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

Ingeniería en Inteligencia Artificial, Metaheuristicas
Fecha: 21 de mayo de 2025

Algoritmos genéticos

Presenta

Angeles López Erick Jesse[1]

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

Resumen En esta práctica se describe el comportamiento, las partes esenciales, configuraciones, implementación y comparación de resultados de los algoritmos genéticos, como un algoritmo bioinspirado utilizado para la búsqueda de óptimos globales en problemas de optimización.

Palabras clave: Algoritmo genético, Evolución, Operador, Resultado óptimo.

[1] eangelesl1700@alumno.ipn.mx

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 modificaciones en el genoma. Estos cambios provocan individuos con rasgos alterados que afectan su supervivencia. Los supervivientes se reproducen y transmiten estos genes alterados. Por otro lado, los cambios que atentan contra la supervivencia de un individuo impiden su reproducción [1].

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 como factor determinante del nivel de supervivencia de cada individuo.

La evolución afecta a los cromosomas, que son estructuras encargadas de transportar la información genómica de una célula a otra. Es mediante la reproducción que 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 la cual los más aptos heredan sus genes a las nuevas generaciones para obtener mejores resultados. Este proceso consta de los siguientes pasos:

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

1.2.1. Ventajas

- Pueden explorar el espacio de soluciones en múltiples direcciones simultáneamente.
- Realizan una exploración amplia, lo que permite escapar de óptimos locales para alcanzar óptimos globales.
- Funcionan bien en problemas complejos y dinámicos, así como en aquellos donde la función objetivo es discontinua, ruidosa o presenta múltiples óptimos locales. También permiten optimización multiobjetivo.
- Cada parte del proceso puede ser implementada con funciones específicas para mejorar el rendimiento del algoritmo; por ejemplo, existen funciones enfocadas únicamente en la exploración.

1.2.2. Desventajas

- La función objetivo es muy sensible para los algoritmos genéticos; si se define incorrectamente, puede impedir encontrar una solución adecuada al problema.
- La elección de los parámetros (tamaño de la población, tasa de cruzamiento, selección de padres, tasa de mutación, entre otros) puede ser compleja. Una mala configuración puede provocar un bajo rendimiento.
- Requieren un alto tiempo de ejecución y potencia de cómputo.
- Existe el riesgo de convergencia prematura, es decir, que un individuo se reproduzca excesivamente y reduzca la diversidad genética, lo cual puede llevar a un óptimo local no deseado.

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) está construido a partir del fenotipo y representa la codificación de sus características, generalmente como un vector.

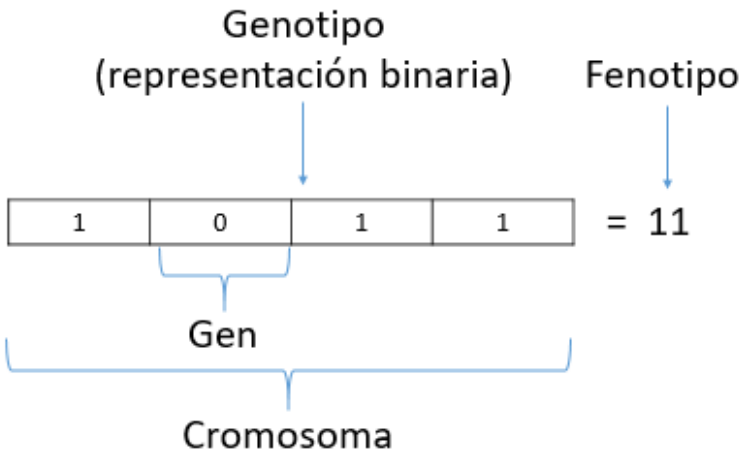

Figura 1.1: Descripción gráfica de genotipo, fenotipo, cromosoma y gen del número “11”

1.2.4. Modelos generacionales y estacionarios

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

Los **modelos generacionales** actualizan la mayor parte de la población. Buscan una evolución rápida y una amplia exploración. Para ello, se selecciona un pequeño conjunto de individuos y se les aplican los algoritmos de selección, cruza y mutación. Finalmente, mediante un algoritmo de reemplazo, se reintegran en la población mediante una función definida.

Por otro lado, los **modelos estacionarios** buscan mantener el equilibrio en la mayor parte de la población, experimentando y mezclando únicamente un pequeño subconjunto. El cambio es más lento, pero más 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 la siguiente generación.

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

- `population`: Subconjunto de la población al 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 Random*) 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)
$u \leftarrow$ 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 u 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]
    eliminar p_i de la población temporal para evitar emparejarlo 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 $t = 3$.

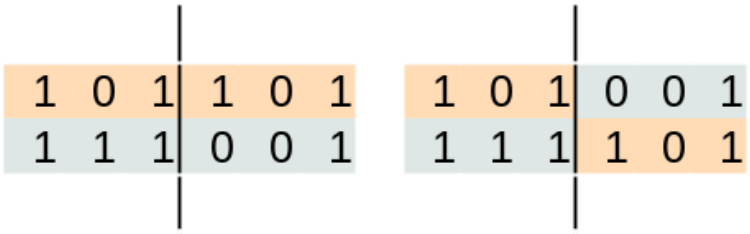

Figura 3.1: Ejemplo de cruza en un solo punto

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:

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

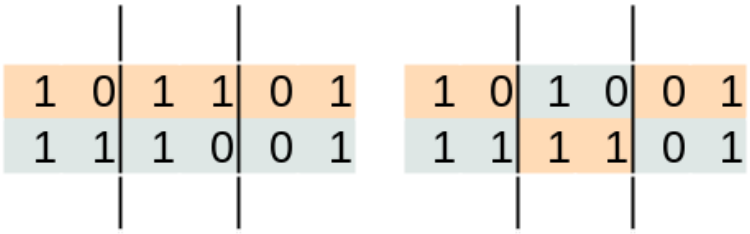

Figura 3.2: Ejemplo de cruza en dos puntos

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 $h_2[i] = p_1[i]$.

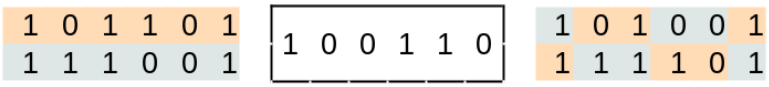

Figura 3.3: Ejemplo de cruza uniforme

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 representa la 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 a 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 de búsqueda distinto (binario, real, permutacional), es necesario que cada uno implemente una función que permita obtener un vecino aleatorio. Un vecino es una solución cercana a la actual, pero con una ligera variación. En este contexto, la función *Single Point* genera un vecino a partir de una solución dada, respetando la estructura del espacio correspondiente.

4.1. Single Point

Este operador evalúa, para cada individuo en la población, si debe aplicarse una mutación con base 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 sin cambios.

Este enfoque permite mantener la diversidad en la población sin alterar drásticamente la estructura de los individuos. Además, se adapta a diferentes tipos de representación (binaria, real o permutacional), aunque depende fuertemente de la calidad de la función *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 una 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, por lo que puede perder soluciones óptimas.

5.2. Elitismo

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 sí preserva los mejores individuos y acelera la convergencia hacia soluciones óptimas, aunque con el riesgo de perder diversidad genética e incluso de estancarse 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
```

Este enfoque mantiene la diversidad poblacional y favorece la exploración de nichos, a cambio de un mayor costo 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
```

Este método mantiene estructuras locales dentro de la población, aunque depende del cálculo de distancias, lo cual puede resultar costoso computacionalmente.

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 [4] 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

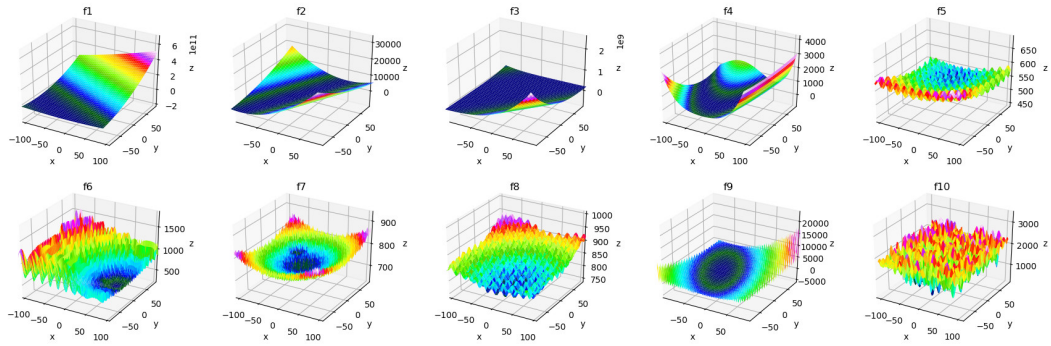

Figura 6.1: Superficies ploteadas de las 10 primeras funciones para dos dimensiones [5]

Capítulo 7

Codigo

7.1. Problem

Los problemas a optimizar siguen una mista estructura. Dado esto, se optó por diseñar una clase que generaliza el problema y permite y facilita la definición de nuevos problemas.

El código 7.1 incluye los siguientes métodos los cuales deben de ser implementados por sus clases hijas:

- *generateInformation*: Función que define la información necesaria para calcular la función objetivo. La información adicional depende del problema.
- *objective*: Función que calcula la utilidad de la solución, si es un problema de minimización tiene que devolver con el signo opuesto.
- *generateInitialSolution*: Función que genera una solución.
- *getRandomNeighbour*: Obtiene un vecino de forma aleatoria.
- *getNextNeighbour*: Si es necesario manejar indices, esta función permite obtener un vecino tras otro.
- *getNeighbours*: Devuelve todos los vecinos.

Listing 7.1: Clase *problem*

```
1 class Problem(ABC):
2   def __init__(self,):
3   self.information:Any = {}
4
5   @abstractmethod
```

```
6     def generateInformation(self, *args, **kwargs) -> None:
          pass
7
8     @abstractmethod
9     def objective(self, solution: Any) -> float: pass
10
11    @abstractmethod
12    def generateInitialSolution(self) -> Any: pass
13
14    @abstractmethod
15    def getRandomNeighbour(self, solution:Any) -> Any: pass
16
17    @abstractmethod
18    def getNextNeighbour(self, solution:Any, *args, **
          kwargs) -> Any: pass
19
20    @abstractmethod
21    def getNeighbours(self, solution: Any) -> list: pass
22
23    @abstractmethod
24    def printInformation(self) -> None: pass
```

Knapsack problem

La clase *knapsackProblem* hereda de la clase *Problem* e implementa las siguientes funciones.

La función 7.2 genera de manera aleatoria un conjunto de elementos en la mochila con pesos y valores aleatorios en un rango de $[1, 10]$. La función 7.3 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 7.4 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 7.5 se encarga de generar un vecino de forma aleatoria, primero selecciona un elemento aleatorio del vector y después lo invierte.

Listing 7.2: Función *generateInformation* 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)],
```

```
    "capacity": capacity
  }
```

Listing 7.3: Función *objective* de Knapsack problem

```
def objective(self, solution):
  total_weight = total_value = 0
  for i, selected in enumerate(solution):
    if selected:
      total_weight += self.information["values"][i][0]
      total_value += self.information["values"][i][1]
  if total_weight > self.information["capacity"]:
    return self.information["capacity"] - total_weight
  return total_value
```

Listing 7.4: Función *generateInitialSolution* de Knapsack problem

```
def generateInitialSolution(self):
  return [random.randint(0, 1) for _ in self.information[
     'values']]
```

Listing 7.5: Función *getRandomNeighbour* de Knapsack problem

```
def getRandomNeighbour(self, solution):
  neighbour = solution[:]
  index = random.randint(0, len(solution) - 1)
  neighbour[index] = 1 - int(neighbour[index])
  return neighbour
```

Para el algoritmo genético, no es necesario especificar el resto de funciones.

7.1.1. Travel Salesman Problem

La función 7.6 genera una matriz cuadrada y simétrica de n valores aleatorios en un rango de $[1, 100]$.

La función 7.7 calcula la energía del sistema. Dado un vector suma todas las distancias de las ciudades con base en la información generada en 7.6, el resultado que devuelve es negativo ya que se busca minimizar. La función 7.8 genera un vector de n números consecutivos (representando las ciudades) y cambia las posiciones mediante la función *shuffle*.

Finalmente, la función 7.9 toma dos indices aleatorios diferentes e invierte los valores de dichas posiciones del vector.

Listing 7.6: Función *generateInformation* de Travel Salesman Problem

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

```
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 7.7: Función *Objective* de Travel Salesman Problem

```
1  def objective(self, solution):
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 += self.information['distances'][
          current_city][next_city]
9
10   return -distance
```

Listing 7.8: Función *generateInitialSolution* de Travel Salesman Problem

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

Listing 7.9: Función *getRandomNeighbour* de Travel Salesman Problem

```
1  def getRandomNeighbour(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
```

7.1.2. Sum function Problem

La función 7.10 unicamente define el tamaño del vector y los rangos de valores. por otro lado, la función 7.11 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 7.12 genera un vector de n elementos aleatorios en los rangos definidos, mientras que la función 7.13 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 7.10: Función *generateInformation* de SumFunctionProblem

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

Listing 7.11: Función *objective* de SumFunctionProblem

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

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

  return -total_sum
```

Listing 7.12: Función *generateInitialSolution* de SumFunctionProblem

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

Listing 7.13: Función *getRandomNeighbour* de CEC 2017

```
def getRandomNeighbour(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
```

7.1.3. CEC 2017

La función 7.14 define la función a optimizar, los rangos de valores, el tamaño de la dimensión y la tasa de cambio (ya que la solución es real). Por otro lado, la función 7.15 calcula la energía del sistema dados los parámetros ya definidos.

La función 7.16 genera un vector de la dimensión definida restringida por la información adicional definida, mientras que la función 7.17 suma o resta en *alpha* a un elemento aleatorio del vector.

Listing 7.14: Función *generateInformation* de CEC 2017

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

Listing 7.15: Función *objective* de CEC 2017

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

Listing 7.16: Función *generateInitialSolution* de CEC 2017

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

Listing 7.17: Función *getRandomNeighbour* de CEC 2017

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

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

7.2. Metaheuristic

De igual manera, el se diseña una clase padre 7.18 que implemente algunas de las funciones compatibles entre algoritmo genético, *hill climbing* y *simulated annealing*.

Esta clase tiene como parametro la clase *Problem* y define los métodos abstractos *resetProblem* y *optimize* los cuales reinician meta parámetros y evalúan el algoritmo, respectivamente.

Adicionalmente se definen 3 funciones:

- *evaluate*: recibe como parámetro una solución y devuelve la aptitud definida por el problema
- *isBetterSolution*: Compara dos soluciones.
- *isSameSolution*: Verifica si dos soluciones tiene la misma aptitud.

Listing 7.18: Clase *Metaheuristic*

```
class Metaheuristic(ABC):
  def __init__(self, problem: Problem):
  self.problem = problem

  @abstractmethod
  def resetProblem(self):
    self.solution = self.problem.generateInitialSolution
       ()
    self.is_best = False

  @abstractmethod
  def optimize(self, *args, **kwargs): pass

  def evaluate(self, state: Any) -> float:
    return self.problem.objective(state)
```

```
15
16    def isBetterSolution(self, solution_1: Any, solution_2:
          Any)-> bool:
17      return (self.problem.objective(solution_1) >
18          self.problem.objective(solution_2))
19
20    def isSameSolution(self, solution_1: Any, solution_2:
         Any) -> bool:
21    return (self.problem.objective(solution_1) ==
22        self.problem.objective(solution_2))
23
24    def printSolution(self) -> None:
25      print(f'Solution: {self.solution}: {self.evaluate(
           self.solution)}')
26
27    def getSolution(self) -> Any:
28      return self.solution
29
30    def setSolution(self, solution: Any):
31      self.solution = solution
```

7.2.1. Genetic Algorithm

La clase *GeneticAlgorithm* hereda de la clase *Metaheuristic* e incluye las clases *GASelectionFunctions*, *GACrossoverFunctions*, *GAMutationFunctions* y *GAReplaceFunctions* que se verán mas adelante.

El constructor del código 7.19 recibe como parametros el problema de optimización y el tamaño de la población. Adicionalmente crea los objetos de las funciones para poder ser accedidas desde esa misma clase y llama la función 7.20, la cual predefine las funciones del algoritmo e inicializa la población.

La función 7.21 se encarga de encontrar la mejor solución realizando las operaciones de selección, cruza, mutación y reemplazo. Nótese que esta función recibe como parámetro *stationary* de tipo *float* el cual define el tamaño de la población que se mantiene estacionaria y cual evoluciona. A la parte que evoluciona se le aplican las operaciones de selección, cruza y mutación para al final aplicar la operación de reemplazo a la población en general.

Adicionalmente, la función 7.22 es el *setter* de la función a utilizar, el resto de funciones tienen la misma estructura. Se utiliza *partial* ya que existen funciones que requieren de parámetros adicionales.

Listing 7.19: Constructor *GeneticAlgorithm*

```
1     class Metaheuristic(ABC):
2 def __init__(self, problem, population_size: int = 16):
```

```
3     super().__init__(problem)
4     self.population_size = population_size
5     self.selection_functions = Selection
6     self.crossover_functions = Crossover
7     self.mutation_functions = Mutation
8     self.replace_functions = Replace
9     self.resetProblem()
```

Listing 7.20: Función *resetProblem*

```
1 def resetProblem(self):
2 super().resetProblem()
3 self.population = [self.problem.generateInitialSolution()
      for _ in range(self.population_size)]
4 self.selection = Selection.tournament
5 self.crossover = Crossover.onePoint
6 self.mutation = Mutation.singlePoint
7 self.replace = Replace.random
```

Listing 7.21: Función *optimize*

```
1 def optimize(self, epochs: int = 1, stationary: float =
     0):
2   for _ in range(epochs):
3     population_sample = random.sample(self.population,
        self.population_size - int(self.population_size *
        stationary))
4
5     population_sample = self.selection(population_sample,
         self.problem.objective)
6     population_sample = self.crossover(population_sample)
7     population_sample = self.mutation(population_sample,
        self.problem)
8     self.population = self.replace(self.population,
        population_sample, self.problem.objective)
```

Listing 7.22: Función *optimize*

```
1 def setSelection(self, selection, selection_rate = None):
2   if selection_rate:
3     self.selection = partial(selection, selection_rate =
        selection_rate)
4   else:
5     self.selection = selection
```

7.2.2. Selection functions

Las funciones de selección reciben como parametros la población de la cual se seleccionan los padres, la función objetivo y retorna el conjunto de individuos de padres de la siguiente generación.

A conitnuación se muestran tres operadores de selección:

- *tournament*. Recibe un parámetro adicional que define la cantidad de individuos que participaran en cada torneo.
- *proportional*
- *negativeAssortativeMating*. Aunque recibe como parámetro la función objetivo no es requerida, ya que es at función busca emparejar aquellos individuos mas diferentes.

Listing 7.23: Función *tournament*

```
@staticmethod
def tournament(population, objective, selection_rate=0.2)
    :
  population_size = len(population)
  parents = []

  for _ in range(population_size):
    k = max(2, math.ceil(population_size * selection_rate
       ))
    candidates = random.sample(population, min(k,
       population_size))
    scores = [objective(ind) for ind in candidates]
    best = candidates[scores.index(max(scores))]
    parents.append(best)

  return parents
```

Listing 7.24: Función *proportional*

```
@staticmethod
def proportional(population, objective):
  population_size = len(population)
  parents = []

  objectives = SelectionFunctions._linealDisplacement(
     population, objective)

  total = sum(objectives)
```

```
9   probabilities = [ objective/total for objective in
        objectives ]
10
11  proportions = [ probabilities[0] ]
12  for i in range(1, len(probabilities)):
13    proportions.append(probabilities[i] + proportions[i -
          1])
14
15  for _ in range(population_size):
16    pos = random.random()
17    for i in range(len(proportions)):
18      if (pos <= proportions[i]):
19        parents.append(population[i])
20        break
21
22  return parents
```

Listing 7.25: Función *negativeAssortativeMating*

```
1  @staticmethod
2  def negativeAssortativeMating(population, objective):
3    population_size = len(population)
4    parents = []
5
6    for _ in range(int(population_size/2)):
7      individual = random.choice(population)
8      distances = [ SelectionFunctions.distance(individual,
           test) for test in population ]
9
10     parents.append(individual)
11     parents.append(population[distances.index(max(
         distances))])
12
13   return parents
```

7.2.3. Crossover functions

A continuación, se muestra la implementación de tres algoritmos de cruza:

- *onePoint* para soluciones binarias.
- *arithmetic* para soluciones reales.
- *uniformOrderBased* para soluciones de permutación.

Listing 7.26: Función *onePoint*

```
def onePoint(population:list):
  generation:list = []
  population_size = len(population)
  for i in range(0, population_size, 2):
    parent1 = population[i]
    parent2 = population[i + 1]

    point:int = random.randint(1, population_size - 1)

    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    generation.extend([child1, child2])

  return generation
```

Listing 7.27: Función *arithmetic*

```
@staticmethod
def arithmetic(population: list, crossover_rate: float =
   None):
  generation = []
  population_size = len(population)

  for i in range(0, population_size - 1, 2):
    parent1 = population[i]
    parent2 = population[i + 1]

    a = crossover_rate if crossover_rate is not None else
        random.random()

    child1 = [a * x + (1 - a) * y for x, y in zip(parent1
       , parent2)]
    child2 = [(1 - a) * x + a * y for x, y in zip(parent1
       , parent2)]

    generation.extend([child1, child2])

  return generation
```

Listing 7.28: Función *uniformOrderBased*

```
@staticmethod
def uniformOrderBased(population: list):
  generation = []
  population_size = len(population)
```

```
5
6   for i in range(0, population_size - 1, 2):
7     parent1 = population[i]
8     parent2 = population[i + 1]
9     size = len(parent1)
10
11    mask = [random.randint(0, 1) for _ in range(size)]
12
13    def create_child(p1, p2):
14      child = [None] * size
15      for j in range(size):
16        if mask[j] == 1:
17          child[j] = p1[j]
18      fill = [gene for gene in p2 if gene not in child]
19      k = 0
20      for j in range(size):
21        if child[j] is None:
22          child[j] = fill[k]
23          k += 1
24      return child
25
26    child1 = create_child(parent1, parent2)
27    child2 = create_child(parent2, parent1)
28    generation.extend([child1, child2])
29
30  return generation
```

7.2.4. Mutation functions

Se utilizo unicamente una función de mutación que busca un vecino inmediato de la solución que se le otorgue dada una probabilidad.

Listing 7.29: Función *arithmetic*

```
1 @staticmethod
2 def singlePoint(population:list, problem, mutation_rate:
     float = 0.2):
3   mutations: list  = []
4   for individual in population:
5     if random.random() <= mutation_rate:
6       mutations.append(problem.getRandomNeighbour(
          individual))
7     else:
8       mutations.append(individual)
9
```

```
10    return mutations
```

7.2.5. Replace functions

Listing 7.30: Función *elitism*

```
1  @staticmethod
2  def elitism(population: list, replace: list, objective):
3    population_size = len(population)
4    replace_size = len(replace)
5
6    # Ordenar poblacion por fitness
7    sorted_population = sorted(population, key=objective,
       reverse=True)
8
9    # Conservar los mejores
10   survivors = sorted_population[:population_size -
       replace_size]
11   survivors += replace
12
13   return survivors
```

Listing 7.31: Función *determinstCrowding*

```
1  @staticmethod
2  def determinstCrowding(population: list, replace: list,
     objective):
3    new_population = []
4    for i in range(0, len(population), 2):
5      p1, p2 = population[i], population[i + 1]
6      o1, o2 = replace[i], replace[i + 1]
7
8      if distance(p1, o1) + distance(p2, o2) <= distance(p1
         , o2) + distance(p2, o1):
9        winner1 = o1 if objective(o1) > objective(p1) else
           p1
10       winner2 = o2 if objective(o2) > objective(p2) else
           p2
11     else:
12       winner1 = o2 if objective(o2) > objective(p1) else
           p1
13       winner2 = o1 if objective(o1) > objective(p2) else
           p2
14
15     new_population.extend([winner1, winner2])
```

```
16
17    return new_population
```

Listing 7.32: Función *restrictedTournament*

```
@staticmethod
def restrictedTournament(population: list, replace: list,
    objective, selection_rate: int = 5):
  survivors = population.copy()

  for child in replace:
    window = random.sample(survivors, min(selection_rate,
        len(survivors)))

    closest = min(window, key=lambda p: distance(p, child
       ))
    if objective(child) > objective(closest):
      survivors[survivors.index(closest)] = child

return survivors
```

7.3. Pruebas de aptitud

El código a continuación se encarga de realizar pruebas para cada cada combinación de los diferentes de los diferentes algoritmos y para diferentes porcentajes de población estacionaria. Se obtienen los valores de mejor y peor valor, media de función objetivo y desviación estándar de los valores obtenidos. Esto con el objetivo de comprar los resultados y determinar las condiciones para la optimización de un problema.

Listing 7.33: Código de pruebas

```
for problem_name, problem in problems.items():
g = GeneticAlgorithm(problem)

for sel_name, sel_func in selection.items():
 for cross_name, cross_func in crossover_set.items():
   for mut_name, mut_func in mutation.items():
     for rep_name, rep_func in replace.items():
       scores = []
       for stationary in np.arange(0, 1.01, 0.1):

         for _ in range(iterations):
          g.resetProblem()
          g.setSelection(sel_func(g))
```

```
            g.setCrossover(cross_func(g))
            g.setMutation(mut_func(g))
            g.setReplace(rep_func(g))
            g.optimize(epochs, stationary)
            score = g.bestIndividual()
            scores.append(score)

        best = max(scores)
        worst = min(scores)
        mean = statistics.mean(scores)
        stddev = statistics.stdev(scores) if len(scores) >
            1 else 0.0

      writer.writerow([
        problem_name, round(stationary,1), sel_name,
            cross_name, mut_name, rep_name,
        best, worst, mean, stddev
      ])
```

Capítulo 8

Resultados

Conclusión

Bibliografía

[1] National Human Genome Research Institute. (2025) Evolución. Accedido el 19 de mayo de 2025. [Online]. Available: https://www.genome.gov/es/genetics-glossary/Evolucion

[2] SCI2S - Universidad de Granada. (2025) Welcome to the sci2s web site — soft computing and intelligent information systems. Accedido el 19 de mayo de 2025. [Online]. Available: https://sci2s.ugr.es/sites/default/files/files/Teaching/GraduatesCourses/Algoritmica/Tema07-AlgoritmosGeneticos-12-13.pdf

[3] National Human Genome Research Institute. (2025) Cromosoma. Accedido el 19 de mayo de 2025. [Online]. Available: https://www.genome.gov/es/genetics-glossary/Cromosoma

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

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