  if neighbour2[index] > 10:
    neighbour2[index] = -10

  return [neighbour1,neighbour2]
```

Este problema no requiere información adicional, así que unicamente es necesario definir la configuración inicial.

Listing 15: Parametros de Función suma

```
epochs:int = 100
state:list[int] = [0,1,2,3,4]
aditional_information:list = []
```

4. Problemas de optimización CEC 2017

En el documento [7] se presentan una serie de problemas sobre optimización numérica de parámetros reales. En este reporte se analizan las 10 primeras funciones que cumplen con la siguiente definición:

- Todas las funciones son problemas de minimización definidos de la siguiente manera:

  $$minf(x),\ x = [x_1, x_2, \ldots, x_D]^T$$

  Donde:

  - x es el vector de variables de dimensión D que representa la solución del problema.
  - D es el numero de dimensiones del problema.

- El óptimo global (la mejor solución) se encuentra desplazada del origen para evitar respuestas que asumen que la respuesta esta cerca del origen:

  $$o = [o_1, o_2, \ldots, o_D]^T$$

  Donde o es el vector del optimo global desplazado.

  El valor óptimo se distribuye de manera aleatoria en el rango de $o \in [-80, 80]^D$

- Las funciones son escalables, es decir, el numero de dimensiones D puede variar.

- El rango de búsqueda de todas las funciones para las variables se delimita por $x \in [-100, 100]^D$

- Implementación de matrices de rotación: Las variables interactúan entre ellas para volver el problema más difícil.

- Para simular problemas reales, las variables se dividen de manera aleatoria en subcomponentes. Cada subcomponente tiene su propia matriz de rotación.

4.1. Funciones

A continuación se definen las 10 primeras funciones.

1) Bent Cigar Function

$$f(x) = x_1^2 + 10^6 \sum_{i=2}^{D} x_i^2$$

2) Zakharov Function

$$f(x) = \sum_{i=1}^{D} x_i^2 + \left(0{,}5 \sum_{i=1}^{D} i x_i\right)^2 + \left(0{,}5 \sum_{i=1}^{D} i x_i\right)^4$$

3) Rosenbrock's Function

$$f(x) = \sum_{i=1}^{D-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$

4) Rastrigin's Function

$$f(x) = \sum_{i=1}^{D} \left[x_i^2 - 10\cos(2\pi x_i) + 10\right]$$

5) Expanded Schaffer's F6 Function

$$g(x, y) = 0{,}5 + \frac{\sin^2(\sqrt{x^2 + y^2}) - 0{,}5}{(1 + 0{,}001(x^2 + y^2))^2}$$

$$f(x) = \sum_{i=1}^{D-1} g(x_i, x_{i+1})$$

6) Lunacek Bi-Rastrigin Function

$$f(x) = \text{mín}\left(\sum_{i=1}^{D}(x_i - \mu_0)^2, dD + s\sum_{i=1}^{D}(x_i - \mu_1)^2\right) + 10\sum_{i=1}^{D}\left[1 - \cos(2\pi z_i)\right]$$

$$\mu_0 = 2{,}5, \quad \mu_1 = -\sqrt{\frac{\mu_0^2}{d}}$$

7) Non-Continuous Rotated Rastrigin's Function

$$f(x) = \sum_{i=1}^{D} \left[z_i^2 - 10\cos(2\pi z_i) + 10\right]$$

$$z_i = \text{Tosz}(\text{Tasy}(x_i))$$

8) Levy Function

$$f(x) = \sin^2(\pi w_1) + \sum_{i=1}^{D-1} (w_i - 1)^2 \left[1 + 10\sin^2(\pi w_i + 1)\right] + (w_D - 1)^2 \left[1 + \sin^2(2\pi w_D)\right]$$

$$w_i = 1 + \frac{x_i - 1}{4}$$

9) Modified Schwefel's Function

$$f(x) = 418{,}9829D - \sum_{i=1}^{D} x_i \sin(\sqrt{|x_i|})$$

10) High Conditioned Elliptic Function

$$f(x) = \sum_{i=1}^{D} 10^{6\frac{i-1}{D-1}} x_i^2$$

Cuyas graficas se observan en la figura 1

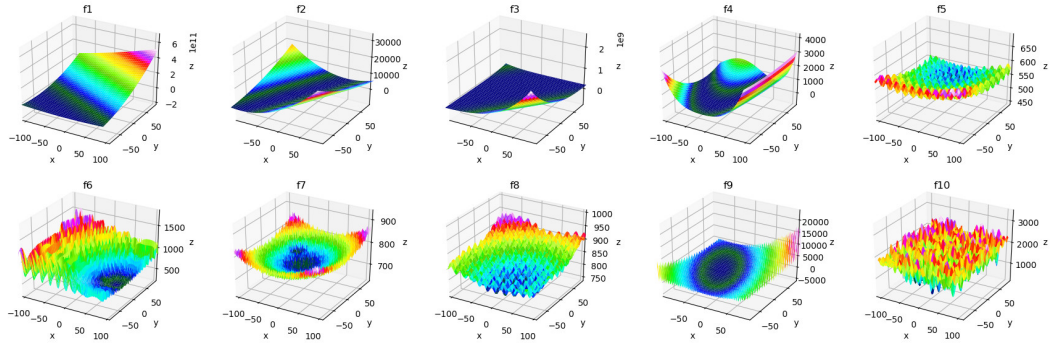

Figura 1: Superficies ploteadas de las 10 primeras funciones para dos dimensiones [8]

4.2. Código

Para los problemas del CEC 2017, se utilizaron las funciones de Duncan Tilley en Python en el repositorio *cec2017-py* [9].

Para el algoritmo de *Random Mutation Hill Climbing* se usa el mismo código que para los problemas anteriores (Código 2 y 6), por lo que unicamente es necesario diseñar la función objetivo y la función para obtener la vecindad.

La función objetivo es dada por el propio problema de optimización la cual recibe como parámetros el vector del estado actual (código 16).

Listing 16: Función objetivo

```
def objectiveFunction(aditional_information, state):
  return -aditional_information([state])[0]
```

Por otro lado, el código 17 muestra la función para obtener a los vecinos. Dado que las funciones existen en un espacio de valores reales, se opta por avanzar una cantidad aleatoria en cualquier dirección.

Listing 17: Función de vecindad

```
def neighboursFunction(state, index: int, only_one =
    False):
  noise = np.random.normal()
  neighbour1 = state[:]

  neighbour1[index] -= noise

  if only_one:
    return neighbour1

  neighbour2 = state[:]
  neighbour2[index] += noise

  return [neighbour1, neighbour2]
```

Para realizar diferentes pruebas, se utilizan los códigos 18 y 19.

La función *run_hill_climbing* recibe como parámetros la función objetivo, el numero de dimensiones, las épocas de entrenamiento y el numero de iteraciones que se desea repetir el proceso. El calculo se almacena en una lista que almacena el valor inicial, el optimo alcanzado y el tiempo de ejecución.

Listing 18: Función run_hill_climbing

```
def run_hill_climbing(function, dimension, epochs,
    iterations):
  results = []

  for i in range(iterations):
    state = np.random.uniform(low=-100, high=100, size=
        dimension).tolist()
    hc = HillClimbing(np.array(state), function,
        objectiveFunction, neighboursFunction)
    start_time = time.time()

```

```
9       # Valores iniciales
10      initial_objective = objectiveFunction(function, hc.
            state)
11      hc.HillClimbing(hc.RandomMutationhillClimbing, epochs
            , is_print=0)
12
13      # Tiempo de ejecucion
14      execution_time = time.time() - start_time
15
16      # Valores finales
17      final_objective = objectiveFunction(function, hc.
            state)
18
19      # Almacenar el resultado
20      result = {
21        "function": function.__name__,
22        "iteration": i + 1,
23        "initial_objective": initial_objective,
24        "final_objective": final_objective,
25        "execution_time": execution_time
26      }
27
28      results.append(result)
29
30    return results
```

Por otro lado, la función *parallel_hill_climbing* se encarga de obtener los mínimos de forma paralela. Como nuevos parámetros se define el nombre del archivo de salida y el numero máximo de procesos en paralelo.

Listing 19: Función parallel_hill_climbing

```
1   def parallel_hill_climbing(dimension, epochs, iterations,
        num_workers, output_file):
2     # Funciones a optimizar
3     all_functions = {f1,f2,f3,f4,f5,f6,f7,f8,f9,f10}
4
5     #  Empaquetar argumentos adicionales
6     run_partial = partial(run_hill_climbing, dimension=
          dimension, epochs=epochs, iterations=iterations)
7
8     # Ejecutar en paralelo
9     all_results = []
10
11    with ProcessPoolExecutor(max_workers=num_workers) as
          executor:
```

```
12      futures = [executor.submit(run_partial, f) for f in
            all_functions]
13
14      for future in futures:
15          all_results.extend(future.result())
16
17    # Exportar resultados a JSON
18    with open(output_file, "w") as f:
19      json.dump(all_results, f, indent=4)
```

Por lo únicamente es necesario llamar a la función especificando los parámetros, de la misma forma que se muestra en el código 20 para obtener un archivo tipo *JSON* con los resultados.

Listing 20: Llamada de la función parallel_hill_climbing

```
parallel_hill_climbing(
  dimension=10,
  epochs=1000000,
  iterations=10,
  num_workers=28,
  output_file="hill_climbing_results.json",
)
```

4.3. Resultados

Considerando vectores de tamaño 100, un millón de épocas, 20 iteraciones sobre cada función y 28 núcleos de procesador para trabajar de forma paralela, se obtuvieron los resultado de la tabla 1 y la tabla 2. Es necesario mencionar que, pese al número de épocas, la búsqueda se detiene cuando no encuentra una mejor opción.

Adicionalmente, la clase *Hill Climbing* siempre busca maximizar el resultado, por lo que para búsqueda de mínimos locales, se tiene que cambiar el signo en la función objetivo, por lo que los resultados obtenidos en la tabla 1 tienen signo negativo.

La tabla 2 por su parte, muestra las estadísticas obtenidas pero del tiempo de ejecución en segundos.

f(x)	**Peor**	**Mejor**	**Promedio**	**Mediana**	**Desviación estándar**
f1	-1.4336e+12	-2.1804e+11	-6.3137e+11	-6.0096e+11	0.4964
f2	-8.82e+236	-2.27e+165	-4.41e+235	-4.42e+212	48.0865
f3	-1.0953e+6	-7.1786e+5	-9.3153e+5	-9.3192e+5	0.0963
f4	-1620.66	-588.09	-799.38	-731.21	0.2302
f5	-3327.35	-1840.57	-2757.48	-2785.81	0.1245
f6	-742.87	-713.81	-728.13	-729.15	0.0101
f7	-18239.97	-13219.15	-15362.38	-15198.37	0.0883
f8	-3694.17	-2613.38	-3257.37	-3306.11	0.0894
f9	-122972.20	-64916.02	-90212.80	-89889.81	0.1456
f10	-22817.03	-16102.13	-19675.01	-19759.43	0.0890

Tabla 1: Estadísticas de resultados de las funciones.

f(x)	**Peor**	**Mejor**	**Promedio**	**Mediana**	**Desviación estándar**
f1	1.3949	2.5994	2.1067	2.0907	0.3010
f2	0.0037	1.0159	0.1740	0.0304	0.2920
f3	0.0379	0.5611	0.2319	0.1944	0.1462
f4	3.2157	5.2511	4.1232	4.0357	0.5391
f5	0.2728	0.6614	0.3403	0.3147	0.0875
f6	0.1920	0.4877	0.3026	0.2955	0.0739
f7	0.1070	0.2585	0.1833	0.1962	0.0406
f8	0.0974	1.2510	0.3685	0.3258	0.2238
f9	0.0611	0.5272	0.1840	0.1092	0.1420
f10	0.2390	1.4073	0.4699	0.3321	0.3459

Tabla 2: Estadísticas de tiempo de ejecución por función.

Referencias

[1] T. b Kute, "Mitu skillologies – artificial intelligence, data science training and development – open source technocrats: Artificial intelligence, data science, machine learning, training in pune, maharashtra, india," 2025, accedido el 11 de marzo de 2025. [En línea]. Disponible: https://mitu.co.in/wp-content/uploads/2022/04/8.-Hill-Climbing-Algorithm.pdf.

[2] Y. C. Yu, S. C. D. You, and D. R. Tsai, "Hill climbing algorithm for license plate recognition," *Adv. Mater. Res.*, vol. 267, pp. 995–1000, jun 2011, accedido el 20 de marzo de 2025. [En línea]. Disponible: https://doi.org/10.4028/www.scientific.net/amr.267.995.

[3] K. J. Prakash and P. Sivakumar, "Memetic algorithm based on hill climbing algorithm for ic partitioning," *3C Tecnol. Innovacion Apl. Pyme*, pp. 181–193, mar 2020, accedido el 20 de marzo de 2025. [En línea]. Disponible: https://doi.org/10.17993/3ctecno.2020.specialissue4.181-193.

[4] S. M. Goldfeld, R. E. Quandt, and H. F. Trotter, "Maximization by quadratic hill-climbing," *Econometrica*, vol. 34, no. 3, p. 541, jul 1966, accedido el 20 de marzo de 2025. [En línea]. Disponible: https://doi.org/10.2307/1909768.

[5] L. Si, J. Yu, W. Wu, J. Ma, Q. Wu, and S. Li, "Rmhc-mr: Instance selection by random mutation hill climbing algorithm with mapreduce in big data," *Procedia Comput. Sci.*, vol. 111, pp. 252–259, 2017, accedido el 20 de marzo de 2025. [En línea]. Disponible: https://doi.org/10.1016/j.procs.2017.06.061.

[6] S. Forrest and M. Mitchell, "Relative building-block fitness and the building-block hypothesis," 1995, accedido el 11 de marzo de 2025. [En línea]. Disponible: https://citeseerx.ist.psu.edu/document?repid=rep1\&type=pdf\&doi=e9a14839d7fc56c17082fa14436f46ccec1c6c83.

[7] Y. S. Ong, P. N. Suganthan, and T. Weise, "Definitions of CEC2017 Benchmark Suite," 2017, accedido el 26 de marzo de 2025. [En línea]. Disponible: ruta/local/o/enlace.

[8] N. H. Awad, M. Z. Ali, P. N. Suganthan, J. J. Liang, and B. Y. Qu, "Problem Definitions and Evaluation Criteria for the CEC 2017 Special Session and Competition on Single Objective Bound Constrained Real-Parameter Numerical Optimization," Tech. Rep., 2016, accedido el 26 de marzo de 2025.

[9] D. Tilley, "GitHub - tilleyd/cec2017-py: Python module for CEC 2017 single objective optimization test function suite," 2025, accedido el 26 de marzo de 2025. [En línea]. Disponible: https://github.com/tilleyd/cec2017-py.