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

Ingeniería en Inteligencia Artificial, Metaheuristicas
Fecha: 13-03-24

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 dos problemas famosos en computación: *Knapsack problem* y *Travel Salesman problem*.

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

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.

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:

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

4. Código

El código a continuación describe la clase *HillClimbing* y las funciones para los cuatro métodos ya vistos: *Hill Climbing* 12, *Steepest-Ascent Hill Climbing* 13, *Next-Ascent Hill Climbing* 14 y *Random-Mutation Hill Climbing* 2.

La clase requiere de los siguientes parámetros:

- *space*: Vector unidimensional que representa la solución.
- *information*: Información adicional opcional para evaluar la solución.
- *evaluateFunction*: Función de evaluación
- *valueFunction*: Define el comportamiento de la función de evaluación.
- *minimize*: Valor booleano que define si se desea maximizar o minimizar

Listing 1: Constructor de la clase HillClimbing

```
def __init__(self,
space: list[Any],
information: list[Any],
evaluateFunction: Callable,
valueFunction: Callable,
minimize: bool,
):

  self.space = space
  self.information = information
  self.evaluateFunction = evaluateFunction
  self.valueFunction = valueFunction
  self.minimize = minimize
```

La función de *Random-Mutation Hill Climbing* (2), selecciona una posición aleatoria dentro del espacio para realizar la mutación. Después llama a la función *valueFunction* y valida si hubo una mejora en la evaluación. Si lo fue, entonces se actualiza el espacio, en caso contrario continua a la siguiente época.

Listing 2: Función Random-Mutation Hill Climbing

```
def RMHC(self, epochs: int):
  value: float = self.evaluate()

  for _ in range(epochs):
    i = random.randrange(0, len(self.space))

    new_space = self.space[:]
```

```
8       new_value: float = self.value(new_space, i, value, 3)
9
10      if (self.minimize and new_value < value) or
11      (not self.minimize and new_value > value):
12        value = new_value
13        self.space = new_space[:]
```

4.1. Knapsack problem

En el código 3 se definen los parametros los cuales son:

- *epochs*: Numero de iteraciones
- *space*: Vector de valores booleanos que define si el objeto se mete en la mochila
- *information*: Arreglo de elementos con su peso y valor respectivo. El ultimo valor es el peso máximo de la mochila.

Listing 3: Parametros de Knapsack problem

```
epochs: int = 2
space: list[bool] = [False, False, False]
information: list = [[3, 2], [3, 4], [5, 6], 10]
```

La función de evaluación 4 extrae el peso máximo de la mochila, y almacena la suma del peso y del valor de todos los atributos que, en la configuración a probar, se encuentran en la mochila.

Listing 4: Evaluación de Knapsack problem

```
def evaluateKnapsack(space: list[bool], information:list):
    max_weight = information[-1]
    weight: int = 0
    value: int = 0

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

      if weight > max_weight:
        return -1

return value
```

Como la operación unicamente requiere intercambiar un elemento en la posición i del vector. Entonces la función 5 unicamente intercambia el valor y valida si mejora la respuesta.

Listing 5: Validación de Knapsack problem

```
def valueKnapsack(space,information,position,value,option):
    space[position] = not space[position]
    new_value: float = evaluateKnapsack(space, information)

```

```
5       if new_value > value:
6           return new_value
7       return value
```

4.2. Travel Salesman

En el código 6 se definen los parametros los cuales son:

- *epochs*: Numero de iteraciones
- *space*: Vector de valores enteros que define la ruta de viaje. Este vector es una permutación de las ciudades.
- *information*: Matriz de adyacencia sobre las distancias entre las ciudades.

Listing 6: Parametros de Knapsack problem

```
epochs: int = 10
space: list[int] = [0, 1, 2, 3, 4]
information:list[list[int]] = [
    [0, 10, 15, 20, 25],
    [10, 0, 35, 30, 40],
    [15, 35, 0, 45, 50],
    [20, 30, 45, 0, 55],
    [25, 40, 50, 55, 0]
]
```

La función de evaluación 7 calcula la distancia recorrida por el agente viajero con base en la secuencia del espacio actual.

Listing 7: Evaluación de Travel Salesman Problem

```
def evaluateTravelSalesman(space,information):
    distance:int = 0
    num_cities:int = len(space)

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

    return distance
```

El código a continuación muestra unicamente el comportamiento para la función de $Random - Mutation$, pero es fácilmente adaptable a los diferentes casos según la variable *option*.

A diferencia de *Knapsack problem*, aquí existen multiples estados. Es por esto que se selecciona un indice de manera arbitraria para hacer el intercambio de valores. Si el intercambio proporciona una ruta mas corta, entonces se actualiza el espacio la el valor de dicha ruta, tal como se muestra en **??**.

Listing 8: Validación de Travel Salesman problem

```
def valueTravelSalesman(space,info,pos,value,option):
if option == 3:
  new_space = space[:]
  i = random.randrange(0, len(space))
  new_space[pos],new_space[i]=new_space[i],new_space[pos]
  new_value=evaluateTravelSalesman(new_space,info)

  if new_value < value:
    space[:] = new_space[:]
    return new_value
return value
```

4.3. Minimizar la función

En el código 9 se definen los parametros los cuales son:

- *epochs*: Numero de iteraciones
- *space*: Vector de valores booleanos que define si el objeto se mete en la mochila
- *information*: Arreglo vacio. No se definen valores ya que el calculo de la función de validación es mediante los elementos que se encuentren en el espacio.

Listing 9: Parametros de Función de suma

```
epochs: int = 2
space: list[int] = [0, 1, 2, 3, 4]
information:list = []
```

La función de evaluación 10 suma el cuadrado de los elementos en el espacio.

Listing 10: Evaluación de Función de suma

```
def evaluateSumFunction(space, information):
    total_sum:float = 0

    for cell in space:
        total_sum += cell**2

    return total_sum
```

Se considera intercambiar un elemento de forma aleatoria por un valor aleatorio dentro del rango de $[-10, 10]$. Se valida si el resultado mejor el resultado (disminuye), si es asi, entonces actualiza el espacio.

Listing 11: Validación de Función suma

```
def valueSumfunction(space,information,
position,value,option):

  if option == 3:
    new_space = space[:]
    val=random.randrange(-10, 10)
    new_space[position] = val
    new_value=evaluateSumFunction(new_space,information)

    if new_value < value:
      space[:] = new_space[:]
    return new_value
```

```
13
14      return value
```

5. Anexos

Listing 12: Función Hill Climbing

```
def HC(self, epochs: int):
  value: float = self.evaluate()

  for _ in range(epochs):
    for i in range(len(self.space)):
      new_space = self.space[:]
      new_value: float = self.value(new_space, i, value)

      if (self.minimize and new_value < value) or
      (not self.minimize and new_value > value):
        value = new_value
        self.space = new_space[:]
```

Listing 13: Función Steepest-Ascent Hill Climbing

```
def SAHC(self, epochs: int):
  value: float = self.evaluate()

  for _ in range(epochs):
    best_options:list = []

    for i in range(len(self.space)):
      new_space = self.space[:]
      new_value: float = self.value(new_space, i, value,1)

      if (self.minimize and new_value < value) or
      (not self.minimize and new_value > value):
        best_options.append([new_value, new_space])

    if best_options:
      if self.minimize:
        _, best_option = min(enumerate(best_options),
        key=lambda x: x[1][0])
      else:
        _, best_option = max(enumerate(best_options),
        key=lambda x: x[1][0])

      if (self.minimize and best_option[0] < value) or
      (not self.minimize and best_option[0] > value):
        value = best_option[0]
        self.space = best_option[1]
```

Listing 14: Función Next-Ascent Hill Climbing

```
def NAHC(self, epochs: int):
  value: float = self.evaluate()

  for _ in range(epochs):
    best_options:list = []

    for i in range(len(self.space)):
      new_space = self.space[:]
      new_value: float = self.value(new_space, i, value,1)

      if (self.minimize and new_value < value) or
      (not self.minimize and new_value > value):
      best_options.append([new_value, new_space])

    if best_options:
      if self.minimize:
        _, best_option = min(enumerate(best_options),
        key=lambda x: x[1][0])
      else:
        _, best_option = max(enumerate(best_options),
        key=lambda x: x[1][0])

    if (self.minimize and best_option[0] < value) or
    (not self.minimize and best_option[0] > value):
      value = best_option[0]
      self.space = best_option[1]
```

Referencias

[1] T. b Kute. MITU Skillologies – Aritificial Intelligence, Data Science Training and Development – Open Source Technocrats : Artificial Intelligence, Data Science, Machine Learning, Training in Pune, Maharashtra, India. Accedido el 11 de marzo de 2025. [En línea]. Disponible:https://mitu.co.in/wp-content/uploads/2022/04/8.-Hill-Climbing-Algorithm.pdf