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

Ingeniería en Inteligencia Artificial, Metaheuristicas
Fecha: 20 de mayo de 2025

Algoritmos genéticos

Presenta

Angeles López Erick Jesse[1]

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

[1] eangelesl1700@alumno.ipn.mx

Resumen

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

Palabras clave: Algoritmo, Algoritmo genético, Resultado óptimo

Índice general

Introducción

Capítulo 1

Algoritmo genético

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 [**?**].

En la naturaleza, para que exista un proceso evolutivo se deben cumplir 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 [**?**, **?**].

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. Este conjunto serán los padres de la nueva generación.

- **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.

1.2.1. 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 poder soportar las optimizaciones multiobjetivo.

- Cada parte del proceso puede ser implementada por diferentes funciones para mejorar el rendimiento del algoritmo, por ejemplo, existen funciones enfocadas unicamente en la exploración.

1.2.2. 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.

1.2.3. 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, generalmente como un vector.

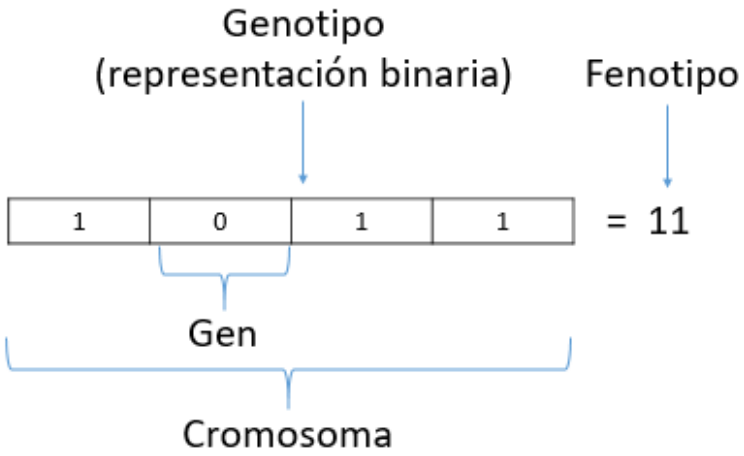

Figura 1.1: Descripción grafica de genotipo, fenotipo, cromosoma y gen del numero “11”

1.2.4. Modelos generacionales y estacionarios

En cada paso evolutivo, la cantidad de la población seleccionada y la forma de como se reintegran los nuevos individuos afectan el comportamiento general de la población, la velocidad en como evoluciona y el espacio de exploración para la búsqueda de óptimos globales.

Los **Modelos generacionales** actualizan la mayor parte de la población, se busca tener evoluciones rápida y una amplia exploración. Para ello se selecciona un pequeño conjunto de individuos y se les aplica los algoritmos de selección, cruza y mutación. Finalmente, mediante el algoritmo de reemplazo se reintegran en la población mediante una función.

Por otro lado, los **modelos estacionarios** buscan mantener el equilibrio en la mayor parte de la población experimentando y mezclando un pequeño conjunto. El cambio es mas lento pero mas estable.

Capítulo 2

Operador de Selección

La función de selección elige las parejas (o subconjuntos) de individuos que fungirán como padres para reproducirse, combinar sus genes y generar a la siguiente nueva generación.

A continuación se presentan algunas funciones de selección, las cuales (en su mayoria) requieren de los siguientes parámetros:

- population: Subconjunto de la población a la cual se le aplica el algoritmo.
- objective: Función de evaluación.

Algunas de las funciones son las siguientes:

2.1. Universal Random

La selección aleatoria universal (*Universal Sampling*) es una variante estocástica de la selección proporcional que busca reducir la varianza introducida por la aleatoriedad. En lugar de seleccionar los individuos uno por uno, se generan múltiples puntos equidistantes sobre el intervalo $[0, 1]$ para garantizar una muestra más representativa de la población, según sus probabilidades acumuladas.

Sea n el número de individuos a seleccionar, se generan n números equidistantes:

$$r_i = \frac{i + u}{n} \quad \text{para } i = 0, 1, ..., n - 1$$

donde $u \sim U(0, 1)$ es un número aleatorio uniformemente distribuido.

Cada número r_i se mapea sobre la distribución acumulada de probabilidades para determinar a qué individuo seleccionar.

Algorithm 1 Universal Random Selection
Input: {population, objective}

scores ← [objective(i) for i in population]
if any score < 0 **then**
    scores ← scores - min(scores) ▷ Normalización para evitar valores negativos
end if
total ← sum(scores)
probabilities ← [score / total for score in scores]
cumulative ← acumulada(probabilities)
u ← random.uniform(0, 1)
for $i = 0$ **to** population_size **do**
    $r \leftarrow \frac{i+u}{population\_size}$
    parents[i] ← individuo correspondiente a r en la distribución acumulada
end for
return parents

Este método asegura una cobertura más uniforme de la población en comparación con la selección proporcional simple, reduciendo la varianza en la selección. Es particularmente útil cuando se desea mantener una representación proporcional sin depender completamente de la aleatoriedad individual de cada extracción.

2.2. Tournament

El algoritmo 2 recibe de manera adicional el parámetro *selection_rate*, que define el porcentaje de la población que participará en cada torneo.

Dada una población P, se selecciona aleatoriamente un subconjunto $Q \subseteq P$. El individuo q_i con la mayor aptitud $f(q_i)$ es seleccionado para formar parte del conjunto de padres S:

$$S = \left\{ \arg\max_{q \in Q} f(q) \,\middle|\, Q \subseteq P \right\}$$

Algorithm 2 Tournament Selection
Input: {population, objective, selection_rate}

```
for i = 0 to population_size do
    k ← population_size × selection_rate
    candidates ← random.sample(population, k)
    scores ← []
    for cada j en candidates do
        scores.append(objective(j))
    end for
    max_score ← max(scores)
    index ← scores.index(max_score)
    parents[i] ← candidates[index]
end for
return parents
```

Este método es eficiente para poblaciones de cualquier tamaño. Además, el parámetro *selection_rate* permite controlar la presión selectiva: torneos pequeños fomentan la diversidad, mientras que torneos grandes intensifican la explotación de los individuos más aptos.

2.3. Proportional

La selección proporcional (también conocida como *roulette wheel selection*) asigna a cada individuo una probabilidad de ser seleccionado proporcional a su aptitud. Es un método estocástico que favorece a los individuos más aptos, pero permite la selección de individuos menos aptos con menor probabilidad, promoviendo así la diversidad.

Sea P la población con n individuos y $f(p_i)$ la función de aptitud del individuo p_i. La probabilidad de seleccionar al individuo p_i se define como:

$$\Pr(p_i) = \frac{f(p_i)}{\sum_{j=1}^{n} f(p_j)}$$

Algorithm 3 Proportional Selection
Input: {population, objective}

```
scores ← [objective(i) for i in population]
if any score < 0 then
    scores ← scores - min(scores)    ▷ Normalización para evitar negativos
end if
total ← sum(scores)
probabilities ← [score / total for score in scores]
cumulative ← acumulada(probabilities)
for i = 0 to population_size do
    r ← random.uniform(0, 1)
    parents[i] ← individuo correspondiente a r en la distribución acumulada
end for
return parents
```

Este método es intuitivo y funciona bien cuando las diferencias de aptitud entre individuos no son extremas. Sin embargo, si las diferencias son muy marcadas, puede provocar convergencia prematura. También es sensible a funciones de aptitud negativas o cercanas a cero, por lo que puede requerir normalización.

2.4. Negative Assortative Mating

El apareamiento negativo asortativo busca maximizar la diversidad genética al emparejar individuos que sean lo más diferentes posible entre sí. Este método es útil para evitar la convergencia prematura, ya que fomenta la exploración del espacio de soluciones.

Se utiliza una función de similitud $d(a, b)$ que cuantifica cuán parecidos son dos individuos a y b. Esta función puede estar basada en distancia euclidiana, Hamming, o alguna otra métrica según el tipo de representación del cromosoma. El objetivo es seleccionar, para cada individuo, una pareja que minimice dicha similitud.

$$\text{Para cada } p_i \in P_{\text{seleccionado}}, \quad p_j = \arg\min_{x \in P} d(p_i, x)$$

Algorithm 4 Negative Assortative Mating
Input: {population, similarity_function}

```
num_pairs ← population_size / 2
selected ← random.sample(population, num_pairs)
parents ← []
for cada p_i en selected do
    distances ← [similarity_function(p_i, x) for x in population]
    remove p_i de la población temporal para evitar emparejar consigo mismo
    partner ← individuo con menor valor de distancia
    parents.append((p_i, partner))
end for
return parents
```

Este método es particularmente útil en etapas tempranas del algoritmo evolutivo, donde la diversidad es crucial. Sin embargo, puede ser computacionalmente costoso, especialmente si se requiere calcular similitud entre todos los pares posibles en poblaciones grandes. Además, no garantiza que los individuos seleccionados sean de alta aptitud, por lo que suele combinarse con otros métodos de selección para mejorar su eficacia.

Capítulo 3

Operador de Cruza

Los operadores de cruza permiten generar nuevos individuos (hijos) a partir de dos padres, combinando partes de sus cromosomas. A continuación se describen distintos métodos de cruza utilizados en algoritmos evolutivos.

3.1. One Point

Dados dos padres p_1 y p_2 con cromosomas de tamaño T, se selecciona una posición al azar $t \in [1, T-1]$ donde se particionan y combinan los genes:

$$h_1 = [0, t)_{p_1} \cup [t, T)_{p_2}$$
$$h_2 = [0, t)_{p_2} \cup [t, T)_{p_1}$$

Tal como se muestra en la figura 3.1, con $t = 3$.

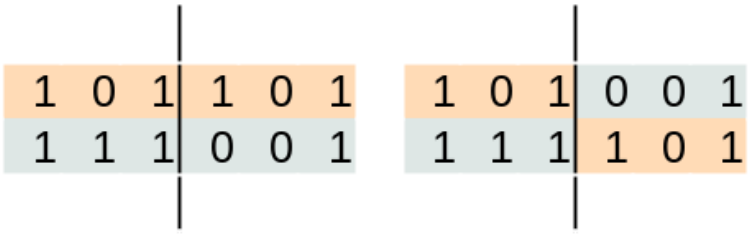

Figura 3.1: Ejemplo de cruza en un solo punto

3.2. Two Point

Tiene un comportamiento similar al anterior. Dados dos padres p_1 y p_2 con cromosomas de tamaño T, se seleccionan dos posiciones al azar t_1, t_2 tales que

$0 \leq t_1 < t_2 \leq T$. Se intercambia el segmento entre ambas posiciones:

$$h_1 = [0, t_1)_{p_1} \cup [t_1, t_2)_{p_2} \cup [t_2, T)_{p_1}$$
$$h_2 = [0, t_1)_{p_2} \cup [t_1, t_2)_{p_1} \cup [t_2, T)_{p_2}$$

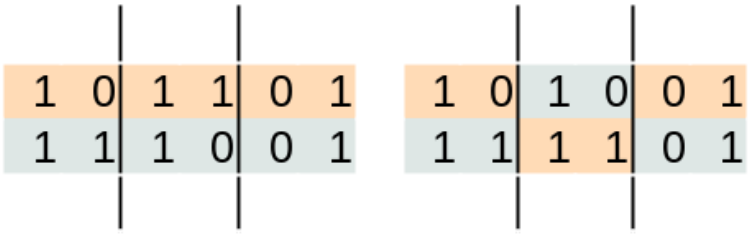

Figura 3.2: Ejemplo de cruza en dos puntos

3.3. Uniform

Dados dos padres p_1 y p_2 con cromosomas de tamaño T, se genera una máscara binaria aleatoria $M \in \{0,1\}^T$. Si $M[i] = 0$, entonces $h_1[i] = p_1[i]$ y $h_2[i] = p_2[i]$; si $M[i] = 1$, entonces $h_1[i] = p_2[i]$ y $h_2[i] = p_1[i]$.

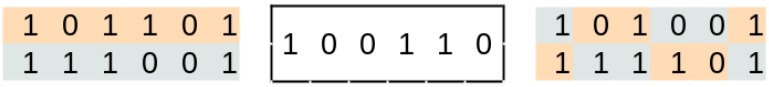

Figura 3.3: Ejemplo de cruza uniforme

3.4. Arithmetic

Para cada par de padres, se genera un valor aleatorio $\alpha \in [0, 1]$ que define una combinación lineal entre sus genes. Para cada posición i del cromosoma:

$$h_1[i] = \alpha \cdot p_1[i] + (1 - \alpha) \cdot p_2[i]$$
$$h_2[i] = (1 - \alpha) \cdot p_1[i] + \alpha \cdot p_2[i]$$

Este operador es común en representaciones reales y mantiene los genes dentro del espacio de búsqueda.

3.5. Blend (BLX-α)

El operador *Blend Crossover* permite generar valores fuera del intervalo definido por los padres. Para cada gen i se define un intervalo extendido:

$$\min_i = \text{mín}(p_1[i], p_2[i]), \quad \max_i = \text{máx}(p_1[i], p_2[i])$$

El hijo se genera como:

$$h[i] = U\left(\min_i - d, \max_i + d\right) \quad \text{donde } d = \alpha \cdot (\max_i - \min_i)$$

El parámetro α controla cuánto se puede explorar fuera del rango original. La función U es la función de distribución uniforme continua.

3.6. Simulated Binary (SBX)

El operador *Simulated Binary Crossover* simula el comportamiento del cruce de un solo punto en cromosomas reales. Dado un parámetro de distribución η, se calcula un factor β basado en una variable aleatoria $u \sim U(0,1)$:

$$\beta = \begin{cases} (2u)^{1/(\eta+1)} & \text{si } u \leq 0{,}5 \\ \left(\frac{1}{2(1-u)}\right)^{1/(\eta+1)} & \text{si } u > 0{,}5 \end{cases}$$

Entonces, los hijos se calculan como:

$$h_1[i] = 0{,}5 \cdot ((1+\beta) \cdot p_1[i] + (1-\beta) \cdot p_2[i])$$
$$h_2[i] = 0{,}5 \cdot ((1-\beta) \cdot p_1[i] + (1+\beta) \cdot p_2[i])$$

3.7. Uniform Order Based

Este operador está diseñado para cromosomas con permutaciones (como en el problema del viajante). Se genera una máscara binaria M de tamaño T. Las posiciones con 1 se copian directamente del primer padre al hijo. Las posiciones con 0 se rellenan, en orden, con los genes del segundo padre que no estén ya presentes en el hijo.

Este método mantiene la posición relativa y evita duplicados.

3.8. Order Based

También usado en permutaciones. Se seleccionan k posiciones aleatorias del primer padre y se copian al hijo. Luego, se rellena el resto del hijo con los genes del segundo padre manteniendo el orden relativo y omitiendo duplicados.

Este operador es útil cuando el orden de los elementos en el cromosoma es relevante, como en rutas o secuencias.

Operador de Cruza	**Espacio que abarca**	**Representación**
One Point	Local	Binaria
Two Point	Local	Binaria
Uniform	Moderado	Binaria
Arithmetic	Local	Real
Blend (BLX-α)	Global (controlado por α)	Real
Simulated Binary (SBX)	Global (controlado por η)	Real
Uniform Order Based	Moderado	Permutacional
Order Based	Moderado	Permutacional

Tabla 3.1: Comparativa de operadores de cruza según el tipo de representación y cobertura del espacio de búsqueda.

Capítulo 4

Operador de Mutación

Una mutación es un cambio en el cromosoma que altera su valor; esta se aplica de forma independiente en cada individuo bajo cierta probabilidad p. La inclusión de mutaciones permite mantener la diversidad genética de la población y evita el estancamiento en óptimos locales cuando hay demasiada similitud entre individuos.

Dado que cada problema puede tener un espacio diferente (binario, reales, permutaciones), cada uno debe implementar una función que permita obtener un vecino aleatorio. Un vecino es una solución cercana a la actual, pero con una ligera variación. En este caso, la función *Single Point* obtiene un vecino dada una solución actual, respetando la estructura del espacio de búsqueda.

4.1. Single Point

Este operador evalúa, para cada individuo en la población, si debe aplicarse una mutación basada en la probabilidad establecida. Si la condición se cumple, el individuo es reemplazado por un vecino generado por la función específica del problema; de lo contrario, permanece igual.

Este algoritmo permite mantener diversidad en la población sin alterar drásticamente la estructura de los individuos. Ademas, de que permite adaptarse a diferentes representaciones (binaria, real, permutaciones), aunque es fuertemente dependiente de la calidad de la función de *getRandomNeighbour*.

Algorithm 5 Single Point Mutation
Input { population, problem, mutation_rate }

```
1:  function SinglePoint(population, problem, mutation_rate)
2:      mutations ← [ ]
3:      for individual in population do
4:          if random() ≤ mutation_rate then
5:              neighbor ← problem.getRandomNeighbour(individual)
6:              mutations.append(neighbor)
7:          else
8:              mutations.append(individual)
9:          end if
10:     end for
11:     return mutations
12: end function
```

Capítulo 5

Operador de Reemplazo

Este último operador se encarga de reintegrar la nueva población generada (producto de la selección, cruza y mutación) con la población actual. Su función es decidir quiénes sobreviven a la siguiente generación, balanceando la exploración del espacio de búsqueda con la preservación de soluciones prometedoras.

5.1. Random

Este enfoque selecciona aleatoriamente un subconjunto de la población actual para conservarlo y luego lo complementa con los nuevos individuos generados. No se considera la aptitud, lo que favorece la diversidad pero puede generar regresión en la calidad.

Algorithm 6 Random Replacement
Input { population, replace }

```
1: function RandomReplace(population, replace)
2:     n ← len(population)
3:     k ← len(replace)
4:     survivors ← random.sample(population, n - k)
5:     survivors ← survivors + replace
6:     return survivors
7: end function
```

Este algoritmo favorece la diversidad pero no garantiza la preservación de buenos individuos, porl o que puede perder soluciones óptimas.

5.2. Elitism

Este método conserva a los mejores individuos de la población actual, asegurando que las soluciones de alta calidad no se pierdan entre generaciones. Luego se complementa la población con los nuevos individuos generados.

Algorithm 7 Elitism Replacement
Input { population, replace, objective }

```
1: function ElitismReplace(population, replace, objective)
2:     n ← len(population)
3:     k ← len(replace)
4:     sorted_pop ← sorted(population, key=objective)
5:     survivors ← sorted_pop[: n − k]
6:     survivors ← survivors + replace
7:     return survivors
8: end function
```

A diferencia de *random*, este si preserva los mejores individuos y acelera la convergencia hacia soluciones óptimas pero con riesgo de perder diversidad genética e incluso llevar a estancarlo en óptimos locales si se descuidan otros aspectos.

5.3. Deterministic Crowding

Este método busca mantener la diversidad genética mediante una competencia local entre padres e hijos similares. Cada hijo compite directamente con su padre más parecido (según una medida de distancia), y sobrevive el de mejor aptitud.

Algorithm 8 Deterministic Crowding Replacement
Input { parents, offspring, objective, distance }

```
1: function DeterministicCrowding(parents, offspring, objective, distance)
2:     survivors ← [ ]
3:     for i = 0 to len(parents) step 2 do
4:         p1, p2 ← parents[i], parents[i+1]
5:         o1, o2 ← offspring[i], offspring[i+1]
6:         if distance(p1, o1) + distance(p2, o2) ¡distance(p1, o2) + distance(p2, o1) then
7:             winners ← [ argmax(p1, o1), argmax(p2, o2) ]
8:         else
9:             winners ← [ argmax(p1, o2), argmax(p2, o1) ]
10:        end if
11:        survivors.extend(winners)
12:    end for
13:    return survivors
14: end function
```

Mantiene la diversidad poblacional y favorece la exploración de nichos a cambio de poder computacional.

5.4. Restricted Tournament Selection (RTS)

Este método también promueve la diversidad al restringir las competencias a individuos similares. Cada hijo compite contra un subconjunto aleatorio de la población, y se reemplaza al más parecido si el hijo tiene mejor aptitud.

Algorithm 9 Restricted Tournament Selection
Input { population, offspring, objective, window_size, distance }

```
1: function RTS(population, offspring, objective, window_size, distance)
2:     for child in offspring do
3:         window ← random.sample(population, window_size)
4:         closest ← argmin([distance(child, w) for w in window])
5:         if objective(child) ¿objective(closest) then
6:             population[population.index(closest)] ← child
7:         end if
8:     end for
9:     return population
10: end function
```

Mantiene estructuras locales de la población pero depende del calculo de distancias, lo que puede ser costoso.

Capítulo 6

Problemas

6.1. 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).

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

6.2. 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 $[1, n]$:

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

6.3. 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:

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

6.4. 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.

6.4.1. 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 6.1

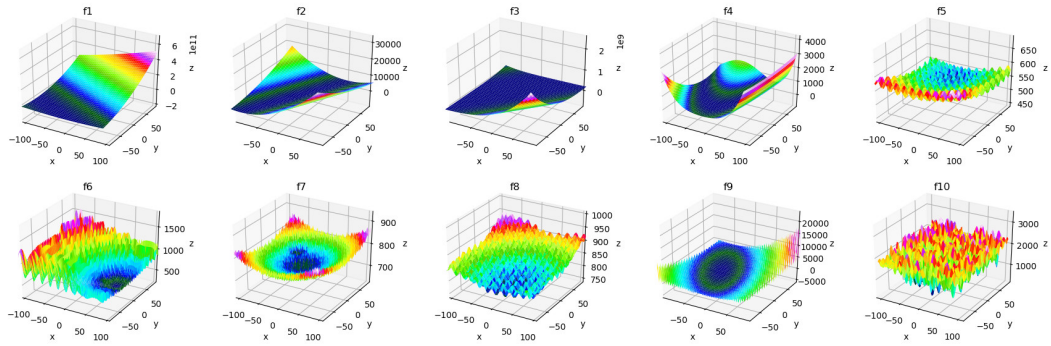

Figura 6.1: Superficies ploteadas de las 10 primeras funciones para dos dimensiones [**?**]

Capítulo 7

Codigo

7.1. Problem

Knapsack problem

La función 7.1 genera de manera aleatoria un conjunto de elementos en la mochila con pesos y valores aleatorios en un rango de $[1, 10]$. La función 7.2 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 7.3 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 7.4 se encarga de generar un vecino de forma aleatoria, primero selecciona un elemento aleatorio del vector y después lo invierte.

Listing 7.1: 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 7.2: 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:
```

```
    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 7.3: 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 7.4: 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 7.1 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.

7.1.1. Travel Salesman Problem

La función 7.5 genera una matriz cuadrada y simétrica de n valores aleatorios en un rango de $[1, 100]$.

La función 7.6 calcula la energía del sistema. Dado un vector suma todas las distancias de las ciudades con base en la información generada en 7.5, el resultado que devuelve es negativo ya que se busca minimizar. La función 7.7 genera un vector de n números consecutivos (representando las ciudades) y cambia las posiciones mediante la función *shuffle*.

Finalmente, la función 7.8 toma dos indices aleatorios diferentes e invierte los valores de dichas posiciones del vector.

Listing 7.5: Función *generate_information* de Travel Salesman Problem

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

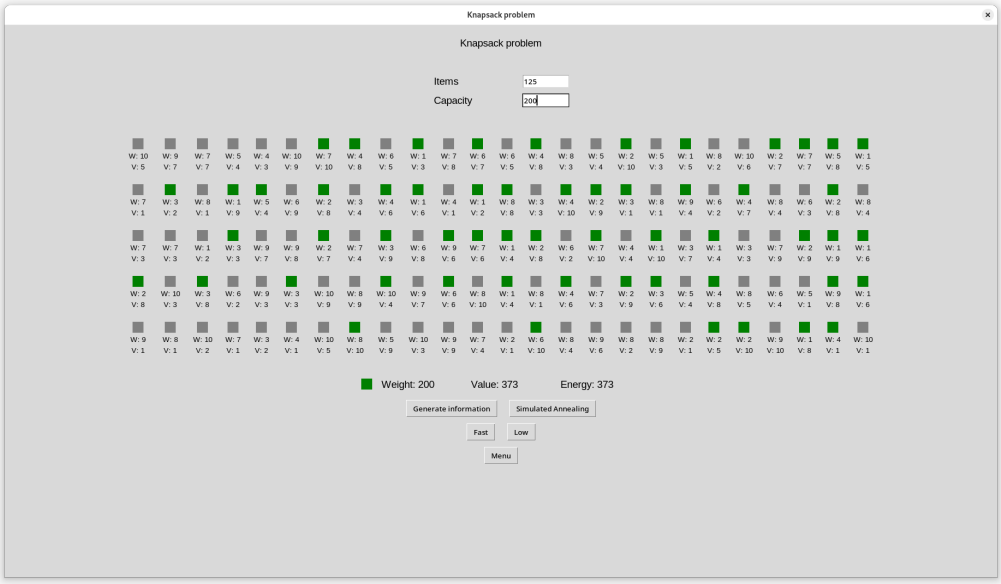

Figura 7.1: Knapsack Problem Interface

```
3
4       for i in range(cities):
5       for j in range(i, cities):
6       if i == j:
7       valor = 0
8       else:
9       valor = random.randint(1, 100)
10      distances[i][j] = valor
11      distances[j][i] = valor
12
13      self.information = {
14          "cities" : cities,
15          "distances" : distances
16      }
```

Listing 7.6: Función *energy* de Travel Salesman Problem

```
1       def energy(self, solution, information):
2       distance = 0
3       num_cities:int = len(solution)
4
5       for i in range(num_cities):
6       current_city = solution[i]
7       next_city = solution[(i + 1) % num_cities]
```

```
8       distance += information['distances'][current_city][
            next_city]
9
10      return -distance
```

Listing 7.7: Función *generate_initial_solution* de Travel Salesman Problem

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

Listing 7.8: Función *random_neighbour* de Travel Salesman Problem

```
1       def random_neighbour(self, solution):
2       neighbour = solution[:]
3       i = j = random.randint(0, len(solution) - 1)
4       while j == i:
5       j = random.randint(0, len(solution) - 1)
6       neighbour[i], neighbour[j] = neighbour[j], neighbour[
            i]
7
8       return neighbour
```

La interfaz de la figura 7.2 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.

7.1.2. Sum function Problem

La función 7.9 unicamente define el tamaño del vector y los rangos de valores. por otro lado, la función 7.10 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 7.11 genera un vector de n elementos aleatorios en los rangos definidos, mientras que la función 7.12 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 7.9: Función *generate_information* de SumFunctionProblem

```
1       def generate_information(self, size, min, max):
2       self.information = {
3           "size": size,
4           "min": min,
5           "max": max
6       }
```

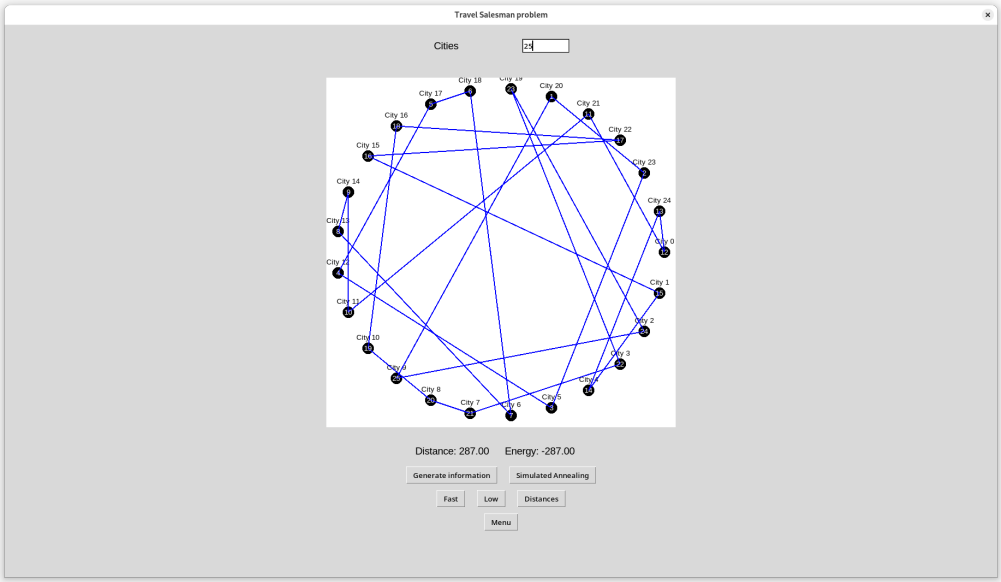

Figura 7.2: Travel Salesman Problem Interface

Listing 7.10: Función *energy* de SumFunctionProblem

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

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

    return -total_sum
```

Listing 7.11: 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 7.12: 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

```

```
7       if neighbour[index] > self.information['max']:
8       neighbour[index] = self.information['min']
9
10      if neighbour[index] < self.information['min']:
11      neighbour[index] = self.information['max']
12
13      return neighbour
```

La interfaz de la figura 7.3 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.

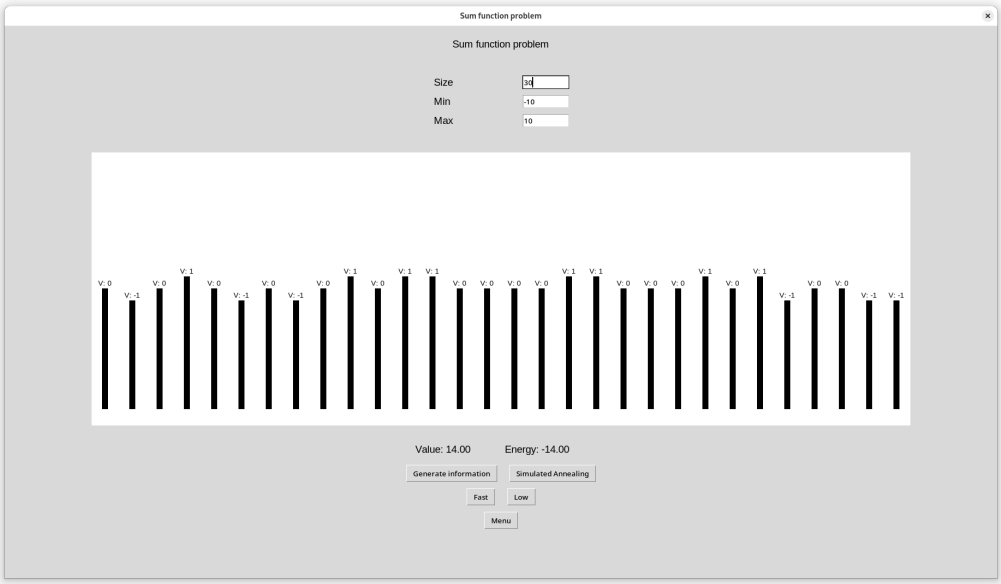

Figura 7.3: Sum Function Problem Interface

7.1.3. CEC 2017

7.2. Metaheuristic

7.2.1. Genetic Algorithm

7.2.2. Selection functions

7.2.3. Crossover functions

7.2.4. Mutation functions

7.2.5. Replace functions

Listing 7.13: Constructor de la clase SimulatedAnnealing

Capítulo 8

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 8.1 y la tabla 8.2.

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 8.1 tienen signo negativo.

La tabla 8.2 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 8.1: 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 8.2: Estadísticas de tiempo de ejecución por función.

Conclusión

Bibliografía