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

Ingeniería en Inteligencia Artificial, Metaheuristicas
Fecha: 18 de mayo de 2025

Algoritmos genéticos

Presenta

Angeles López Erick Jesse[1]

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

Resumen

Se describe el comportamiento de los algoritmos geneticos para resolver problemas de optimización local.

Palabras clave: Algoritmo, Algoritmo genético, Resultado óptimo

[1] eangelesl1700@alumno.ipn.mx

Índice

1. Algoritmos genéticos

1.1. Evolución natural

La evolución, en relación con la genómica, es el proceso por el cual los organismos vivos cambian con el tiempo a través de cambios en el genoma, esto cambios provocan individuos con rasgos alterados que afectan su supervivencia. Los supervivientes se reproducen y trasmiten estos genes alterados, los cambios que atentan contra la supervivencia de algún individuo, impide la reproducción del mismo [1].

En la naturaleza, para que exista un proceso evolutivo deben de cumplirse las siguientes condiciones:

- Una entidad o individuo con la capacidad de reproducción.
- Una población de dichos individuos.
- Diferencias entre los individuos de la población.
- La variedad es un factor que determina el nivel de supervivencia de ese individuo.

La evolución afecta los cromosomas, estos son estructuras que transporta la información genómica de una célula a otra, y es mediante la reproducción en donde se combinan los cromosomas de los padres para formar nuevas estructuras [2, 3].

1.2. Evolución artificial

Los algoritmos genéticos simulan el comportamiento evolutivo de una población en donde los mas aptos heredan sus genes a las nuevas generaciones para obtener nuevos resultados. Este proceso tiene los siguientes pasos:

- **Selección**: Se seleccionan las parejas (o grupos) de individuos de la población con las mejores aptitudes.
- **Cruza**: Se realiza una combinación entre los genomas de las parejas seleccionadas para producir un numero de hijos con códigos genéticos diferentes.
- **Mutación**: Sea realiza algún cambio aleatorio en el genoma de cualquier elemento de la nueva población.
- **Reemplazo**: Criterio que define que elementos de la nueva generación reemplazaran a la generación anterior.

1.3. Ventajas

- Puede explorar el espacio de soluciones en múltiples direcciones al mismo tiempo.

- Realizan una amplia exploración, permitiendo escapar de óptimos locales para conseguir óptimos globales.

- Trabajan bien en problemas complejos y cambiantes, así como aquéllos en los que la función objetivo es discontinua, ruidosa, o que tiene muchos óptimos locales, ademas de poner soportar las optimizaciones multiobjetivo.

1.4. Desventajas

- La función objetvo es muy sensible para los algotimos GA, ya que si se elige mal o se define incorrectamente, puede que sea incapaz de encontrar una solución al problema.

- La elección de los parámentros (tamaño de la población, cruzamiento, selección de padres, mutación, entre otras más) de los algoritmos GA puede llegar a ser muy complejo. Una mala elección en los parámetros puede provocar un mal desempeño.

- Consumen mucho tiempo de ejecución y potencia de cómputo.

- Existe el riesgo de encontrarse con una convergencia prematura; es decir, se puede reproducir abundantemente un individuo haciendo que merme la diversidad de la población demasiado pronto, provocando que converja hacia un óptimo local, el cual representa a ese individuo.

1.5. Fenotipo y genotipo

El fenotipo es la forma que toma la posible solución del problema, como números, cadenas, grafos, tablas, imágenes, etc. El **genotipo** (también llamado cromosoma) esta construido a partir del fenotipo y representa la codificación de sus características.

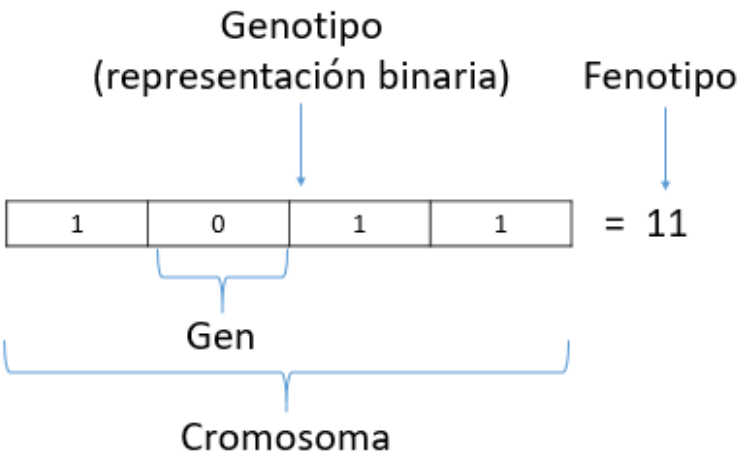

Figura 1: Descripción grafica de genotipo, fenotipo, cromosoma y gen del numero “11”

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

```
1         def simpleSimulatedAnnealing(self, temperature,
              min_temperature, alpha, max_iter):
2         current_solution = self.solution
3         current_energy = self.energy_function(
              current_solution, self.information)
4         best_solution = current_solution[:]
5         best_energy = current_energy
6
7         while temperature > min_temperature:
8         for _ in range(max_iter):
9         neighbour = self.generate_neighbour_function(
              current_solution)
10        neighbour_energy = self.energy_function(neighbour
              , self.information)
11
12        delta = neighbour_energy - current_energy
13        if delta > 0 or random.random() < math.exp(delta
              / temperature):
14        current_solution = neighbour
15        current_energy = neighbour_energy
16
17        if current_energy > best_energy:
18        best_solution = current_solution[:]
19        best_energy = current_energy
20
21        temperature *= alpha
22
23        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

```
1         def generate_information(self, items, capacity):
2         self.information = {
3             "items": items,
4             "values": [(random.randint(1,10), random.
                  randint(1, 10)) for _ in range(items)],
5             "capacity": capacity
6         }
```

Listing 4: Función *energy* de Knapsack problem

```
1         def energy(self, solution, information):
2         total_weight = total_value = 0
3         for i in range(len(solution)):
4         if solution[i] == 1:
5         total_weight += information['values'][i][0]
6         total_value += information['values'][i][1]
7
8         if total_weight > information['capacity']:
9         return information['capacity'] - total_weight
10        return total_value
```

Listing 5: Función *generate_initial_solution* de Knapsack problem

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

Listing 6: Función *random_neighbour* de Knapsack problem

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

La interfaz de la figura 2 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.

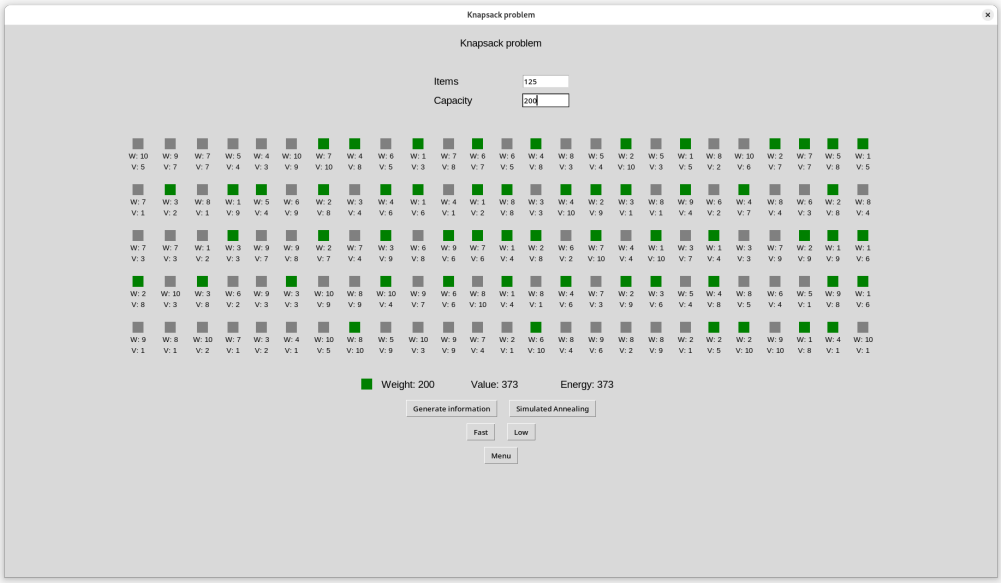

Figura 2: Knapsack Problem Interface

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 3 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.

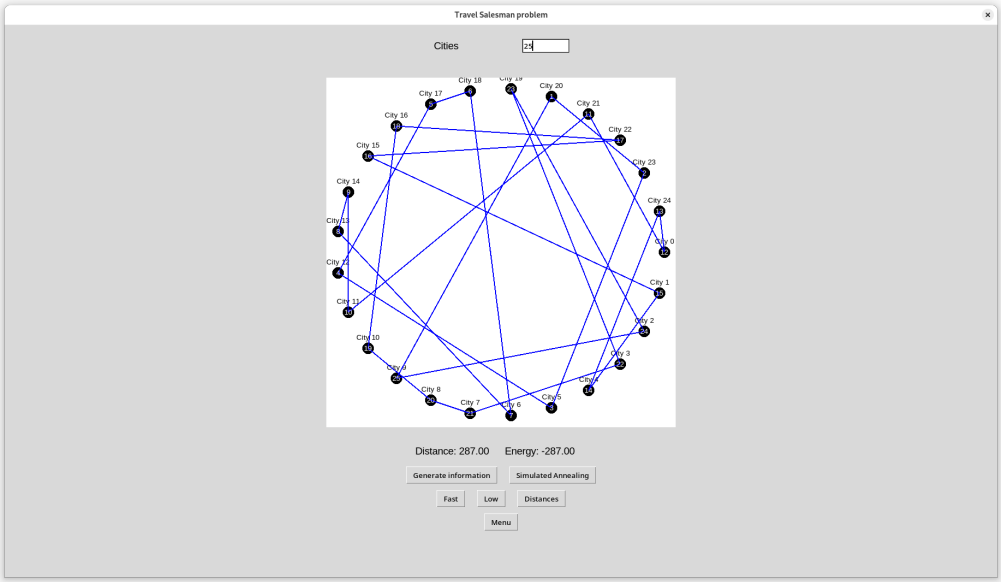

Figura 3: Travel Salesman Problem Interface

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 4 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.

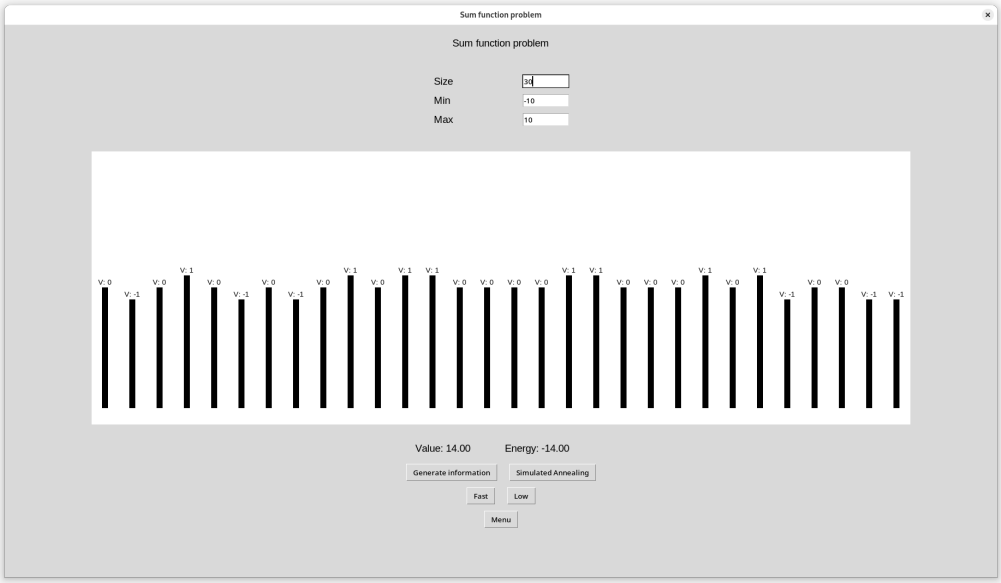

Figura 4: Sum Function Problem Interface

4. Problemas de optimización CEC 2017

En el documento [**?**] 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 5

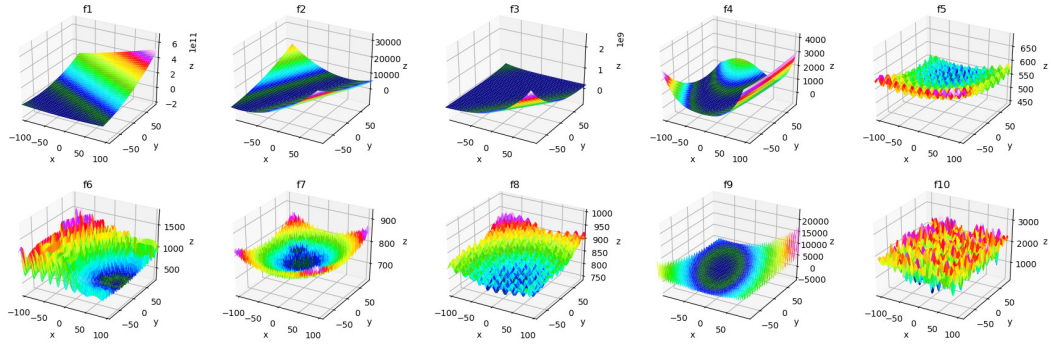

Figura 5: Superficies ploteadas de las 10 primeras funciones para dos dimensiones [**?**]

4.2. Código

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

Para el algoritmo de *Simulated Annealing* se usa el mismo código que para los problemas anteriores, por lo que unicamente es necesario diseñar la función

de generación de información, energía, generador de soluciones iniciales y la función para obtener un vecino aleatorio.

La función que genera la información del código 15 define el rango de valores, la función a utilizar de los problemas, la dimensión de la solución y un *alpha* que determina la distribución de probabilidad dado que el espacio es continuo.

Listing 15: Función *generate_information*

```
1        def generate_information(self, function, low,
             high, dimention, alpha = 1):
2        self.information = {
3            "low" : low,
4            "hight": high,
5            "dimention": dimention,
6            "function": function,
7            "alpha": alpha
8        }
```

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

Listing 16: Función *energy*

```
1        def energy(self, solution, information):
2        return - self.information["function"]([solution])
             [0]
```

La función 17 se por el propio problema de optimización la cual recibe como parámetros el vector del estado actual (código 16).

Listing 17: Función *generate_initial_solution*

```
1        def generate_initial_solution(self, information):
2        solution = np.random.uniform(
3        low=self.information["low"],
4        high=self.information["hight"],
5        size=self.information["dimention"]).tolist()
6
7        return solution
```

Por otro lado, el código 18 muestra la función para obtener a un vecino de forma aleatoria. Dado que las funciones existen en un espacio de valores reales, se opta por avanzar una cantidad aleatoria en cualquier dirección.

Listing 18: Función *random_neighbour*

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

```
        index = random.randint(0, len(solution) - 1)
        alpha = random.uniform(-self.information["alpha"
            ], self.information["alpha"])

        neighbour[index] += alpha
        return neighbour
```

El código 19 itera sobre sobre las 10 funciones, genera la información con una solución de 100 dimensiones y genera 20 datos por función.

Listing 19: Gneración de datos

```
        for function in functions:
        cec_problem = Cec2017()
        dimension = 5
        cec_problem.generate_information(function, -100,
            100, 100, dimension)

        sa = SimulatedAnnealing(cec_problem.information,
        cec_problem.energy,
        cec_problem.generate_initial_solution,
        cec_problem.random_neighbour)

        for i in range(20):
        sa.reset()

        start_time = time.time()
        while sa.temperature > sa.min_temperature:
        sa.stepSimpleSimulatedAnnealing()
        elapsed_time = time.time() - start_time

        record = {
            "function": function.__name__,
            "energy": sa.energy,
            "time_seconds": elapsed_time,
            "dimension": dimension
        }

        dataset.append(record)
```

4.3. Resultados

Considerando vectores de tamaño 100, una temperatura inicial de 1000, una temperatura mínima de 1, 100 iteraciones (dentro de *Simulated Annealing*)y 20 iteraciones sobre cada función. Se obtuvieron los resultado de la tabla 1 y

la tabla 2.

Adicionalmente, la clase *Simulated Annealing* 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	-1.2460e+10	-2.4494e+04	-6.2340e+08	-4.9583e+04	2.7861e+09
f2	-2.6553e+97	-1.1543e+10	-1.3276e+96	-3.7496e+14	5.9373e+96
f3	-1.0030e+06	-8.5706e+05	-9.1060e+05	-9.0083e+05	4.3764e+04
f4	-1025.97	-568.33	-655.39	-617.40	106.62
f5	-2755.05	-1976.85	-2464.06	-2579.62	274.80
f6	-905.67	-768.02	-830.47	-831.76	43.01
f7	-9114.76	-3071.32	-4114.49	-3701.26	1358.74
f8	-3125.81	-2326.66	-2670.50	-2676.52	259.03
f9	-71191.06	-65976.84	-68020.84	-68168.09	1238.82
f10	-17334.65	-12570.57	-14615.18	-14440.98	1404.16

Tabla 1: Estadísticas de energía por función.

f(x)	**Peor**	**Mejor**	**Promedio**	**Mediana**	**Desviación estándar**
f1	0.5162	0.4168	0.4609	0.4649	0.0266
f2	0.5118	0.4309	0.4777	0.4790	0.0250
f3	0.5805	0.4552	0.5201	0.5231	0.0295
f4	0.5738	0.4678	0.5154	0.5051	0.0337
f5	0.5311	0.4374	0.4921	0.4915	0.0256
f6	0.5812	0.4854	0.5318	0.5296	0.0290
f7	0.7874	0.6239	0.6861	0.6834	0.0465
f8	0.6256	0.4952	0.5541	0.5497	0.0320
f9	0.6721	0.5378	0.5995	0.5967	0.0425
f10	0.7984	0.6259	0.6938	0.6884	0.0468

Tabla 2: Estadísticas de tiempo de ejecución por función.

5. 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.

Respecto a los resultados obtenidos mediante *Random Mutation Hill Climbing* a los problemas del Cec 2017, los peores valores fueron incluso mejores, lo que indica que, aunque *Hill Climbing* puede obtener valores con mayor precisión de forma local, le es mas difícil conseguir un mejor resultado dado que no puede escapar de los óptimos locales.

Como próximas pruebas se plantea la posibilidad de utilizar las mismas semillas para ambos algoritmos y conocer el comportamiento de ambos dadas las condiciones iniciales. Ademas, se propone utilizar el algoritmo de *Simulated Annealing* sobre si mismo para calcular los mejores parámetros iniciales que puedan reducir el numero de operaciones realizadas. Finalmente, se propone la mezcla de ambos, utilizando primero *Simulated Annealing* para encontrar el óptimo global (o intentar encontrarlo) y utilizar *Hill Climbing* para aumentar la precisión de la solución obtenida.

Referencias

[1]

[2]

[3]