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

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

Recocido simulado

Presenta

Angeles López Erick Jesse[1]

Disponible en:
`github.com/JesseAngeles/Metaheuristicas`

Resumen

Se describe el comportamiento del algoritmo *Simulated Annealing* para resolver problemas de optimización local. Se estudia el comportamiento del algoritmo, ventajas y desventanas, pseudocodigo, comparativa con *RMHC* y propuestas para resolver problemas en computación como *Knapsack problem* y *Travel Salesman problem*.

Palabras clave: Algoritmo, Pseudocódigo, Resultado óptimo, Simulated Annealing

[1] eangelesl1700@alumno.ipn.mx

Índice

1. Simulated Annealing **3**
1.1. Recocido físico . . . 3
1.2. Recocido simulado . . . 3
1.3. Ventajas . . . 4
1.4. Desventajas . . . 4
1.5. Aplicaciones . . . 4
1.6. SA vs RMHC . . . 5

2. Pseudocódigo **6**

3. Problemas **7**
3.1. Knapsack problem . . . 7
3.2. Travel Salesman Problem (TSP) . . . 8
3.3. Minimizar la función . . . 10
3.4. Código . . . 11
3.4.1. Simulated Annealing . . . 11
3.4.2. Knapsack problem . . . 13
3.4.3. Travel Salesman Problem . . . 14
3.4.4. Minimizar la función . . . 16

4. Conclusión **19**

Referencias **20**

1. Simulated Annealing

1.1. Recocido físico

Trabajar con los metales en sus estados puros puede resultar contraproducente. Se busca obtener las propiedades deseadas mediante diferentes tratamientos, entre ellos, el recocido (*Annealing*).

Este proceso térmico busca cambiar las propiedades físicas y mecánicas del material con el objetivo de volverlo mas trabajable [1].

Este proceso calienta un metal (como el acero) a altas temperaturas para enfriarlo lentamente con el objetivo de eliminar defectos estructurales, como tensiones internas, que pudieron haber aparecido en procesos de deformación o de enfriamiento rápido. Este enfriamiento permite que los átomos se reorganicen en una configuración mas estable y de menor energía [2].

1.2. Recocido simulado

El recocido simulado (*Simulated Annealing*) se inspira del recocido físico pero aplicado a un contexto matemático. Se busca resolver problemas de optimización mediante un proceso de exploración del espacio de soluciones.

Se compone de cinco elementos:

- **Solución inicial**: Se comienza con una solución inicial aleatoria o previamente determinada por el problema
- **Energía**: Se define una función objetivo (de energía) que mide la calidad de la solución. Esta función es única para cada problema de optimización (máximos y mínimos).
- **Enfriamiento**: Se sigue un proceso análogo al enfriamiento controlado. En cada iteración se genera una nueva solución vecina a partir de la solución actual.

  - Si la nueva solución es **mejor** se acepta como la solución actual.
  - Si la nueva solución es **peor**, la decisión de aceptarla depende de una probabilidad (la probabilidad disminuye gradualmente conforme avanza el proceso).

  A temperaturas altas se permiten cambios grandes de enfriamiento, pero a medida que el sistema se "enfría" la probabilidad de que una solución peor sea aceptada disminuye.
- **Temperatura**: Probabilidad de aceptar peores soluciones. Inicia alta y decrece en cada iteración (decrecimiento lineal, exponencial, factorial, etc).

- **Convergencia**: El algoritmo se detiene en una temperatura aceptable o se alcanzo un máximo de iteraciones sin encontrar mejoras significativas.

1.3. Ventajas

- **Versatilidad**: El *Simulated Annealing* no requiere un conocimiento previo del problema, lo que lo convierte en un algoritmo versátil, aplicable a una amplia gama de problemas de optimización.
- **Exploración eficaz**: Gracias a su capacidad de escapar de óptimos locales, el algoritmo permite explorar el espacio de soluciones y aumenta la probabilidad de encontrar soluciones globales óptimas.
- **Facilidad de implementación**: El algoritmo es relativamente sencillo de implementar, ya que no requiere operaciones complejas como derivadas ni grandes cantidades de datos. Los cálculos son directos y accesibles.

1.4. Desventajas

- **Soluciones subóptimas**: Aunque el *Simulated Annealing* aumenta las probabilidades de encontrar una solución óptima, no garantiza que se logre el resultado deseado. En algunos casos, el algoritmo puede quedarse atrapado en óptimos locales.
- **Sensibilidad a los parámetros**: El rendimiento del algoritmo depende de la correcta configuración de múltiples parámetros, como la temperatura inicial, la tasa de enfriamiento y el número de iteraciones. Ajustar estos parámetros puede ser un proceso complejo y que requiera varias pruebas y ajustes.
- **Dependencia de la aleatoriedad**: Al ser un algoritmo estocástico, el resultado puede depender de la aleatoriedad del proceso, lo que significa que el rendimiento puede variar considerablemente entre distintas ejecuciones.

1.5. Aplicaciones

- Refinamiento cristalográfico de estructuras macro moleculares biológicas. Se busca disminuir la brecha de los datos reales de los observados en laboratorio en función de ciertos parámetros [3].
- Definir la topología de volcanes y la perdida de energía de muones (partícula elemental similar al electrón) en la roca dada una imagen que se basa en muones atmosféricos de alta energía(herramienta geofísica para comprender el subsuelo de la Tierra) [4].

- Reconocimiento biométrico de personas mediante SA. Utilizado para clasificar e identificar personas dados datos biométricos acústicos y visuales [5].
- Reconstrucción de tomografías computarizadas por emisión de fotones (prueba que utiliza una sustancia radioactiva y una cámara especial para crear imágenes tridimensionales) [6].

1.6. Simulated Annealing vs Random Mutation Hill Climbing

Característica	**Simulated Annealing (SA)**	**Random Mutation Hill Climbing (RMHC)**
Naturaleza	Estocástico con control de aceptación de peores soluciones	Estocástico, pero acepta solo mejoras
Inspiración	Enfriamiento de metales en física	Subida de colinas con mutaciones aleatorias
Óptimo global	Sí (con una probabilidad que disminuye con el tiempo)	No
Salir del óptimo local	Sí (probabilidad de salir, probabilidad de empeorar)	No
Exploración	Alta al inicio, luego se vuelve más explotativo	Limitada, puede atascarse en óptimos locales
Parámetros	temperatura, temperatura mínima, enfriamiento, iteraciones	Numero de iteraciones, máximo de búsqueda en vecindad
Convergencia	Lenta pero robusta	Rápida pero propensa a estancarse
Uso típico	Problemas complejos de optimización global	Problemas donde hay muchas mejoras locales pequeñas
Complejidad de implementación	Moderada, debido al control de temperatura y aceptación probabilística	Baja, simple de implementar

Tabla 1: Comparativa entre Simulated Annealing y Random Mutation Hill Climbing

2. Pseudocódigo

Algorithm 1 Simulated Annealing

```
current_state = random state in states
old_energy = cost(current_state)
for temp = temp_max to temp_min step next_emp do
    for i = 0 to iMax do
        neighbour = successor_func(current_state)
        new_energy = cost(neighbour)
        delta = new_energy − old_energy
        if delta > 0 then
            if random() < exp(-delta / (K * temp)) then
                old_energy = new_energy
            end if
        else
            old_energy = new_energy
        end if
    end for
end for
```

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 puede ser cualquier cadena de tamaño n cuyo peso no exceda el peso máximo:

$$s_0 = \{x \in \{0, 1\}^n | w(x) \leq W\}$$

- 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 genere el vecino sera *Bit flip* que intercambia un 0 por un 1 y viceversa en una posición aleatoria 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 y regresando a la primera.

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:

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

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 puede ser cualquier permutación de C:

$$s_0 = \{x \in [1, n]^n \mid x \text{ es una permutación de } C\}$$

- 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. Simulated Annealing

Se define una clase de *SimulatedAnnealing* que permita realizar el recocido simulado en el código 1. Requiere de los siguientes parámetros:

- *information*: Información adicional necesaria para proponer la solución inicial del problema y para calcular la energía del sistema.
- *energy_function*: Función que calcula la energía del sistema. Recibe como parámetros una solución y la información adicional.
- *initial_solution_function*: Función generadora de soluciones iniciales. Se aleatorizá el inicio de la búsqueda para aumentar la capacidad exploratoria del algoritmo.
- *generate_neighbour_function*: Función que obtiene un vecino de forma aleatoria dada una solución.

Listing 1: Constructor de la clase SimulatedAnnealing

```
1 class SimulatedAnnealing:
2   def __init__(self, information, energy_function ,
       initial_solution_function,
       generate_neighbour_function):
3     self.information = information
4     self.energy_function = energy_function
5     self.initial_solution_function =
         initial_solution_function
6     self.generate_neighbour_function =
         generate_neighbour_function
7     self.solution = self.initial_solution_function(self.
         information)
```

La función 2 ejecuta el algoritmo de recocido simulado. Recibe los siguientes parámetros:

- *temperature*: Es la temperatura inicial del sistema.
- *min_temperature*: Es la temperatura mínima que el sistema puede adquirir antes de finalizar.
- *alpha*: Es la tasa de decrecimiento de la temperatura, debe de estar en el rango $[0, 1]$.
- *max_iter*: Es el numero de iteraciones a realizar antes de actualizar la temperatura del sistema.

Primero se genera una solución aleatoria. Después se itera siempre que la temperatura sea mayor a la temperatura mínima y sobre el numero de iteraciones.

E cada iteración se visita un nuevo vecino y se calcula la energía. Si es mayor entonces se se actualizan todos los datos. En caso contrario se genera un numero aleatorio en el rango de $[0, 1]$ y se mueve a esa dirección si es menor a $e^{\frac{delta}{temperature}}$. En cada iteración se almacena la mejor solución con su respectiva energía, esto dado que el algoritmo se puede salir de óptimos globales.

Listing 2: Función simpleSimulatedAnnealing

```
def simpleSimulatedAnnealing(self, temperature,
    min_temperature, alpha, max_iter):
  current_solution = self.solution
  current_energy = self.energy_function(current_solution,
      self.information)
  best_solution = current_solution[:]
  best_energy = current_energy

  while temperature > min_temperature:
    for _ in range(max_iter):
      neighbour = self.generate_neighbour_function(
          current_solution)
      neighbour_energy = self.energy_function(neighbour,
          self.information)

    delta = neighbour_energy - current_energy
    if delta > 0 or random.random() < math.exp(delta /
        temperature):
      current_solution = neighbour
      current_energy = neighbour_energy

      if current_energy > best_energy:
        best_solution = current_solution[:]
        best_energy = current_energy

    temperature *= alpha

        self.solution = best_solution
```

Cada problema debe de incluir información, una función de energía, una función generadora de soluciones iniciales y una función para obtener un vecino de forma aleatoria.

3.4.2. Knapsack problem

La función 3 genera de manera aleatoria un conjunto de elementos en la mochila con pesos y valores aleatorios en un rango de $[1, 10]$. La función 4 calcula la energía del sistema la cual suma todos los pesos y valores de los elementos que se encuentran en la solución, si el peso es menor al capacidad de la mochila entonces devuelve el valor de la mochila, en caso contrario devuelve la diferencia de el peso actual menos la capacidad máxima.

La función 5 genera soluciones aleatorias de combinaciones y no regresa ninguna de ellas hasta que el peso de la solución sea menor a la capacidad máxima. Finalmente, la función 6 se encarga de generar un vecino de forma aleatoria, primero selecciona un elemento aleatorio del vector y después lo invierte.

Listing 3: Función *generate_information* de Knapsack problem

```
def generate_information(self, items, capacity):
  self.information = {
    "items": items,
    "values": [(random.randint(1,10), random.randint(1,
       10)) for _ in range(items)],
    "capacity": capacity
  }
```

Listing 4: Función *energy* de Knapsack problem

```
def energy(self, solution, information):
  total_weight = total_value = 0
  for i in range(len(solution)):
    if solution[i] == 1:
      total_weight += information['values'][i][0]
      total_value += information['values'][i][1]

  if total_weight > information['capacity']:
    return information['capacity'] - total_weight
  return total_value
```

Listing 5: Función *generate_initial_solution* de Knapsack problem

```
def generate_initial_solution(self, information):
  while True:
    solution = [random.randint(0,1) for _ in information[
       'values']]
    if self.energy(solution, information) > 0:
      return solution
```

Listing 6: Función *random_neighbour* de Knapsack problem

```
def random_neighbour(self, solution):
  neighbour = solution[:]
  index = random.randint(0, len(solution) - 1)
  neighbour[index] = not neighbour[index]
  return neighbour
```

La interfaz de la figura 1 simula 125 elementos generados de forma aleatorio para ser metidos en una mochila 200 de capacidad. En verde son los objetos que están en la prueba actual de la mochila y en gris los que no. Adicionalmente se utiliza un cuadrado verde para validar que el peso actual sea menor que la capacidad máxima de la mochila.

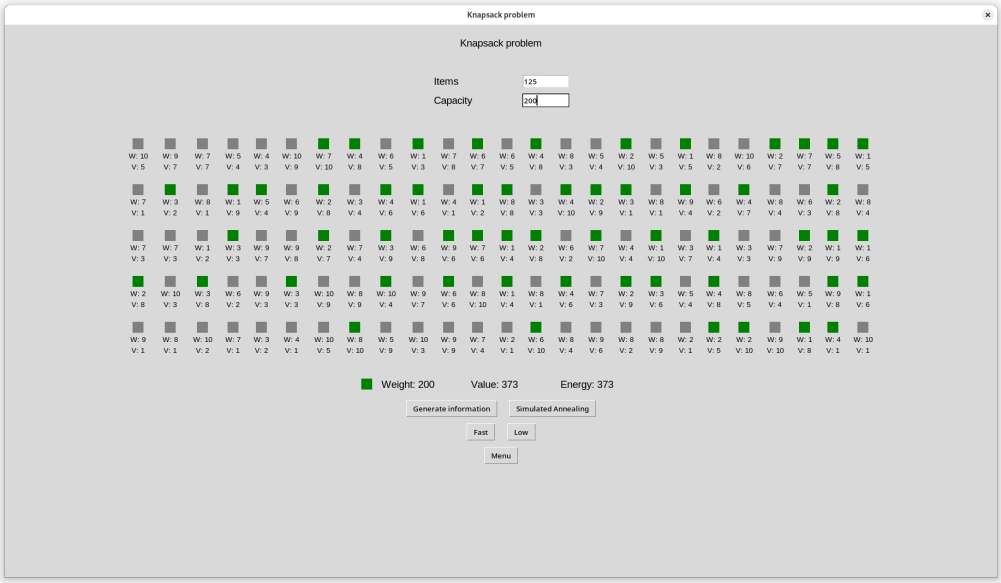

Figura 1: Knapsack Problem Interface

3.4.3. Travel Salesman Problem

La función 7 genera una matriz cuadrada y simétrica de n valores aleatorios en un rango de $[1, 100]$.

La función 8 calcula la energía del sistema. Dado un vector suma todas las distancias de las ciudades con base en la información generada en 7, el resultado que devuelve es negativo ya que se busca minimizar. La función 9 genera un vector de n números consecutivos (representando las ciudades) y cambia las posiciones mediante la función *shuffle*.

Finalmente, la función 10 toma dos indices aleatorios diferentes e invierte los valores de dichas posiciones del vector.

Listing 7: Función *generate_information* de Travel Salesman Problem

```
def generate_information(self, cities):
  distances = [[0]*cities for _ in range(cities)]

  for i in range(cities):
    for j in range(i, cities):
      if i == j:
        valor = 0
      else:
        valor = random.randint(1, 100)
      distances[i][j] = valor
      distances[j][i] = valor

  self.information = {
    "cities" : cities,
    "distances" : distances
  }
```

Listing 8: Función *energy* de Travel Salesman Problem

```
def energy(self, solution, information):
  distance = 0
  num_cities:int = len(solution)

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

  return -distance
```

Listing 9: Función *generate_initial_solution* de Travel Salesman Problem

```
def generate_initial_solution(self, information):
  solution =  list(range(information['cities']))
  random.shuffle(solution)
  return solution
```

Listing 10: Función *random_neighbour* de Travel Salesman Problem

```
def random_neighbour(self, solution):
  neighbour = solution[:]
```

```
  i = j = random.randint(0, len(solution) - 1)
    while j == i:
      j = random.randint(0, len(solution) - 1)
  neighbour[i], neighbour[j] = neighbour[j], neighbour[i]

return neighbour
```

La interfaz de la figura 2 simula 25 ciudades y muestra la exploración de diferentes posibilidades en el espacio de búsqueda. El botón de *distances* muestra la matriz de adyacencia del grafo.

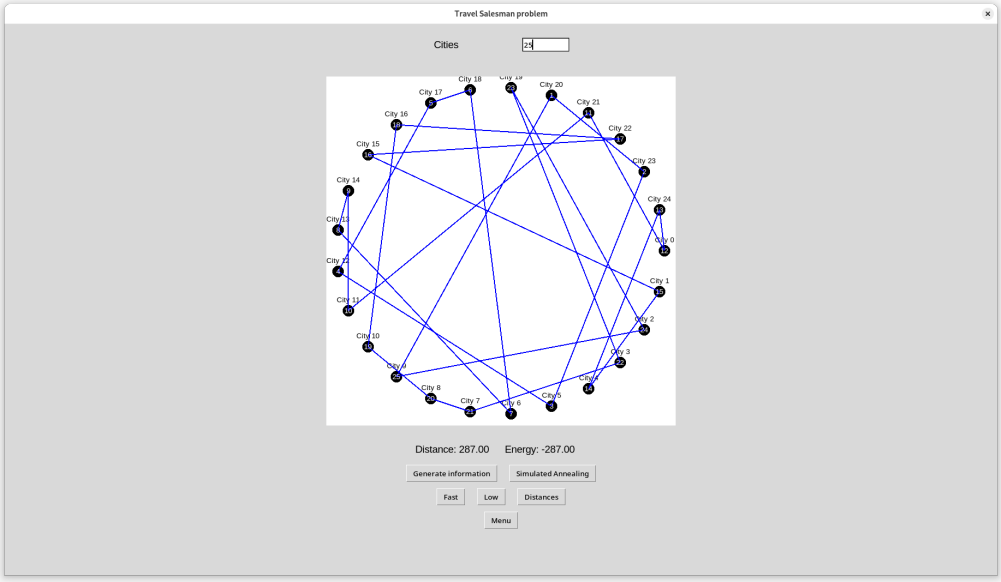

Figura 2: Travel Salesman Problem Interface

3.4.4. Minimizar la función

La función 11 unicamente define el tamaño del vector y los rangos de valores. por otro lado, la función 12 calcula la energía del sistema dada por la suma de los cuadrados, dado que es una función de minimización se invierte el signo.

La función 13 genera un vector de n elementos aleatorios en los rangos definidos, mientras que la función 14 suma o resta en uno a un elemento aleatorio del vector (dado que se considera una configuración circular, se ajusta el valor si el nuevo valor no se encuentra en el rango).

Listing 11: Función *generate_information* de SumFunctionProblem

```
def generate_information(self, size, min, max):
  self.information = {
    "size": size,
    "min": min,
    "max": max
  }
```

Listing 12: Función *energy* de SumFunctionProblem

```
def energy(self, solution, _):
  total_sum:float = 0

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

  return -total_sum
```

Listing 13: Función *generate_initial_solution* de SumFunctionProblem

```
def generate_initial_solution(self, information):
  solution = [random.randint(information['min'],
     information['max']) for _ in range(information['size
     '])]
  return solution
```

Listing 14: Función *generate_neighbour* de SumFunctionProblem

```
def random_neighbour(self, solution):
  neighbour = solution[:]
  index = random.randint(0, len(solution) - 1)
  sign = random.choice([-1 , 1])
  neighbour[index] += sign

  if neighbour[index] > self.information['max']:
    neighbour[index] = self.information['min']

  if neighbour[index] < self.information['min']:
    neighbour[index] = self.information['max']

  return neighbour
```

La interfaz de la figura 3 simula un vector de 30 elementos en el rango de -10 a 10. Las barras crecen y decrecen según se explore el espacio de búsqueda.

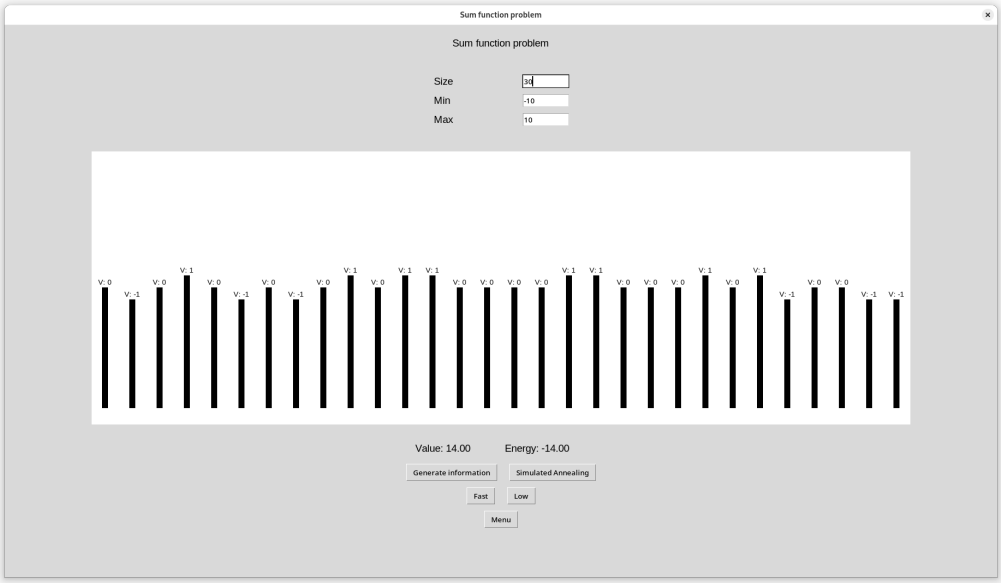

Figura 3: Sum Function Problem Interface

4. Conclusión

Simulated Annealing tiene muchas mas posibilidades de encontrar un óptimo global pero, de forma redundante se encuentra a su propio problema de optimización el cual consiste en definir los meta parámetros del algoritmo, como la temperatura máxima, mínima, tasa de decrecimiento, etc. Podria resultar interesante aplicar Simulated Annealing para si mismo y analizar la influencia de dichos meta parámetros.

Otro problema es la precisión al momento de encontrar una mejor solución, pues aunque el algoritmo no se haya estancado en alguna posición, si se detiene cuando la temperatura ha superado el umbral establecido.

La implementación de un almacenamiento de los mejores resultados permite registrar si en algún punto se desvío del camino, esto ayudaría a reiniciar el algoritmo pero colocando esa posición como la solución inicial, o incluso, si sabemos que el resultado es próximo, podría optarse por otros métodos como *hill Climbing*, para encontrar una solución mas precisa.

Referencias

[1] L. Gavin. (2025) Recocido explicado: proceso, tipos y ventajas. Accedido el 6 de abril de 2025. [Online]. Available: https://www.madearia.com/es/blog/annealing-explained-process-types-and-advantages/

[2] Alsimet. (2025) Tratamientos térmicos: el recocido de metal. Accedido el 6 de abril de 2025. [Online]. Available: https://www.alsimet.es/es/noticias/tratamientos-termicos-el-recocido-de-metal

[3] A. T. Brunger, "Simulated annealing in crystallography," *Annu. Rev. Physical Chemistry*, vol. 42, no. 1, pp. 197–223, Oct. 1991, accedido el 6 de abril de 2025. [Online]. Available: https://doi.org/10.1146/annurev.pc.42.100191.001213

[4] A. V.-R. et al., "Simulated annealing for volcano muography," *J. South Amer. Earth Sci.*, vol. 109, p. 103248, Aug. 2021, accedido el 6 de abril de 2025. [Online]. Available: https://doi.org/10.1016/j.jsames.2021.103248

[5] L. Huang, C. Yu, H. Zhuang, and S. Morgera, "Biometric fusion by simulated annealing," *Int. J. Knowledge-based Intell. Eng. Syst.*, vol. 16, no. 2, pp. 87–98, Mar. 2012, accedido el 6 de abril de 2025. [Online]. Available: https://doi.org/10.3233/kes-2010-0234

[6] S. Webb, "Spect reconstruction by simulated annealing," *Phys. Medicine Biol.*, vol. 34, no. 3, pp. 259–281, Mar. 1989, accedido el 6 de abril de 2025. [Online]. Available: https://doi.org/10.1088/0031-9155/34/3/001