Universidad Nacional Autónoma de México
Escuela Nacional de Estudios Superiores
Unidad Morelia

Prácticas didácticas para el estudio y comprensión de metaheurísticas utilizadas en la resolución de problemas de optimización difíciles

Investigadores participantes:

Adriana Menchaca Méndez [1]
Saúl Zapotecas Martínez [2]
Elizabeth Montero Ureta [3]
Carlos A. Coello Coello [4]
Katya Rodríguez Vázquez [5]
Sergio Rogelio Tinoco Martínez [1]

Alumnos participantes:

Jesús Armando Ortíz Peñafiel [1]
Angélica Nayeli Rivas Bedolla[1]

Morelia, Michoacán, México. Marzo, 2021

[1]Escuela Nacional de Estudios Superiores, Unidad Morelia, UNAM, México
[2]Universidad Autónoma Metropolitana, Unidad Cuajimalpa, México
[3]Facultad de Ingeniería, Universidad Andrés Bello, Chile
[4]Centro de Investigación y de Estudios Avanzados del IPN, Unidad Zacatenco, México
[5]Instituto de Investigaciones en Matemáticas Aplicadas, UNAM, México

Trabajo realizado con el apoyo del Programa UNAM-DGAPA-PAPIME PE102320.

Agradecimientos

Agradecemos el apoyo de los alumnos y las alumnas de la Licenciantura en Tecnologías para la Información en Ciencias, de la ENES Unidad Morelia, por sus comentarios y ayuda en el mejoramiento de este material didáctico.

Resumen

En el área de computación existen varios problemas de optimización que son difíciles de resolver. Por ejemplo, problemas NP-difíciles y problemas para los cuales no funcionan las técnicas clásicas de programación matemática.

Una de las técnicas más utilizadas para resolver este tipo de problemas son las heurísticas. Estas técnicas nos permiten encontrar soluciones buenas al problema, sin garantizar encontrar la solución óptima, en un tiempo razonable. En varias escuelas de educación superior los alumnos cursan asignaturas que tienen como objetivo enseñarles técnicas metaheurísticas para resolver problemas de optimización difíciles. Una metaheurística es una heurística que nos permite resolver de una manera más general diferentes tipos de problemas. Sin embargo, estos temas tienen una complejidad elevada y es necesario que los alumnos desarrollen prácticas que les ayuden a comprender el funcionamiento de las metaheurísticas estudiadas y además desarrollen la habilidad de adaptarlas para resolver exitosamente diferentes tipos de problemas.

En este documento se presentan cinco metaheurísticas: 1) Búsqueda tabú, 2) recocido simulado, 3) programación evolutiva, 4) estrategias evolutivas y 5) algoritmos genéticos. Cada metaheurística es abordada en un capítulo que contiene la descripción de la misma, un ejemplo de cómo aplicarla en un problema de optimización difícil y ejercicios para que los alumnos las apliquen en diferentes problemas. Además, como material complementario se anexan las implementaciones, utilizando *Python*, de las metaheurísticas utilizadas en los ejemplos.

Índice general

Capítulo 1

Búsqueda Tabú

La **búsqueda tabú** (BT) es una metaheurística propuesta por Fred Glover [14, 15, 16]. Puede ser vista como un método de búsqueda local que tiene incorporada una memoria que le ayuda a mejorar la estrategia de búsqueda. Por ejemplo, permite escapar de óptimos locales y mejora los costos de cómputo al evitar caer en ciclos o zonas ya exploradas. La BT ha sido utilizada para resolver distintos problemas de optimización y se le han ido incorporando otros elementos [18, 17, 19, 13]. Por ejemplo, en optimización combinatoria se ha utilizado en el problema de asignación cuadrática, problema del clan máximo, problema de la mochila, problema del agente viajero, etc. En [18] presentan una lista extensa de trabajos donde han aplicado BT para resolver problemas relacionados con modelado de proteínas, redes de telecomunicación, diseño de estructuras electromagénticas, planificación de tareas en multiprocesadores, etc.

1.1. Metodología de búsqueda tabú

BT parte siempre de una solución x dentro del espacio de búsqueda S. Posteriormente, itera una serie de pasos hasta alcanzar un criterio de paro. Cada solución en el espacio de búsqueda tiene definido un vecindario $N(x) \subset S$. Un movimiento dentro de BT permite pasar de la solución actual a una solución dentro de su vecindario. A diferencia de un método de descenso, la BT puede permitir pasar a una solución que no mejore e incluso empeore el valor en la función objetivo. De esta forma, BT intenta salir de óptimos locales y aspira a encontrar un óptimo global.

BT define estructuras de memoria de corto y/o largo plazo que almacenan información sobre el proceso de búsqueda. Esta información se utiliza para redefinir el vecindario con el objetivo de mejorar la estrategia de búsqueda. Regularmente, se utiliza una **lista tabú** T en la que se almacenan soluciones o características de las soluciones que se desean evitar durante cierto tiempo del proceso de búsqueda. El tiempo de permanencia de una solución o característica en la lista tabú es conocido como **tiempo tabú**. De esta forma el nuevo vecindario se define como $N^*(x) = N(x) \setminus T$. BT también permite expandir el vecindario inicial, usualmente lo hace utilizando memorias de largo plazo: $N^*(x) = (N(x) \cup A) \setminus T$. Por ejemplo, A almacena las características de las mejores soluciones conocidas en el pasado e inserta soluciones en el vecindario que cuenten con algunas de estas características. Las memorias de largo plazo también pueden ser utilizadas para evitar caer en soluciones que ya se han explorado en el pasado.

Un elemento importante que debe considerarse cuando se diseña un algoritmo basado en BT es el balance entre el proceso de *intensificación* y el proceso de *diversificación*. El proceso de intensificación modifica la estrategia de búsqueda con la finalidad de obtener soluciones que históricamente han sido buenas. De manera contraria, el proceso de diversificación intenta generar soluciones que contengan características que nunca han aparecido en el proceso de búsqueda. El Algoritmo 1 muestra de manera muy general la forma en la que opera una BT simple.

Como se mencionó anteriormente, se han incorporado otros elementos a la BT. Un ejemplo, son los llamados **criterios de aspiración**, los cuales permiten que ciertos movimientos que son tabú puedan ser utilizados. El criterio de aspiración más simple consiste en permitir un movimiento tabú, si la solución que genera supera a la mejor solución conocida hasta ese momento. Existen otras propuestas de criterios de aspiración, ver por ejemplo: [5, 23, 20].

1.2. Problema de la mochila binario

(0/1 Knapsack problem)

Con la finalidad de ejemplificar la funcionalidad de BT utilizaremos el problema de la mochila binario (0/1 KP por sus siglas en inglés). 0/1 KP

Algoritmo 1: Búsqueda Tabú Simple

Entrada: Máximo número de iteraciones I_{max}, tiempo tabú t_{tabu}, parámetros de entrada para el problema que se quiere resolver.
Salida : Mejor solución encontrada.

1 Generar solución inicial x_0;
2 Actualizar mejor solución encontrada: $x_{best} \leftarrow x_0$, $f_{best} \leftarrow f(x_0)$;
3 Colocar el contador de iteraciones en 0: $k \leftarrow 0$;
4 Definir la solución actual: $x \leftarrow x_0$;
5 Iniciar la lista tabú como un conjunto vacío: $T = \emptyset$;
6 **while** $k < I_{max}$ **do**
7 Determinar el vecindario $N(x)$;
8 Determinar el vecindario reducido $N^*(x) = N(x) \setminus T$;
9 Escoger la mejor solución y entre las soluciones que están en $N^*(x)$;
10 $x \leftarrow y$;
11 **if** $f(x)$ *es mejor que* $f(x_{best})$ **then**
12 $x_{best} = x$;
13 $f_{best} = f(x)$;
14 **end**
15 $k \leftarrow k + 1$;
16 Actualizar la lista tabú T considerando el tiempo tabú t_{tabu};
17 **end**
18 Regresar x_{best}, f_{best};

considera un conjunto finito de objetos, donde cada objeto tiene un peso y un valor específicos, y una mochila que solo puede soportar cierto peso. El objetivo es encontrar el subconjunto de objetos que se pueden introducir en la mochila maximizando su valor. Formalmente se puede definir como sigue:

$$\begin{array}{rll} \text{maximizar:} & f(\vec{x}) = \sum_{i=1}^{n} p_i \cdot x_i & \\ \text{tal que:} & g_1(\vec{x}) = \sum_{i=1}^{n} w_i \cdot x_i \leq c & \\ & x_i \in \{0,1\} & i \in \{1,\ldots,n\} \end{array} \tag{1.1}$$

donde $x_i \in \{0,1\}$ (0 indica que el objeto i no está en la mochila y 1 que el objeto i está en la mochila) son las componentes del vector solución $\vec{x}$, p_i y w_i son el valor y el peso del objeto i respectivamente, n es el número de objetos y c es el peso que puede soportar la mochila. El conjunto de soluciones factibles que cumplen con g_1 es llamado *región factible* y se denota como Ω. A continuación se muestra un posible diseño de los elementos principales de una BT, considerando lo siguiente: $n = 5$, $p = \{5, 14, 7, 2, 23\}$, $w = \{2, 3, 7, 5, 10\}$ y $c = 15$.

Representación de una solución

Cada solución será vista como una cadena binaria de tamaño 5. Cada posición de la cadena es una variable del problema e indica con un 1 que el objeto i sí está en la mochila y con un 0 que no.

- 11001. Esta solución consiste en introducir los objetos 1, 2, y 5 a la mochila.

Solución inicial

Generamos una solución aleatoria factible, empezando con la cadena binaria 00000. Posteriormente, se eligen posiciones de la cadena aleatoriamente y se cambia su valor a 1 mientras no se exceda el peso permitido para la mochila.

Función objetivo y restricción

Para conocer el valor en la función objetivo, f, y de la restricción, g_1, de la solución 11001 hacemos el cálculo de la Ec. 1.1.

- $f(11001) = 5 \cdot 1 + 14 \cdot 1 + 7 \cdot 0 + 2 \cdot 0 + 23 \cdot 1 = 42$

- $g_1(11001) = 2 \cdot 1 + 3 \cdot 1 + 7 \cdot 0 + 5 \cdot 0 + 10 \cdot 1 = 15$

Dado que $g_1(11001) \leq 15$, 11001 es una solución factible.

Movimiento

Un movimiento es una transformación aplicada a una solución candidata. Altera los valores asignados de algunas variables.

- *Bitflip*. Cambia el valor de una variable: $11001 \rightarrow 11101$. En este caso la nueva solución mete el objeto 3 a la mochila.

Vecindario

El vecindario de una solución es un conjunto de soluciones que se pueden generar a partir de los movimientos definidos.

- N define para cada solución candidata $x \in \Omega$ un conjunto $N(x) \subseteq \Omega$.
- $N(x)$ son en algún sentido "cercanas" a x.
- Si aplicamos *Bitflip* a todas las variables de la solución $x = 11001$, podemos generar las siguientes soluciones:

  $$\{01001, 10001, 11101, 11011, 11000\}$$

  Si definimos nuestro vecindario como las soluciones factibles que se pueden generar a partir de movimientos *Bitflip* tenemos que:

  $$N(x) = \{01001, 10001, 11000\}$$

  Dado que $f(01001) = 37$, $f(10001) = 28$ and $f(11000) = 19$, nuestra siguiente solución sería 01001. El vecindario también puede contener soluciones infactibles pero se debe agregar un mecanismo que permita lidiar con ellas. Por ejemplo, funciones de penalización.

Lista tabú

En este caso almacenaremos características de soluciones que no se pueden permitir durante un lapso de tiempo. La representación de cada elemento de la lista tabú será (p, v, t), donde p indica la posición, v el valor que no se puede usar y t el número de iteraciones que aún estará prohibido. El valor de t (cuando la solución entra a la lista tabú) debe considerarse a la hora de realizar el diseño de la BT o bien definirlo como un parámetro de entrada. En este caso usaremos un **tiempo tabú** de $\lfloor \frac{n}{2} \rfloor$.

- Si pasé de la solución 11001 a la solución 01001, entonces guardaré en mi lista tabú $(1, 1, 2)$. Esto quiere decir que durante las próximas dos iteraciones no podré tener un valor de 1 en la posición 1 de mi cadena binaria. En cada iteración debo actualizar los tiempos tabú de cada elemento de la lista tabú.

1.3. Ejercicios

(80 puntos) Implementar, en el lenguaje de su preferencia un algoritmo basado en BT para resolver el problema del **agente viajero**:

- Dada una lista de ciudades y la distancia entre cada par de ellas, encontrar el camino más corto que visite cada ciudad exactamente una vez y regrese a la ciudad de origen.

Consideraciones:

- Etiquete a las ciudades como 0, 1, 2, $\cdots$, $N - 1$.
- El viajero siempre parte de la ciudad n_0.
- Tiene el mismo costo ir de la ciudad A a la ciudad B que ir de la ciudad B a la ciudad A.
- Una solución al problema es una permutación de las N ciudades.
- La solución inicial se genera utilizando un algoritmo voraz. Se parte de la ciudad n_0, se revisa el costo de ir a las $N - 1$ ciudades restantes y se elige la de menor costo. Posteriormente, se repite el proceso. Es importante considerar que estamos generando permutaciones y por lo tanto no puede haber ciudades repetidas en la solución.

- El vecindario de una solución se genera de la siguiente forma: Se elige al azar una posición de la permutación. Se generan $N-2$ soluciones nuevas moviendo la ciudad de esa posición a cualquiera de las otras $N-2$ posiciones posibles.
- En la lista tabú se almacena la ciudad que se utilizó para generar el vecindario y tendrá un tiempo tabú de $\lfloor N/2 \rfloor$.

La **entrada** al programa será:

1. La primera línea tendrá el número de ciudades N.
2. La segunda línea tendrá el número máximo de iteraciones I_{max} del algoritmo de BT.
3. Las siguientes $N-1$ líneas indicarán el costo de ir de una ciudad a otra. Es decir:
   - La primera línea tiene el costo de ir de la ciudad n_0 a la ciudad n_1, a la ciudad n_2 y hasta la ciudad n_{N-1}.
   - La segunda línea tiene el costo de ir de la ciudad n_1 a la ciudad n_2 y hasta la ciudad n_{N-1}.
   - Así sucesivamente hasta llegar a la última línea que tiene el costo de ir de la ciudad n_{N-2} a la ciudad n_{N-1}.

La **salida** del programa será:

1. Recorrido que debe seguir el viajero (permutación).
2. Costo de seguir el recorrido encontrado.

A continuación se muestra un ejemplo:

Entrada:

```
10
100
49 30 53 72 19 76 87 45 48
19 38 32 31 75 69 61 25
41 98 56 6 6 45 53
```

```
52 29 46 90 23 98
63 90 69 50 82
60 88 41 95
61 92 10
82 73
5
```

Salida:

```
0 5 2 7 8 4 1 3 9 6
467
```

(20 puntos) Permitir al usuario realizar M ejecuciones del algoritmo de BT, implementado en el punto anterior, para resolver una instancia del problema del agente viajero. Dicha instancia estará almacenada en un archivo que tendrá los parámetros de entrada tal y como se indicó en el punto anterior. Después de las M ejecuciones se debe reportar lo siguiente:

1. Mejor solución encontrada considerando las M ejecuciones.
2. Peor solución encontrada considerando las M ejecuciones.
3. Solución que corresponde a la mediana considerando las M ejecuciones.
4. Media del valor de la función objetivo considerando las M ejecuciones.
5. Desviación estándar del valor de la función objetivo considerando las M ejecuciones.

En los primeros tres puntos indica tanto el valor de x (permutación) como el valor de la función objetivo f (costo de hacer el recorrido). *Nota*: Para este punto el usuario debe indicar el nombre del archivo de entrada y el número M de ejecuciones a realizar.

Capítulo 2

Recocido Simulado

Recocido simulado (RS) es una metaheurística basada en el proceso físico por el cual un sólido se enfría lentamente hasta alcanzar una configuración de energía mínima. Su objetivo es encontrar mínimos globales en funciones que tienen varios mínimos locales. El algoritmo de metrópolis [29] es el pionero de los métodos de recocido simulado. Kirkpatrick, Gelett y Vecchi en 1983 [26] utilizaron RS para resolver el problema del agente viajero (TSP por sus siglas en inglés). RS se ha utilizado en otros problemas de optimización difíciles como el problema de coloreado de grafos, el problema de asignación cuadrática, el problema de la mochila y problemas relacionados con asignación de recursos, programación de horarios, asignación de ubicaciones, diseño de rutas, diseño de redes, forestación, planeación, etc. En [27] describen varios trabajos relacionados con los problemas antes mencionados.

2.1. Metodología de recocido simulado

El algoritmo de RS consiste en una cadena de Markov no homogénea. Es decir, es un proceso estocástico discreto en el que la probabilidad de pasar de un estado a otro depende solamente del estado inmediato anterior y del tiempo en el que se encuentre la cadena. Supongamos que la solución actual es x, RS define un vecindario para x, denotado por $N(x)$, y elige una solución y del vecindario, $y \in N(x)$. La probabilidad de ir de la solución x a la solución y es 1, si $f(y) \leq f(x)$, de lo contrario es $e^{\frac{-(f(y)-f(x))}{T(t)}}$, donde f es la función de costo a minimizar y $T(t)$ es la temperatura en el tiempo t. Al inicio de la búsqueda, la temperatura es elevada y la probabilidad de aceptar soluciones

peores a la que se tiene es alta. Conforme pasa el tiempo la temperatura va bajando y la probabilidad de aceptar soluciones peores es menor. De esta forma, RS intenta salir de óptimos locales y aspira a encontrar un óptimo global.

Los elementos principales de un algoritmo basado en RS son: 1) generación de la solución inicial, 2) elección de la temperatura inicial, 3) función de variación de la temperatura, 4) temperatura final y 5) definición del vecindario. Existen trabajos donde han estudiado el uso de diferentes funciones de distribución de probabilidad o funciones para hacer el cambio de temperatura y la importancia de la definición del vecindario, ver [4]. Con respecto al cambio en la temperatura, típicamente se utiliza $T(t+1) = \alpha T(t)$. El Algoritmo 2 muestra de manera muy general la forma en la que opera un RS.

2.2. Problema del agente viajero

Travelling Salesman Problem (TSP)

Con la finalidad de ejemplificar el uso de RS utilizaremos el problema del agente viajero. TSP es un problema de optimización combinatoria que considera un conjunto finito de ciudades y la distancia entre cada par de ellas. El objetivo es encontrar el camino más corto que visite cada ciudad exactamente una vez y regrese a la ciudad de origen. Formalmente se puede definir como sigue:

$$\begin{array}{rl} \text{minimizar:} & f(x) = d(x_n, x_1) + \sum_{i=1}^{n-1} d(x_i, x_{i+1}) \\ \text{tal que} & x_i \in \{1, 2, \cdots, n\} \end{array} \tag{2.1}$$

donde $d(x_i, x_j)$ es la distancia de ir de la ciudad x_i a la ciudad x_j, n es el número de ciudades y x es una permutación de las n ciudades. Consideremos el problema del agente viajero con $n = 5$ y la siguiente matriz de distancias:

$$\text{distancias} = \begin{array}{c|ccccc} & c_1 & c_2 & c_3 & c_4 & c_5 \\ c_1 & 0 & 49 & 30 & 53 & 72 \\ c_2 & 49 & 0 & 19 & 38 & 32 \\ c_3 & 30 & 19 & 0 & 41 & 98 \\ c_4 & 53 & 38 & 41 & 0 & 52 \\ c_5 & 72 & 32 & 98 & 52 & 0 \end{array} \tag{2.2}$$

Algoritmo 2: Recocido Simulado

Entrada: Temperatura inicial T_0, temperatura final T_f y parámetros de entrada para el problema que se quiere resolver.

Salida : Mejor solución encontrada.

Generar solución inicial x_0;
$T(0) \leftarrow T_0$;
$x(0) \leftarrow x_0$;
Actualizar la información de la mejor solución encontrada: $x_{best} \leftarrow x_0$, $f_{best} \leftarrow f(x_0)$;
$t \leftarrow 0$;
while $T(t) \leq T_f$ **do**
    Elegir aleatoriamente una solución y del vecindario, $y \in N(x(t))$;
    if $f(y) \leq f(x_{best})$ **then**
        $x_{best} = y$;
        $f_{best} = f(y)$;
    end
    if $f(y) \leq f(x(t))$ *or* $Random(0,1) < e^{\frac{-(f(y)-f(x(t)))}{T(t)}}$ **then**
        $x(t+1) \leftarrow y$;
    else
        $x(t+1) \leftarrow x(t)$;
    end
    $T(t+1) \leftarrow$ Actualizar$(T(t))$;
    $t \leftarrow t+1$;
end
Regresar x_{best}, f_{best};

Dado que siempre se regresa a la ciudad de la que se partió (lista circular), cualquier ciudad puede considerarse como la ciudad inicial. Por lo tanto, podemos dejar fija la primer posición de la permutación. En este ejemplo, asumiremos que la ciudad de origen es la ciudad 1 (c_1). A continuación se muestra un posible diseño de los elementos principales de RS.

Representación de una solución

Cada solución será vista como una permutación de las n ciudades, donde el primer elemento siempre será la ciudad 1, c_1.

- $[1, 5, 3, 2, 4]$. Esta solución consiste en partir de la ciudad c_1, posteriormente visitar la ciudad c_5, luego la ciudad c_3, después la ciudad c_2, luego la ciudad c_4 y finalmente regresar a la ciudad c_1.

Solución inicial

Utilizaremos una estrategia voraz para generar la solución inicial. Empezamos en la ciudad c_1, se revisa el costo de ir a las $N-1$ ciudades restantes y se elige la de menor costo. Posteriormente, se repite el proceso. Es importante considerar que estamos generando permutaciones y por lo tanto no puede haber ciudades repetidas en la solución.

- $[1, 3, 2, 5, 4]$. Estando en la ciudad c_1, revisamos la fila 1 de la matriz de distancias, ver (2.2), y vemos que la ciudad c_3 es la menos costosa de visitar con una distancia de 30. Posteriormente, revisamos la fila 3 y la ciudad c_2 es la más cercana a c_3 con una distancia de 19. Revisando la fila 2, vemos que la ciudad más cercana a c_2 es c_3 pero no podemos tener ciudades repetidas, así que tomamos la segunda más cercana que es c_5 con una distancia de 32. Finalmente, elegimos la ciudad faltante, c_4.

También se puede generar la permutación totalmente aleatoria y en ese caso, cada que se realice una ejecución del algoritmo, la solución inicial puede ser diferente.

Función objetivo

Para conocer el valor en la función objetivo, f, hacemos el cálculo de la Ec. 2.1. Considerando la solución $[1, 5, 3, 2, 4]$ tenemos que:

- $f([1,5,3,2,4]) = d(c_4,c_1) + d(c_1,c_5) + d(c_5,c_3) + d(c_3,c_2) + d(c_2,c_4) = 53 + 72 + 98 + 19 + 38 = 280$
- $f([1,3,2,5,4]) = d(c_4,c_1) + d(c_1,c_3) + d(c_3,c_2) + d(c_2,c_5) + d(c_5,c_4) = 53 + 30 + 19 + 32 + 52 = 186$

Vecindario

El vecindario de una solución es un conjunto de soluciones que se pueden generar realizando pequeños cambios a la solución actual. N define para cada solución candidata $x \in \Omega$ un conjunto $N(x) \subseteq \Omega$.

- Para el problema del agente viajero vamos a definir el vecindario como sigue: Se elige al azar una posición de la permutación, sin considerar la posición 1 porque es la ciudad de origen. Se mueve la ciudad de esa posición a cualquiera de las otras $N-2$ posiciones posibles generando $N-2$ soluciones nuevas. De esta forma, todas las soluciones vecinas son factibles. Si el vecindario tuviera soluciones infactibles, sería necesario agregar un mecanismo que permita lidiar con ellas.
- El vecindario de la solución $[1,3,2,5,4]$, considerando la posición aleatoria 5, es: $N([1,3,2,5,4]) = \{[1,4,3,2,5],\ [1,3,4,2,5],\ [1,3,2,4,5]\}$. Donde:
  $f([1,4,3,2,5]) = 217$,
  $f([1,3,4,2,5]) = 213$ y
  $f([1,3,2,4,5]) = 211$.
- La versión más simple de RS elige de manera aleatoria cualquiera de las soluciones del vecindario. En este caso, se podría generar una única solución del vecindario de manera aleatoria para ahorrar recursos de cómputo. Sin embargo, se pueden crear otras estrategias. Por ejemplo, asignar una probabilidad dependiendo del valor de la función objetivo.

Temperatura

La temperatura inicial y final, y la función de cambio en la temperatura son elementos clave en RS y de ellos depende que el algoritmo logre encontrar un óptimo global. Desafortunadamente, determinar estos elementos es una tarea difícil. Una posible opción es utilizar algoritmos para ajuste de parámetros [31].

- En este caso la temperatura inicial y final serán proporcionadas por el usuario y utilizaremos una función de cambio lineal:
  $T(t+1) = 0.9T(t)$

2.3. Ejercicios

(80 puntos) Implementar, en el lenguaje de su preferencia un algoritmo basado en RS para resolver **el problema de la mochila**:

- Dado un conjunto finito de objetos, donde cada objeto tiene un peso y un valor específicos, y una mochila que solo puede soportar cierto peso, encontrar el subconjunto de objetos que se pueden introducir en la mochila maximizando su valor:

$$\begin{array}{rll} \text{maximizar:} & f(\vec{x}) = \sum_{i=1}^{n} p_i \cdot x_i & \\ \text{tal que} & g_1(\vec{x}) = \sum_{i=1}^{n} w_i \cdot x_i \leq c & \\ & x_i \in \{0,1\} & i \in \{1,\ldots,n\} \end{array} \tag{2.3}$$

  donde p_i y w_i son el valor y el peso del objeto i respectivamente, n es el número de objetos y c es el peso que puede soportar la mochila.

Consideraciones:

- La representación de una solución será una cadena binaria de tamaño n. Donde cada posición i, indica con un 1 que el objeto i está en la mochila o con un 0 que no está.
- La solución inicial se genera eligiendo objetos aleatoriamente mientras no se exceda la capacidad de la mochila.
- El vecindario de una solución se define como sigue: Para cada posición de la cadena binaria, se revisa su valor y se cambia. Si la solución generada es factible (no excede la capacidad de la mochila), entonces forma parte del vecindario.
- La selección de una solución dentro del vecindario se hará de manera aleatoria.
- Para variar la temperatura se usará: $T(t+1) = 0.99T(t)$.

La **entrada** al programa será:

1. La primera línea tendrá la temperatura inicial y final, separadas por un espacio.
2. La segunda línea tendrá el número de objetos N.
3. La tercera línea tendrá la capacidad c de la mochila.
4. Las siguientes N líneas tendrán el valor p_i y el peso w_i de cada objeto i, separados por un espacio.

La **salida** del programa será:

1. Lista de objetos que estarán en la mochila y su correspondiente cadena binaria, separados por un espacio.
2. Valor de la mochila.
3. Peso de la mochila.

A continuación se muestra un ejemplo:

Entrada:
```
1000 0.1
5
15
5 2
14 3
7 7
2 5
23 10
```

Salida:
$[0, 1, 4]$ `11001`
```
42
15
```

(20 puntos) Permitir al usuario realizar M ejecuciones del algoritmo de RS, implementado en el punto anterior, para resolver una instancia del problema

de la mochila. Dicha instancia estará almacenada en un archivo que tendrá los parámetros de entrada tal y como se indicó en el punto anterior. Después de las M ejecuciones se debe reportar lo siguiente:

1. Mejor solución encontrada considerando las M ejecuciones.
2. Peor solución encontrada considerando las M ejecuciones.
3. Solución que corresponde a la mediana considerando las M ejecuciones.
4. Media del valor de la función objetivo considerando las M ejecuciones.
5. Desviación estándar del valor de la función objetivo considerando las M ejecuciones.

En los primeros tres puntos indica tanto el valor de x (lista de objetos y su correspondiente cadena binaria) como el valor de las funciones f (valor de la mochila) y g (peso de la mochila). *Nota*: Para este punto el usuario debe indicar el nombre del archivo de entrada y el número M de ejecuciones a realizar.

Reto. Cambia el diseño de RS para tratar de mejorar su desempeño en el problema de la mochila.

Capítulo 3

Programación Evolutiva

La **programación evolutiva** (PE) [12, 11] es considerada uno de los principales paradigmas de la *computación evolutiva*[1]. Fue propuesta por Lawrence J. Fogel en los años 1960s, su principal componente es la adaptación. En la PE la inteligencia es vista como un comportamiento adaptativo: la capacidad para predecir el ambiente es un prerrequisito de adaptabilidad y por lo tanto un comportamiento inteligente. En sus inicios, la PE se utilizó en tareas de predicción y la representación de las soluciones eran máquinas de estados finitos (MEF). En [12] describen un experimento que tiene como objetivo encontrar una MEF que dada una secuencia de números enteros prediga si el siguiente número es un número primo o no. Dentro del diseño del algoritmo, además de considerar el éxito de la predicción, se prefieren MEF simples. Es decir, MEF con pocos estados. El resultado obtenido del algoritmo basado en PE, considerando una secuencia de los primeros 202 números enteros, es una MEF con un único estado que siempre dice "no". El porcentaje de éxito de esta MEF es de alrededor del 81 %. Este resultado es interesante porque el algoritmo basado en PE obtiene la MEF más simple con el mayor porcentaje de éxito posible.

A partir de los años 1990s, la PE se empezó a utilizar para resolver problemas de optimización real y desde entonces es donde más se ha aplicado [6]. Un algoritmo basado en PE debe considerar los siguientes aspectos: representación de los individuos, mutación, selección de sobrevivientes y autoadaptación. Existen propuestas de incorporar un operador de recombinación. Sin embargo, no se ha comprobado que mejore el algoritmo [8, 10, 25]. En

[1]La computación evolutiva engloba un conjunto de técnicas inspiradas en la teoría Neo-Darwiniana que han sido diseñadas para resolver problemas de optimización.

esta práctica nos centraremos en el diseño para optimización real. Es decir, minimizar o maximizar una función de la forma: $f : \mathbb{R}^n \to \mathbb{R}$.

3.1. Metodología de programación evolutiva

PE crea una **población inicial**, para esto genera aleatoriamente μ individuos. Cada individuo es representado por un arreglo de valores flotantes de tamaño $2n$. Las primeras n componentes corresponden a las n variables del problema y las siguientes n componentes corresponden a los tamaños de paso para la mutación de cada una de las variables del problema. Posteriormente, se aplica el operador de **mutación** a cada invidividuo para generar μ individuos nuevos (**hijos**). Esto es porque PE considera que todos los individuos de la población son **padres**. Finalmente, PE calcula la aptitud de cada individuo, basándose en la función que se está optimizando, y realiza la **selección de sobrevivientes**. PE comúnmente utiliza una *selección más*: une las poblaciones de padres e hijos y se queda con los μ mejores individuos. Este tipo de selección es denotada como $(\mu + \mu)$. El proceso de generación de hijos y selección de sobrevivientes se repite por un determinado número de generaciones.

El operador de mutación trabaja de la siguiente forma. Dado el padre:

$$i = [x_1, x_2, \cdots, x_n, \sigma_1, \sigma_2, \cdots, \sigma_n]$$

Se genera el hijo:

$$i' = [x'_1, x'_2, \cdots, x'_n, \sigma'_1, \sigma'_2, \cdots, \sigma'_n]$$

Utilizando:

$$\sigma'_j = \sigma_j \cdot (1 + \alpha \cdot N(0,1)) \tag{3.1}$$

$$x'_j = x_j + \sigma'_j \cdot N(0,1) \tag{3.2}$$

$N(0,1)$ regresa un número aleatorio utilizando una distribución Gaussiana con media 0 y desviación estándar 1. Generalmente, se usa $\alpha \approx 0.2$ y para evitar desviaciones estándar muy pequeñas se revisa si $|\sigma'_j| < \varepsilon_0$, en caso de que se cumpla, se hace $\sigma'_j = \varepsilon_0$. Existen otras propuestas de operadores de mutación para PE, ver por ejemplo [9]. La autoadaptación se da cuando se mutan los tamaños de paso, ya que estos controlan la variación que se tendrá entre padres e hijos. La idea es que durante el proceso de búsqueda

se ajusten automáticamente los tamaños de paso. El Algoritmo 3 muestra la forma en la que opera PE en problemas de optimización sobre espacios continuos.

Algoritmo 3: Programación Evolutiva

Entrada: Tamaño de la población μ, número máximo de generaciones G, menor desviación estándar permitida ε_0, parámetro de mutación α y parámetros de entrada para el problema que se quiere resolver.

Salida : Mejor solución encontrada.

```
1  padres ← Población inicial de tamaño μ;
2  Evaluar cada individuo con f(x⃗);
3  t ← 1;
4  while t <= G do
5  |   hijos ← Población vacía;
6  |   foreach i ∈ padres do
7  |   |   σ'_j^(i) ← máx(ε_0, σ_j^(i) · (1 + α · N(0,1)));
8  |   |   x'_j^(i) = x_j^(i) + σ'_j^(i) · N(0,1);
9  |   |   hijo ← Crear un nuevo individuo con σ'_j^(i) y x'_j^(i);
10 |   |   Evaluar al nuevo individuo con f(x⃗);
11 |   |   Agregar el nuevo individuo a la población hijos;
12 |   end
13 |   padres ← Mejores μ individuos en padres ∪ hijos;
14 |   t ← t + 1;
15 end
16 x_best ← Mejor individuo de la población actual;
17 Regresar x_best, f_best;
```

3.2. Función de Beale

Con la finalidad de ejemplificar la funcionalidad de PE utilizaremos la función de Beale. La función de Beale es un problema de optimización con-

tinua de dos variables y está definida como sigue:

$$\text{mín } f(x_1, x_2) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2 \tag{3.3}$$

donde $-4.5 \leq x_1, x_2 \leq 4.5$. El mínimo global está en $x^* = (3, 0.5)$ y $f(x^*) = 0$.

Representación de un individuo

Cada individuo será visto como un arreglo de tamaño 4. Las primeras dos componentes corresponden a las variables x_1 y x_2 de la función de Beale. Las siguientes dos componentes corresponden a los tamaños de paso para la mutación, σ_1 y σ_2.

- $[-1.5, 2.8, 0.3, 0.15]$

Población inicial

Generamos un individuo de la siguiente forma: Utilizando una distribución uniforme, generamos dos números aleatorios en el intervalo $[-4.5, 4.5]$ y los guardamos en las primeras dos posiciones de un arreglo de tamaño 4. Posteriormente, generamos otros dos números aleatorios en el intervalo $(0, 1)$ y los guardamos en las últimas dos posiciones de nuestro arreglo. Repetimos el mismo procedimiento para generar la cantidad de individuos que se deseen. En este caso usaremos una población de tamaño $\mu = 100$.

Aptitud

La aptitud de un individuo está relacionada con qué tan buena es la solución que representa, dicho individuo, en determinado problema. Regularmente los valores de aptitud más grandes pertenecen a las mejores soluciones. En este caso tenemos un problema de minimización, por lo tanto, podemos definir la aptitud de un individuo i como $F_a(i) = -f(x_1^{(i)}, x_2^{(i)})$, donde f es la función de Beale. De esta forma las soluciones con un valor menor en la función de Beale, tendrán un valor de aptitud mayor.

- $F_a([-1.5, 2.8, 0.3, 0.15]) = -f(-1.5, 2.8) = 895.21$

Mutación y autoadaptación

Supongamos que el individuo padre es $i = [-1.5, 2.8, 0.3, 0.15]$, que $\alpha = 0.2$, $\varepsilon_0 = 0.01$ y que utilizando una distribución normal, con media 0 y desviación estándar 1, se generaron los siguientes números aleatorios: 1.63, 1.49, 0.22 y -0.5. Aplicamos la Ec. 3.1 para mutar los valores de los tamaños de paso:

$$\sigma_1' = \sigma_1 \cdot (1 + \alpha \cdot N(0,1)) = 0.3 \cdot (1 + 0.2 \cdot 1.63) = 0.3978$$
$$\sigma_2' = \sigma_2 \cdot (1 + \alpha \cdot N(0,1)) = 0.15 \cdot (1 + 0.2 \cdot 1.49) = 0.1947$$

Una vez que tenemos estos valores, podemos mutar las variables de decisión utilizando la Ec. 3.2:

$$x_1' = x_1 + \sigma_1' \cdot N(0,1) = -1.5 + 0.3978 \cdot 0.22 = -1.4124$$
$$x_2' = x_2 + \sigma_2' \cdot N(0,1) = 2.8 + (-0.1947) \cdot (-0.5) = 2.7026$$

Finalmente, construimos el individuo hijo:

$$i' = [-1.4124, 2.7026, 0.3978, 0.1947]$$

Cuando se aplica el operador de mutación, se debe revisar que las variables de decisión estén dentro del rango que indica el problema. En caso de que no, se debe definir una estrategia. En este ejemplo se asignará el valor del límite que rebasó. De igual forma se debe revisar que los tamaños de paso no sean menores a ε_0. En caso de que sí, se deben igualar con ε_0.

Selección de sobrevivientes

En este caso utilizaremos una selección más. Por lo tanto, a partir de 100 padres, vamos a generar 100 hijos. Unimos ambas poblaciones y seleccionamos los 100 mejores individuos de acuerdo a su aptitud.

3.3. Ejercicios

(80 puntos) Implementar, en el lenguaje de su preferencia, un algoritmo basado en PE para resolver el siguiente problema:

$$\text{mín } f(\vec{x}) = -20 \exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e \tag{3.4}$$

La función f es conocida como la **función de Ackley**, su dominio es $|x_i| \leq 30$ y su mínimo global está en $x_i = 0$ y $f(\vec{x}) = 0.0$.

Consideraciones:

- Cuidar que las variables de decisión no se salgan del rango especificado en el problema. Si eso ocurre, deberá contar con un mecanismo que vuelva a generar el valor de la variable o que lo ajuste al rango deseado. Explique en su reporte qué mecanismo fue utilizado.

La **entrada** al algoritmo será:

1. La primera línea tendrá el número de variables que tendrá la función de Ackley n.
2. La segunda línea tendrá el tamaño de la población y el número máximo de generaciones, separados por un espacio.
3. La tercera línea tendrá el valor de los parámetros α y ε_0, separados por un espacio.

La **salida** del programa será:

1. La mejor solución encontrada.
2. El valor de la función objetivo para dicha solución.

A continuación se muestra un ejemplo:

Entrada:
```
2
100 200
2.0 0.0001
```

Salida:
$[7.48188924e-08, -6.67209514e-07]$
```
f(x) = 0.000
```

(20 puntos) Permitir al usuario realizar M ejecuciones del algoritmo de PE, implementado en el punto anterior, para resolver una instancia del problema de Ackley. Después de las M ejecuciones se debe reportar lo siguiente:

1. Mejor solución encontrada considerando las M ejecuciones.
2. Peor solución encontrada considerando las M ejecuciones.
3. Solución que corresponde a la mediana considerando las M ejecuciones.
4. Media del valor de la función objetivo considerando las M ejecuciones.
5. Desviación estándar del valor de la función objetivo considerando las M ejecuciones.

En los primeros tres puntos indica tanto el valor de x como el valor de la función objetivo f (función de Ackley). *Nota*: Se deben reportar los resultados del algoritmo para las siguientes instancias: 5, 10 y 20 variables de decisión. Recuerde que puede variar los parámetros del algoritmo (tamaño de la población, número de generaciones, α y ε_0). Indique los valores que se utilizaron para cada instancia del problema.

Capítulo 4

Estrategias Evolutivas

Las **estrategias evolutivas** (EEs) fueron propuestas por Hans-Paul Schwefel e Ingo Rechenberg [37, 32] en los años 1960s. Las EEs, al igual que la programación evolutiva, están basadas en el proceso evolutivo de los seres vivos y son consideradas uno de los tres principales paradigmas de la computación evolutiva[1].

Una característica que distingue a las EEs de otros algoritmos evolutivos es la **auto-adaptación**: los operadores de mutación son capaces de adaptar la forma de las distribuciones utilizadas para mutar a los individuos, de acuerdo al comportamiento de la función objetivo en el vecindario de cada individuo. Desde sus inicios, las EEs se han utilizando comúnmente para resolver problemas de optimización sobre espacios continuos.

La primera aplicación de una EE fue en el diseño de boquillas y formas de ala. En este caso, la función objetivo se evaluaba a partir de experimentos. Es decir, no se contaba con una expresión matemática de la función objetivo. Los resultados obtenidos arrojaron estructuras de alto desempeño que nunca antes habían sido consideradas por los ingenieros.

Las EEs han mostrado un buen desempeño tanto en problemas de prueba como en aplicaciones en ingeniería [39, 1, 40]. En [40] se describen varios trabajos en los que se han utilizado EEs para resolver problemas relacionados con tecnología de la construcción, combustibles energéticos, ingeniería civil e ingeniería eléctrica y electrónica.

La EE más simple es conocida como $(1 + 1)$-EE y trabaja de la siguiente forma: Primero crea un individuo inicial $\vec{x}$ que es conocido como **padre**.

[1]La computación evolutiva engloba un conjunto de técnicas inspiradas en la teoría Neo-Darwiniana que han sido diseñadas para resolver problemas de optimización.

Después, genera un **hijo** $\vec{y} = \vec{x} + \vec{z}$, donde $\vec{z}$ es un vector generado con números aleatorios utilizando una distribución normal con media 0 y desviación estándar σ: $z_i = N(0, \sigma)$. El valor de σ controla qué tanto son perturbadas las variables de decisión a la hora de generar al hijo. Este valor se ajusta cada determinado número de iteraciones, es decir se **auto-adapta**, utilizando la **regla del éxito** 1/5 [33]. Esta regla enuncia que la *tasa de mutaciones exitosas* (es decir, mutaciones en las que el hijo es mejor que el padre), debería ser exactamente de 1/5. La regla del éxito del 1/5 es aplicada cada k iteraciones de la siguiente forma:

$$\sigma = \begin{cases} \sigma/c & \text{Si } p_s > 1/5, \\ \sigma \cdot c & \text{Si } p_s < 1/5, \\ \sigma & \text{Si } p_s = 1/5 \end{cases} \tag{4.1}$$

Donde p_s es el porcentaje de mutaciones exitosas y $0.817 \leq c \leq 1$ [38]. Si se tienen demasiadas mutaciones exitosas ($p_s > 1/5$), σ debería ser incrementado para realizar una *búsqueda más amplia* en el espacio de soluciones. Si se tienen muy pocas mutaciones exitosas ($p_s < 1/5$), σ debería ser decrementado para *concentrar la búsqueda* alrededor del individuo actual. Finalmente, padre e hijo compiten y el que tenga una mejor aptitud será el padre de la siguiente iteración. El Algoritmo 4 muestra la forma en la que opera esta versión simple de las EEs.

Actualmente existen versiones de las EEs conocidas como $(\mu + \lambda)$-EE y (μ, λ)-EE que trabajan con poblaciones de soluciones. Su nombre se debe a la cantidad de hijos que generan y al mecanismo de selección de sobrevivientes que utilizan. De igual forma, existen varias propuestas para los otros componentes de las EEs [7, 40]. De entre las EEs más destacadas está la llamada "Covariance Matrix Adaptation Evolution Strategy (CMA-ES)" propuesta en 1996 por Hansen et. al. [22].

4.1. Metodología de estrategias evolutivas

Las EEs parten de un conjunto inicial de individuos, cada uno de los cuales es una posible solución al problema de optimización. Posteriormente, aplican operadores de variación y selección a los individuos de la población.

Algoritmo 4: $(1+1)$-EE

Entrada: Valor inicial de σ, constante c, número máximo de generaciones G y parámetros de entrada para el problema que se quiere resolver.

Salida : Mejor solución encontrada.

```
1  t ← 0;
2  m_successful ← 0;
3  Obtener solución inicial x⃗;
4  Evaluar f(x⃗);
5  while t ≤ G do
6  |  Mutar la solución x⃗: x'_i = x_i + σ · N_i(0,1);
7  |  if x⃗' es mejor que x⃗ then
8  |  |  x⃗ ← x⃗';
9  |  |  m_successful ← m_successful + 1;
10 |  end
11 |  t ← t + 1;
12 |  if t mod k == 0 then
13 |  |  p_s ← m_successful/k;
14 |  |  If p_s > 1/5 then σ ← σ/c ;
15 |  |  If p_s < 1/5 then σ ← σ · c ;
16 |  |  m_successful ← 0;
17 |  end
18 end
19 Regresar x⃗;
```

Los operadores de variación (cruza y mutación) permiten crear nuevos individuos y los operadores de selección eligen individuos de acuerdo a su aptitud. La aptitud de cada individuo depende de la función objetivo que se esté optimizando, regularmente los mejores individuos tienen asignado un valor de aptitud más alto. A continuación se describen algunas propuestas para cada componente de una EE, asumiendo que se está resolviendo un problema de optimización donde la función objetivo es del tipo $f : \mathbb{R}^n \rightarrow \mathbb{R}$.

Representación

Cada individuo almacena tanto las variables del problema que se desea resolver como los parámetros que utiliza la EE en el operador de mutación. Por ejemplo: $[x_1, \cdots, x_n, \sigma_1, \cdots, \sigma_m]$. $x_1, \cdots, x_n$ son valores flotantes asociados a cada una de las n variables de decisión del problema. $\sigma_1, \cdots, \sigma_m$ definen las formas de las distribuciones utilizadas para mutar a los invidividuos (tamaños de paso). Regularmente se utiliza $m = 1$ ó $m = n$. En el primer caso, se utiliza un solo tipo de distribución para mutar todas las variables, mientras que en el segundo caso, se usa un tipo de distribución diferente para cada variable del problema.

Selección de padres

Las EEs seleccionan a los individuos que serán padres de manera aleatoria, utilizando una distribución uniforme. Es decir, todos los individuos tienen la misma probabilidad de ser padres, sin importar su aptitud.

Cruza

Regularmente, el operador de recombinación o cruza utiliza dos individuos (padres) para generar un nuevo individuo (hijo). Dos tipos de recombinaciones posibles son las siguientes: i) **recombinación discreta** y ii) **recombinación intermedia**. En la recombinación discreta se recorren uno a uno los componentes de cada padre y se decide de manera aleatoria si la información del nuevo individuo se toma del padre 1 ó del padre 2. La recombinación intermedia genera al nuevo individuo utilizando el promedio de ambos padres para cada componente. Supongamos que a partir de los padres $\vec{x}$ y $\vec{y}$, se va a generar al hijo $\vec{z}$, los operadores quedarían de la siguiente forma:

- Recombinación discreta: $z_i = x_i$ ó y_i (elegido aleatoriamente).

- Recombinación intermedia: $z_i = \frac{x_i + y_i}{2}$.

Algunos autores recomiendan utilizar recombinación discreta en las variables de decisión e intermedia para los parámetros del algoritmo [6]. También es posible utilizar más de dos individuos padres y aplicar estos operadores de recombinación. Cuando se utilizan más de dos padres se conoce como **recombinación global** y cuando se utilizan dos padres, se denomina **recombinación local**.

Mutación

El operador de mutación utiliza una distribución normal. Este tipo de distribución requiere de dos parámetros: la media μ y la desviación estándar σ. Regularmente, μ es igual con 0. σ se va estableciendo conforme la búsqueda avanza. Es decir, se va auto-adaptando. Esto se logra porque, como vimos en la sección anterior, cada individuo almacena los valores de σ utilizados y estos valores también son perturbados por el operador de mutación. Es importante, tener en cuenta que primero se aplica la mutación a los valores de σ. Una vez que se tienen los nuevos valores σ', se utilizan para mutar a las variables de decisión. De esta forma, cuando se evalúa al individuo (cálculo de su aptitud) se evalúa tanto el valor de σ que se utilizó para generar ese individuo como su valor en la función objetivo: Si un individuo tiene una buena aptitud, significa que la mutación en las variables de decisión dieron un buen resultado. Las variables de decisión se mutan de la siguiente forma:

$$x_i' = x_i + \sigma_i' \cdot N(0,1) \tag{4.2}$$

Donde $N(0,1)$ genera un número aleatorio utilizando una distribución normal con media 0 y desviación estándar 1. Recordemos que dependiendo del diseño de la EE, se puede tener una única σ para todas las variables o se puede tener una σ_i para cada variable x_i o algunas variables pueden compartir un valor de σ y tener m cantidad de σ. En esta práctica solo vamos a estudiar el caso en el que se tiene un único valor de σ para todas las variables o un valor de σ para cada variable. Puede consultar otras variantes en [2, 3, 6, 7].

Mutación con un único tamaño de paso

Cuando se tiene un único valor σ en cada individuo: $[x_1, \cdots, x_n, \sigma]$, el valor σ se muta de la siguiente manera:

$$\sigma' = \sigma \cdot e^{\tau \cdot N(0,1)} \tag{4.3}$$

Donde $N(0,1)$ genera un número aleatorio utilizando una distribución normal con media 0 y desviación estándar 1. τ es un parámetro de entrada al algoritmo que es visto como una tasa de aprendizaje. Regularmente, se usa $\tau = 1/\sqrt{n}$, donde n es el número de variables del problema. Para evitar valores muy pequeños en σ, si $\sigma' < \varepsilon_0$, entonces se hace $\sigma' = \varepsilon_0$. En [2], Bäck explica las razones del por qué usar la Ec. 4.3 para mutar el tamaño de paso.

Mutación con un tamaño de paso por variable de decisión

La motivación de utilizar un tamaño de paso por variable es que en muchos problemas se tiene una pendiente diferente para cada variable. En este caso los valores de cada σ se mutan de la siguiente forma:

$$\sigma_i' = \sigma_i \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)}$$

Donde $N(0,1)$ genera un número aleatorio utilizando una distribución normal con media 0 y desviación estándar 1. $N_i(0,1)$ indica que se genera un número aleatorio por cada σ_i. τ y τ' son parámetros de entrada al algoritmo que son vistos como tasas de aprendizaje. Se recomienda utilizar $\tau' = 1/\sqrt{2n}$ y $\tau = 1/\sqrt{2\sqrt{n}}$. Una vez más, para evitar valores muy pequeños en σ, si $\sigma' < \varepsilon_0$, entonces se hace $\sigma' = \varepsilon_0$.

Selección de sobrevivientes

Las EEs suelen utilizar alguno de los siguientes esquemas de selección de sobrevivientes:

- Selección (μ, λ). A partir de μ posibles padres, se generan λ hijos, con $\lambda \geq \mu$. De los λ hijos, se seleccionan a los μ mejores. Estos individuos formarán parte de la siguiente iteración. Este tipo de selección se recomienda en problemas en los que el óptimo es dinámico o cuando la función es multimodal. Se recomienda utilizar $\lambda = 7\mu$.

- Selección $(\mu + \lambda)$. A partir de μ posibles padres, se generan λ hijos. Se unen las poblaciones de padres e hijos y se seleccionan a los mejores μ individuos. Estos individuos formarán parte de la siguiente iteración. Este tipo de selección se considera **elitista**.

4.2. Función de Ackley

Con la finalidad de ejemplificar el uso de una EE utilizaremos la función de Ackley. La función de Ackley es un problema de optimización continua de n variables y está definida como sigue:

$$\text{mín } f(\vec{x}) = -20 \exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e \tag{4.4}$$

donde $|x_i| \leq 30$ y su mínimo global está en $x_i = 0$ y $f(\vec{x}) = 0.0$.

Representación de un individuo

Cada individuo será visto como un arreglo de tamaño $n + 1$. Las primeras n componentes corresponden a las variables de decisión del problema x_i y la última componente corresponde con el tamaño de paso σ. En este caso usaremos un único tamaño de paso para todas las variables de decisión.

- Si $n = 2$, un posible individuo es $[-10.3, 7.84, 0.84]$.

Población inicial

Generamos un individuo de la siguiente forma: Utilizando una distribución uniforme, generamos n números aleatorios en el intervalo $[-30, 30]$ y los guardamos en las primeras n posiciones de un arreglo de tamaño $n+1$. Posteriormente, generamos otro número aleatorio en el intervalo $(0, 1)$ y lo guardamos en la última posición de nuestro arreglo. Repetimos el mismo procedimiento para generar la cantidad de individuos que se deseen. En este caso usaremos una población de tamaño $\mu = 100$.

Aptitud

La aptitud de un individuo está relacionada con qué tan buena es la solución, que representa dicho individuo, en determinado problema. Regularmente los valores de aptitud más grandes pertenecen a las mejores soluciones. En este caso tenemos un problema de minimización, por lo tanto, podemos definir la aptitud de un individuo i como $F_a(i) = -f(x_1^{(i)}, x_2^{(i)})$, donde f es la función de Ackley.

- $F_a([-10.3, 7.84, 0.84]) = -f(-10.3, 7.84) = -18.39$

Cruza

En este caso vamos a utilizar una recombinación discreta local. Es decir, para generar un nuevo invidividuo, vamos a seleccionar dos individuos de la población actual de manera aleatoria. Supongamos que son $x = [-10.3, 7.84, 0.84]$ y $y = [2.4, -3.84, 0.98]$. Posteriormente, vamos a generar tres números aleatorios en el intervalo $(0, 1)$ utilizando una distribución uniforme. Si el número aleatorio es menor a 0.5 se tomará la componente del padre x para formar al hijo z, de lo contrario se tomará la componente del padre y. Supongamos que los números aleatorios son: 0.45, 0.24 y 0.56, el nuevo individuo es:

$$z = [-10.3, 7.84, 0.98]$$

Mutación

Dado que tenemos un único valor de σ para mutar todas las variables, utilizaremos la Ec. 4.3 para mutar el tamaño de paso. Sea $z = [-10.3, 7.84, 0.98]$ el hijo que deseamos mutar, $\tau = 1/\sqrt{2}$ y 1.112 un número aleatorio generado con una distribución normal con media 0 y desviación estándar 1, obtenemos:

$$\sigma' = \sigma \cdot e^{\tau \cdot N(0,1)} = 0.98 \cdot e^{0.7071 \cdot 1.112} = 2.1513$$

Para mutar las variables de decisión, usamos la Ec. 4.2. Supongamos que los números aleatorios generados son 0.872 y 1.533.

$$x_1' = x_1 + \sigma' \cdot N(0,1) = -10.3 + 2.1513 \cdot 0.872 = -8.424$$

$$x_2' = x_2 + \sigma' \cdot N(0,1) = 7.84 + 2.1513 \cdot 1.533 = 11.1379$$

El individuo mutado es:

$$z' = [-8.424, 11.1379, 2.1513]$$

y su aptitud es:

$$F_a(z') = -f(-8.424, 11.1379) = 19.056$$

Selección de sobrevivientes

Para este ejemplo, vamos a utilizar una selección (μ, λ), donde $\mu = 100$ y $\lambda = 700$. Es decir, a partir de 100 individuos se van a generar 700 hijos y pasarán a la siguiente iteración los mejores 100 hijos. El Algoritmo 5 muestra el funcionamiento completo de la EE utilizada en este ejemplo.

4.3. Ejercicios

(20 puntos) Implementar, en el lenguaje de su preferencia, el Algoritmo 4 para resolver la función de Ackley.

(40 puntos) Implementar, en el lenguaje de su preferencia, un algoritmo basado en EEs para resolver la función de Ackley. El objetivo es mejorar los resultados obtenidos con el diseño propuesto en el ejemplo.

(20 puntos) Permitir al usuario realizar M ejecuciones del algoritmo de EE, implementado en el punto anterior, para resolver una instancia del problema de Ackley. Después de las M ejecuciones se debe reportar lo siguiente:

1. Mejor solución encontrada considerando las M ejecuciones.
2. Peor solución encontrada considerando las M ejecuciones.
3. Solución que corresponde a la mediana considerando las M ejecuciones.
4. Media del valor de la función objetivo considerando las M ejecuciones.
5. Desviación estándar del valor de la función objetivo considerando las M ejecuciones.

Algoritmo 5: (μ, λ)-EE

Entrada: Número máximo de generaciones G, tasa de aprendizaje τ, mínimo valor de tamaño de paso ε_0, tamaño de la población μ, tamaño de la población de hijos λ y parámetros de entrada para el problema que se quiere resolver.

Salida : Mejor solución encontrada.

```
1  t ← 0;
2  población ← Obtener población inicial;
3  while t ≤ G do
4  |   cont ← 0;
5  |   hijos ← ∅;
6  |   while cont < λ do
7  |   |   padres ← Seleccionar aleatoriamente dos individuos de la
   |   |   población actual (población);
8  |   |   hijo ← Generar un hijo aplicando los operadores de cruza y
   |   |   mutación a los padres (padres);
9  |   |   hijos ← hijos ∪ {hijo};
10 |   |   cont ← cont + 1;
11 |   end
12 |   población ← Mejores μ individuos de hijos;
13 |   t ← t + 1;
14 end
15 x_best ← Mejor individuo de la población actual;
16 Regresar x_best, f_best;
```

En los primeros tres puntos indica tanto el valor de x como el valor de la función objetivo f (función de Ackley). *Nota*: Se deben reportar los resultados del algoritmo para las siguientes instancias: 5, 10 y 20 variables de decisión. Recuerde que puede variar los parámetros del algoritmo. Indique los valores que se utilizaron para cada instancia del problema.

(20 puntos) Realizar una tabla comparativa que reporte los resultados de las tres versiones de EEs implementadas: $(1 + 1) - EE$, $(\mu, \lambda) - EE$ y la versión propuesta. Se deben reportar los resultados para problemas con 2, 5, 7 y 10 variables.

Capítulo 5

Algoritmos Genéticos

Los **algoritmos genéticos** (AGs) son métodos de búsqueda basados en la evolución de los seres vivos, fueron desarrollados originalmente por John H. Holland [24] y forman parte de los tres principales paradigmas de la computación evolutiva[1].

En la naturaleza, los individuos compiten por los recursos limitados. Las características de cada individuo son determinadas por su contenido genético. Regularmente, los individuos más fuertes son los que pueden sobrevivir y reproducirse, por lo que su información genética prevalece. La evolución de los seres vivos se atribuye a los cambios genéticos en las especies. Las especies que no son capaces de adaptarse a los cambios en el ambiente están destinadas a desaparecer.

En [42, 28, 34, 35] se mencionan algunas aplicaciones de lo AGs en problemas de control y procesamiento de señales, robótica, reconocimiento de patrones, reconocimiento de voz, diseños en ingeniería, sistemas de clasificación y diversos problemas de optimización combinatoria.

5.1. Metodología de algoritmos genéticos

Los AGs trabajan con poblaciones de soluciones potenciales a un problema de optimización. Las soluciones son codificadas (información genética) y posteriormente manipuladas utilizando operadores genéticos (recombinación y mutación). Cada solución es considerada un individuo y tiene asociado un

[1]La computación evolutiva engloba un conjunto de técnicas inspiradas en la teoría Neo-Darwiniana que han sido diseñadas para resolver problemas de optimización.

valor de aptitud. La aptitud refleja qué tan buena es la solución en comparación con las otras soluciones de la población, con respecto al problema de optimización que se está resolviendo. Los individuos con un mayor valor de aptitud tienen probabilidades altas de reproducirse y sobrevivir. De esta forma, la información genética de las mejores soluciones se mantiene en la población. En la actualidad, existen varios tipos de representaciones, operadores genéticos y operadores de selección que pueden utilizarse para diseñar un AG [41, 6, 30]. En esta práctica sólo consideraremos algunas propuestas. El Algoritmo 6 muestra de manera general cómo opera un AG.

El AG clásico, también conocido como AG simple, utiliza una *representación binaria*. Es decir, la solución está representada como una cadena binaria, la cual es vista como el **cromosoma** del individuo. Dicho cromosoma está dividido (subcadenas) y cada parte es conocida como **gen**. Cada gen representa una de las variables de decisión del problema de optimización. Para elegir a los individuos que van a reproducirse, los cuales son llamados **padres**, el AG clásico utiliza una *selección de ruleta*. Este tipo de selección asigna una probabilidad a cada individuo que es proporcional a su aptitud. Dado que se trabaja con probabilidades, es importante definir la aptitud de los individuos tal que todos los individuos tengan una aptitud mayor o igual a cero y que el mejor individuo de la población tenga asignado el mayor valor de aptitud. Posteriormente, simula una ruleta donde el tamaño de división correspondiente a cada individuo está dado por la probabilidad antes asignada. Para generar dos nuevas soluciones, las cuales son consideradas como **hijos**, se utilizan dos padres y se realiza lo siguiente: dependiendo de una probabilidad p_c, se decide si los hijos son generados de manera asexual o con un operador de cruza de un punto. p_c suele estar en $[0.5, 1)$. Si es de manera asexual los hijos son copias de los padres. Si es aplicando cruza de un punto, el primer hijo toma desde la posición 0 del cromosoma del primer padre y hasta la posición k (punto de cruza), k es elegido aleatoriamente. La segunda parte del cromosoma se toma desde la posición $k + 1$ del cromosoma del segundo padre y hasta el final. El segundo hijo se genera de manera similar pero empezando con el cromosoma del segundo padre y terminando con el cromosoma del primer padre. Una vez generados los dos hijos, se les aplica el operador de mutación conocido como *Bitflip*. Este operador recorre el cromosoma posición por posición y con una probabilidad p_m cambia el valor del bit. p_m suele ser un valor muy pequeño. Si la población de padres es de tamaño μ, se generan μ hijos. Los hijos formarán la población de la siguiente **generación** (iteración).

Es importante mencionar que aunque el AG clásico sustituye completamente a la población actual con la población de hijos generados, en 1994 se demostró que es necesario retener en la población a la mejor solución encontrada hasta el momento para que el AG pueda converger a la solución óptima [36]. Esto se conoce como **elitismo**. En los AGs, a diferencia de la programación evolutiva y las estrategias evolutivas, el operador de búsqueda principal es el operador de cruza y el operador de mutación es un operador secundario que puede ayudar a salir de óptimos locales.

Representación

En un inicio, los AGs utilizaban representación binaria y se debía tener un mecanismo para codificar las variables del problema de optimización usando cadenas de 0's y 1's. En la actualidad, existen varias propuestas de representación. Por ejemplo, representación entera y representación real. El mapeo entre las variables y su cromosoma dependerá de la representación que se esté usando y del problema de optimización que se esté resolviendo.

Representación binaria para problemas de optimización en espacios continuos. Una forma de utilizar una subcadena de 0's y 1's para representar una variable x, tal que $x \in \mathbb{R}$, es discretizar su rango, fijando una precisión y tratando esos valores como si fueran enteros. Por ejemplo, si x está en el intervalo $[l_{inf}, l_{sup}]$ y deseamos una precisión de p dígitos, la cadena binaria debe tener la siguiente longitud:

$$\text{num\_bits} = \lceil \log_2((l_{sup} - l_{inf}) * 10^p) \rceil \tag{5.1}$$

Es importante observar que no hay una relación uno a uno entre las cadenas binarias de tamaño num_bits y los números enteros en el intervalo $[l_{inf} \cdot 10^p, l_{sup} \cdot 10^p]$. Para hacer el mapeo entre una cadena binaria b y su valor real r podemos hacer lo siguiente:

1. Obtener el valor entero d correspondiente a la cadena binaria b.
2. Obtener r con: $r = l_{inf} + \frac{(l_{sup} - l_{inf}) \cdot d}{2^{tamano} - 1}$

Representación real para problemas de optimización en espacios continuos. Debido a que existen muchos problemas de optimización donde

Algoritmo 6: Algoritmo genético

Entrada: Número máximo de generaciones G, tamaño de la población μ, probabilidad de cruza p_c, probabilidad de mutación p_m y parámetros de entrada para el problema que se quiere resolver.

Salida : Mejor solución encontrada.

```
1  pob_actual ← Población inicial de tamaño μ;
2  Asignar aptitud a cada individuo de pob_actual considerando f(x⃗);
3  t ← 1;
4  while t <= G do
5  |  padres ← Utilizar un método de selección para determinar los μ
   |  individuos, de pob_actual, que actuarán como padres;
6  |  hijos ← ∅;
7  |  foreach i, j ∈ padres do
8  |  |  if random(0,1) < p_c then
9  |  |  |  Aplicar operador de cruza a i y j y generar dos hijos h_1 y
   |  |  |  h_2;
10 |  |  else
11 |  |  |  Generar dos hijos h_1 y h_2 de manera asexual: h_1 ← i,
   |  |  |  h_2 ← j;
12 |  |  end
13 |  |  Aplicar operador de mutación a h_1 y h_2, usando p_m;
14 |  |  hijos ← hijos ∪{h_1, h_2};
15 |  end
16 |  pob_actual ← Seleccionar los μ individuos sobrevivientes;
17 |  t ← t + 1;
18 end
19 x_best ← Mejor individuo de la población actual;
20 Regresar x_best, f_best;
```

la función objetivo es del tipo $f : \mathbb{R}^n \to \mathbb{R}$, a lo largo del tiempo se han propuesto operadores genéticos que permiten trabajar con los valores de las variables directamente. De esta forma, si el problema de optimización tiene n variables, el cromosoma de cada individuo está formado por n genes, donde cada gen es un valor flotante.

Representación para permutaciones. Aunque en los inicios de los AGs se utilizaban cadenas binarias para codificar una permutación, la representación más natural es una secuencia de enteros en la que no puede haber repeticiones. En la actualidad, existen varios operadores genéticos que trabajan sobre cadenas de enteros y producen soluciones válidas (cadenas de enteros, donde los enteros no pueden repetirse).

Población inicial

Los AGs parten de un conjunto de individuos que se generan aleatoriamente. Regularmente, se desea que los individuos estén distribuidos uniformemente en todo el espacio de búsqueda. Por lo anterior, se suele utilizar una distribución uniforme para generar los cromosomas de cada individuo. Una vez que se tienen los cromosomas, se les asigna una aptitud a cada individuo utilizando la función objetivo a optimizar.

Selección de padres

Existen diferentes mecanismos para seleccionar a los individuos que actuarán como padres. A continuación mencionaremos algunos.

Selección proporcional. Los métodos de selección proporcional asignan una probabilidad de selección a cada individuo, la cual depende tanto del valor de aptitud del individuo como de la aptitud del resto de los individuos. Por ejemplo, $p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$, donde p_i es la probabilidad de seleccionar como padre al individuo i, f_i es la aptitud del individuo i y f_j es la aptitud del individuo j. Como se mencionó anteriormente, la aptitud debe ser mayor o igual a 0 y un mayor valor de aptitud significa que el individuo es mejor. Existen varios métodos de selección proporcional; aquí solo mencionaremos dos: *la ruleta* y *universal estocástica*, ver Algoritmos 7 y 8. El *método de la ruleta* asigna a cada individuo una porción de la ruleta (valor esperado). Para seleccionar a un individuo, se simula que se gira la ruleta, generando

individuo	$y = f(x)$	$p(y)$	$y' = f(x) + 100$	$p(y')$
1	10	0.1428	110	0.2340
2	15	0.2142	115	0.2446
3	20	0.2857	120	0.2553
4	25	0.3571	125	0.2659

Cuadro 5.1: Asignación de probabilidades de acuerdo a la aptitud de cada individuo.

un número aleatorio y se observa a cuál individuo corresponde la porción en la que se detuvo. Este proceso se repite hasta seleccionar el número total de padres requeridos. En el *método de universal estócastica*, se calculan los valores esperados de cada individuo. Sea E_i el valor esperado del individuo i, el número de copias del individuo i es al menos $\lfloor E_i \rfloor$. Hacer una copia más del individuo i, depende de un valor aleatorio y de la parte decimal de E_i.

Las principales desventajas de los métodos de selección proporcional son las siguientes:

1. Dado que se generan varias copias de los mejores individuos, estos métodos pueden ocasionar convergencia prematura.

2. Si todos los individuos de la población tienen valores de aptitud muy parecidos, no existe presión de selección y la búsqueda se vuelve completamente aleatoria.

3. Trasladar la función de aptitud, puede ocasionar cambios significativos en las probabilidades de selección de cada individuo. Por ejemplo, $y = f(x)$ y $y' = f(x)+100$ asignan probabilidades diferentes. Ver Tabla 5.1.

Selección por jerarquías. Estos métodos jerarquizan a los individuos de acuerdo a su aptitud. Posterioremente, asignan una probabilidad de selección de acuerdo a su jerarquía y se hace uso de algún mecanismo como la ruleta para elegir a los individuos. A continuación se muestra una forma de hacer el cálculo de las probabilidades. Sea s un valor tal que $1 < s \leq 2.0$, μ el número de individuos en la población, 0 la jerarquía para el peor individuo y $\mu - 1$ la jerarquía para el mejor individuo, la probabilidad del individuo con jerarquía i está dada por:

$$p_{lineal}(i) = \frac{(2-s)}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)} \tag{5.2}$$

Algoritmo 7: La ruleta

Entrada: Población de tamaño μ.
Salida : Población de μ individuos seleccionados.

Sea f_m la aptitud promedio de la población, calcular el valor esperado para cada individuo i: $E_i = \frac{f_i}{f_m}$;
padres $\leftarrow \emptyset$;
for $k = 1$ **to** μ **do**
    Generar un número aleatorio entre 0 y μ, utilizando una distribución uniforme: $r \leftarrow rnd(0.0, \mu)$;
    $suma \leftarrow 0$;
    $i \leftarrow -1$;
    while $suma < r$ **do**
        $i \leftarrow i + 1$;
        $suma \leftarrow suma + E_i$;
    end
    Seleccionar al individuo i como padre:
    padres $\leftarrow$ **padres** $\cup \{i\}$;
end
Regresar **padres**;

Algoritmo 8: Universal estocástica

Entrada: Población de tamaño μ.
Salida : Población de μ individuos seleccionados.

Sea f_m la aptitud promedio de la población, calcular el valor esperado para cada individuo i: $E_i = \frac{f_i}{f_m}$;
Generar un número aleatorio entre 0 y 1, utilizando una distribución uniforme: $r \leftarrow rnd(0, 1)$;
padres $\leftarrow \emptyset$;
$suma \leftarrow 0$;
for $i \leftarrow 1$; $i \leq \mu$; $i++$ **do**
    $suma \leftarrow suma + E_i$;
    while $suma > r$ **do**
        Seleccionar al individuo i como padre:
        padres $\leftarrow$ **padres** $\cup \{i\}$;
        $r \leftarrow r + 1$;
    end
end
Regresar **padres**;

En este caso, existe un mapeo lineal entre la jerarquía de cada invidivudo y su probabilidad de ser seleccionado. El valor de s controla la presión de selección. Por ejemplo, si $s = 2$, el peor individuo tiene una probabilidad 0 de ser seleccionado. Si se desea una presión de selección más alta, se puede hacer uso de mapeos exponenciales, por ejemplo:

$$p_{exp}(i) = \frac{1 - e^{-i}}{c} \tag{5.3}$$

Donde el valor de c depende del tamaño de la población y se elige de tal forma que la sumatoria de todas las probabilidades sea 1.

Selección por torneo. En este caso, se seleccionan aleatoriamente t individuos de la población. Como este proceso se repite varias veces, la selección se puede hacer con o sin reemplazo. Regularmente, el mejor individuo de los t individuos es seleccionado como padre[2]. Este proceso se repite hasta seleccionar el total de padres requeridos. Si la selección de los t individuos se hace sin reemplazo y la población es de tamaño μ, después de $\frac{\mu}{t}$ iteraciones no se tienen más individuos. Por lo anterior, se deberán considerar nuevamente todos los invididuos de la población.

Recombinación

La recombinación o cruza es el proceso por medio del cual se crea una solución hija a partir de la información contenida en dos o más soluciones padres. Debido a que este proceso es considerado uno de los más importantes en un AG, varios grupos de investigación se han enfocado en el estudio y diseño de operadores de recombinación. Los operadores de cruza se aplican de acuerdo a una probabilidad p_c, típicamente $p_c \in [0.5, 1.0]$. Generalmente, se seleccionan dos padres y se genera un número aleatorio r utilizando una distribución uniforme en el intervalo $[0, 1)$. Si $r < p_c$, se generan dos hijos utilizando recombinación, de lo contrario, se crean dos hijos asexualmente. Es decir, se hace una copia de los padres.

Recombinación de n puntos para representación binaria. El primer operador de este tipo fue propuesto por Holland [24] y es conocido como cruza de un punto. Posteriormente, fue generalizado para poder aplicarse

[2]Existen versiones en las que se utiliza una probabilidad para decidir si se elige al mejor o al peor individuo como padre.

con n puntos de cruza. Este operador realiza n cortes (puntos de cruza) en los cromosomas de dos padres. Los cortes se realizan aleatoriamente y son los mismos para ambos cromosomas. Para generar a los dos hijos se realiza lo siguiente: Se empieza a copiar la información del primer padre al primer hijo y la información del segundo padre al segundo hijo. Cuando se llega al primer punto de cruza se intercambian los padres. Es decir, la información del segundo padre se copia al primer hijo y la información del primer padre se copia al segundo hijo. Posteriormente, cada vez que se encuentra un punto de cruza se vuelven a intercambiar los padres.

Recombinación para representación real. Cuando estamos utilizando representación real, no se recomienda utilizar los operadores de cruza para representación binaria. Esto se debe a que no se pueden generar todos los valores posibles en el espacio de búsqueda. Por ejemplo, si utilizamos cruza de n puntos, los únicos valores que podrán tomar las variables son los contenidos en la población inicial. Existen varias propuestas de operadores de recombinación para representación real. En esta práctica solo mencionaremos la *cruza intermedia completa*, también conocida como *cruza aritmética completa*. Sean $\vec{x}$ y $\vec{y}$ los padres y α un valor en $(0, 1)$, obtenemos dos hijos de la siguiente forma:

$$\begin{aligned} \text{Hijo } 1 &= \alpha \cdot \vec{x} + (1 - \alpha) \cdot \vec{y} \\ \text{Hijo } 2 &= \alpha \cdot \vec{y} + (1 - \alpha) \cdot \vec{x} \end{aligned} \tag{5.4}$$

Recombinación para permutaciones. Cuando estamos trabajando con permutaciones, no es posible utilizar operadores de cruza para representación binaria porque generan cadenas de enteros con valores repetidos. Es decir, soluciones no válidas. Por esta razón, se han propuesto varios operadores de cruza para permutaciones. Uno de ellos es el operador llamado *Partially Mapped Crossover (PMX)* propuesto por Goldberg y Lingle [21]. PMX trabaja de la siguiente forma:

1. Elige aleatoriamente dos puntos de cruza.
2. Para el primer hijo toma el segmento del segundo padre que está entre los puntos de cruza y lo copia en las mismas posiciones.
3. Para el segundo hijo toma el segmento del primer padre que está entre los puntos de cruza y lo copia en las mismas posiciones.

4. Para el resto de las posiciones del primer hijo, va copiando los valores del primer padre. Si en una posición se tiene un valor que está contenido en el segmento que se copió del segundo padre, entonces se pone el valor por el cual fue intercambiado.

5. Para el segundo hijo hacemos lo mismo pero copiando la información del segundo padre.

Mutación

La mutación es el proceso por medio del cual se realizan pequeños cambios en el cromosoma de un individuo. Al igual que en la cruza existen diferentes operadores de mutación para cada una de las representaciones. A continuación mencionaremos algunos.

Mutación para representación binaria. El procedimiento de mutación más utilizado en cadenas binarias trabaja de la siguiente forma. Se define una probabilidad de mutación p_m, usualmente esta probabilidad es muy pequeña. Se recorre la cadena binaria posición por posición. En cada posición se genera un número aleatorio r entre 0 y 1, si $r < p_m$, se cambia el valor de la cadena en esa posición. Si la cadena binaria es de tamaño l, se espera que se realicen $l \cdot p_m$ cambios en la cadena.

Mutación para representación real. En este tipo de representación se distinguen dos procesos de mutación: *uniforme* y *no uniforme*. La *mutación uniforme* es similar al proceso de mutación en representación binaria. Cuando se decide cambiar un gen, se genera un número aleatorio en el intervalo de la variable que está representando dicho gen. Si la variable a mutar es x_i, la variable mutada es $x_i' \in [l_{inf}, l_{sup}]$. En la *mutación no uniforme*, se agrega un ruido a la variable que se desea mutar: $x_i' = x_i + \delta$. Una posible opción es hacer $\delta = N(0, \sigma)$ y se recomienda una probabilidad de 1 para mutar las variables. Es decir, todos los genes del individuo son mutados.

Mutación para permutaciones. Uno de los procesos de mutación más sencillos es la *mutación por inserción*. Aquí se elige un valor de la cadena de números enteros y se mueve a cualquier otra posición. Una generalización de este operador, es la *mutación por desplazamiento*. Aquí se mueven n valores de la cadena. Es decir, se aplica mutación por inserción n veces. Existen otras propuestas de mutación más elaboradas como la *mutación heurística*.

Este operador selecciona n posiciones de la cadena. Posteriormente, genera todas las permutaciones posibles considerando los valores de las n posiciones. Se crea una solución por permutación y se elige aquella que corresponda al mejor individuo.

Selección de sobrevivientes

Una vez que la población actual genera un conjunto de hijos, se debe decidir qué individuos formarán parte de la siguiente generación. Los primeros AGs utilizaban una *selección basada en la edad*. Por ejemplo, en el AG simple los individuos tienen un tiempo de vida de un ciclo. Es decir, los padres son reemplazados en cada iteración (generación) por sus hijos, sin importar su aptitud. Posterioremente, se utilizaron otros esquemas de selección como la selección $(\mu + \lambda)$ y la selección (μ, λ) que utilizan las estrategias evolutivas. O bien, una selección basada en edad pero que garantice conservar siempre al mejor individuo.

5.2. Función de Beale

Con la finalidad de ejemplificar el uso de un AG utilizaremos la función de Beale. Recordemos que la función de Beale es un problema de optimización continua de dos variables y está definida como sigue:

$$\text{mín } f(x_1, x_2) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2 \tag{5.5}$$

donde $-4.5 \leq x_1, x_2 \leq 4.5$. El mínimo global está en $x^* = (3, 0.5)$ y $f(x^*) = 0$.

Representación de una solución

Para resolver este problema vamos a utilizar representación binaria y una precisión de 3 dígitos. El número de bits requeridos por variable son:

$$\text{num\_bits} = \lceil \log_2((4.5 - (-4.5)) * 10^3) \rceil = 14$$

El cromosoma de cada invididuo será de longitud 28. Los primeros 14 bits corresponden a x_1 y los últimos 14 bits a x_2. Por ejemplo, si tenemos un individuo i con el siguiente cromosoma:

$$i = 01110100001000110101011001$$

Los valores reales x_1 y x_2 de i son:

$$x_1 = -4.5 + \frac{(9)(7432)}{2^{14} - 1} = -0.4172$$

$$x_2 = -4.5 + \frac{(9)(13657)}{2^{14} - 1} = 3.0024$$

Población inicial

Generamos un individuo de la siguiente forma: Para cada posición de la cadena binaria, generamos un número aleatorio r en el intervalo $(0, 1)$, utilizando una distribución uniforme. Si $r < 0.5$, ponemos un 0. De lo contrario, ponemos un 1. Repetimos el mismo proceso hasta generar la cantidad de individuos que se deseen. Para este ejemplo, utilizaremos una población de tamaño 100.

Aptitud

La aptitud de un individuo está relacionada con qué tan buena es la solución que está representando. Los valores de aptitud más grandes pertenecen a los mejores individuos. Dada la definición de la función de Beale, sabemos que $f(x_1, x_2) \geq 0$ y que el óptimo está en $f(3, 0.5) = 0$. Por lo anterior, podemos definir la aptitud de un individuo i como $F_a(i) = \frac{1}{f(x_1^{(i)}, x_2^{(i)})+1}$, donde f es la función de Beale.

- $F_a(01110100001000110101011001) = \frac{1}{F_a(-0.4172, 3.0024)+1}$
  $= \frac{1}{70.6879} = 0.0141$

Selección de padres

Para seleccionar a los individuos que serán padres utilizaremos el método de universal estocástica. Recordemos que este método considera la aptitud para poder hacer el cálculo de las probabilidades. Si se tienen valores de aptitud menores o iguales a cero, el método no funcionará.

Recombinación

Utilizaremos cruza de dos puntos y la probabilidad de cruza será $p_c = 0.9$. Es decir, para cada pareja de padres, generamos un número aleatorio $r \in (0, 1)$, utilizando una distribución uniforme. Si $r < p_c$, aplicamos cruza de dos puntos para generar dos soluciones hijas. De lo contrario, hacemos una copia de los padres que actuarán como soluciones hijas. Suponga que $r = 0.57$, que los puntos de cruza son 2 y 10, y que los individuos padres son:

$$i = 01|11010000|100011010101011001$$
$$j = 10|01010110|101000010001111011$$

donde el símbolo | denota un punto de cruza. Las soluciones hijas son:

$$i' = 01010101101000110101010110 0$$
$$j' = 10110100001010000100011110 11$$

Mutación

En este ejemplo, vamos a utilizar una probabilidad de mutación $p_m = 0.001$. Por cada solución hija, creada a partir de la cruza de dos padres, vamos a generar un número aleatorio $r \in (0, 1)$ para cada una de las 28 posiciones de su cadena binaria. Las posiciones en las que $r < 0.001$, serán modificadas. Imagine que para el individuo:

$$i' = 0101010110100011010101011 00$$

salió $r = 0.00018$ en la posición 15. El individuo i' mutado será:

$$i' = 0101010110100\mathbf{0}1010101011 00$$

Selección de sobrevivientes

Para decidir cuáles individuos formarán parte de la siguiente generación, vamos a utilizar una selección basada en la edad aplicando elitismo: En caso de que la peor solución hija tenga una aptitud menor que la mejor solución de la población actual, conservamos la mejor solución de la población actual y los $\mu - 1$ mejores hijos, donde $\mu = 100$ es el tamaño de la población. De lo contrario, las soluciones hijas van a reemplazar a toda la población actual.

Es importante mencionar que el diseño propuesto es un diseño simple de un AG. Si se revisa el número de copias que se están realizando de los mejores individuos, se observará que es un valor muy alto y esto puede provocar convergencia a soluciones no óptimas.

5.3. Ejercicios

(10 puntos) Implementar, en el lenguaje de su preferencia, el AG que se diseñó en el ejemplo anterior para resolver la función de Beale.

(10 puntos) Utilizar el AG del punto anterior pero ahora con representación real, cruza intermedia completa y mutación uniforme.

(10 puntos) Utilizar las dos versiones del AG que se tienen para resolver la función de Ackley.

(20 puntos) Realizar una tabla comparativa que reporte los resultados de ambas versiones en los dos problemas de prueba (función de Beale y función de Ackley). En el caso de la función de Ackley, resolver instancias con 5, 10 y 20 variables. Para construir la tabla comparativa debes realizar M ejecuciones de cada versión resolviendo cada uno de los problemas y reportar lo siguiente:

1. Mejor solución encontrada considerando las M ejecuciones.
2. Peor solución encontrada considerando las M ejecuciones.
3. Solución que corresponde a la mediana considerando las M ejecuciones.
4. Media del valor de la función objetivo considerando las M ejecuciones.
5. Desviación estándar del valor de la función objetivo considerando las M ejecuciones.

En los primeros tres puntos indica tanto el valor de x como el valor de la función objetivo f. Nota. Recuerde que puede variar los parámetros p_c, p_m y tamaño de la población para obtener mejores resultados. Indique los valores que utilizó para cada problema.

(40 puntos) Diseñar e implementar un AG que resuelva el problema del agente viajero.

- Dada una lista de ciudades y la distancia entre cada par de ellas, encontrar el camino más corto que visite cada ciudad exactamente una vez y regrese a la ciudad de origen.

La **entrada** al programa será:

1. La primera línea tendrá el número de ciudades N.
2. La segunda línea tendrá los parámetros del AG: probabilidad de cruza p_c, probabilidad de mutación p_m y tamaño de la población P.
3. Las siguientes $N-1$ líneas indicarán el costo de ir de una ciudad a otra. Es decir:
   - La primera línea tiene el costo de ir de la ciudad n_0 a la ciudad n_1, a la ciudad n_2 y hasta la ciudad n_{N-1}.
   - La segunda línea tiene el costo de ir de la ciudad n_1 a la ciudad n_2 y hasta la ciudad n_{N-1}.
   - Así sucesivamente hasta llegar a la última línea que tiene el costo de ir de la ciudad n_{N-2} a la ciudad n_{N-1}.

La **salida** del programa será:

1. Recorrido que debe seguir el viajero (permutación).
2. Costo de seguir el recorrido encontrado.

Consideraciones:

- Etiquete a las ciudades como 0, 1, 2, $\cdots$, $N-1$.
- El viajero siempre parte de la ciudad n_0.
- Tiene el mismo costo ir de la ciudad A a la ciudad B que ir de la ciudad B a la ciudad A.
- Una solución al problema es una permutación de las N ciudades.

A continuación se muestra un ejemplo:

Entrada:

```
10
0.9 0.01 100
49 30 53 72 19 76 87 45 48
19 38 32 31 75 69 61 25
41 98 56 6 6 45 53
52 29 46 90 23 98
63 90 69 50 82
60 88 41 95
61 92 10
82 73
5
```

Salida:

```
0 5 2 7 8 4 1 3 9 6
467
```

(10 puntos) Permitir al usuario realizar M ejecuciones del AG, implementado en el punto anterior, para resolver una instancia del problema del agente viajero. Dicha instancia estará almacenada en un archivo que tendrá los parámetros de entrada tal y como se indicó en el punto anterior. Después de las M ejecuciones se debe reportar lo siguiente:

1. Mejor solución encontrada considerando las M ejecuciones.
2. Peor solución encontrada considerando las M ejecuciones.
3. Solución que corresponde a la mediana considerando las M ejecuciones.
4. Media del valor de la función objetivo considerando las M ejecuciones.
5. Desviación estándar del valor de la función objetivo considerando las M ejecuciones.

En los primeros tres puntos indica tanto el valor de x (permutación) como el valor de la función objetivo f (costo de hacer el recorrido). *Nota*: Para este punto el usuario debe indicar el nombre del archivo de entrada y el número M de ejecuciones a realizar.

Bibliografía

[1] T. Back, M. Emmerich, and O. M. Shir. Evolutionary algorithms for real world applications [application notes]. *IEEE Computational Intelligence Magazine*, 3(1):64–67, 2008.

[2] Thomas Back. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, Oxford, UK, 1996.

[3] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies – a comprehensive introduction. *Natural Computing*, 1:3–52, 2002. An optional note.

[4] Alan W. Johnson Darrall Henderson, Sheldon H. Jacobson. *The Theory and Practice of Simulated Annealing*, volume 57, pages 287–319. Springer, Boston, MA, 2003.

[5] D. de Werra and A. Hertz. Tabu search techniques. *OR Spektrum*, 11:131–141, 1989.

[6] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2nd edition, 2015.

[7] Michael Emmerich, Ofer M. Shir, and Hao Wang. *Evolution Strategies*, pages 1–31. Springer International Publishing, Cham, 2018.

[8] D. B. Fogel and J.W. Atmar. Comparing genetic operators with gaussian mutations in simulated evolutionary processes using linear systems. *Biological Cybernetics*, 63:111–114, 1990.

[9] David Fogel. *Blondie24: Playing at the Edge of AI*. The Morgan Kaufmann Series in Artificial Intelligence, San Francisco, 01 2002.

[10] David B. Fogel and Lauren C. Stayton. On the effectiveness of crossover in simulated evolutionary optimization. *Biosystems*, 32(3):171 – 182, 1994.

[11] Lawrence J. Fogel. *Intelligence through Simulated Evolution: Forty Years of Evolutionary Programming*. John Wiley & Sons, Inc., USA, 1999.

[12] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution*. Wiley, Chichester, WS, UK, 1966.

[13] Michel Gendreau and Jean-Yves Potvin. *Tabu Search*, pages 165–186. Springer US, Boston, MA, 2005.

[14] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986. Applications of Integer Programming.

[15] Fred Glover. Tabu search - part i. *INFORMS Journal on Computing*, 2:4–32, 01 1990.

[16] Fred Glover. Tabu search—part ii. *ORSA Journal on Computing*, 2:4–32, 02 1990.

[17] Fred Glover. *Tabu Search and Adaptive Memory Programming*. Springer, Boston, MA, USA, 1997.

[18] Fred Glover and Manuel Laguna. *Tabu Search*, volume 7. Springer US, USA, 1997.

[19] Fred Glover and Manuel Laguna. *Tabu Search*, pages 2093–2229. Springer US, Boston, MA, 1998.

[20] Fred Glover and Eric Taillard. A user's guide to tabu search. *Annals of Operations Research*, 41:28–41, 1993.

[21] D. E. Goldberg and R. Lingle. Alleles, loci and the traveling salesman problem. In *Proceedings of the 1st International Conference on Genetic Algorithms and Their Applications*, 1985.

[22] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 312–317, 1996.

[23] A. Hertz and D. de Werra. The tabu search metaheuristic: How we used it. *Annals of Mathematics and Artificial Intelligence*, 1:111–121, 1990.

[24] John H. Holland. *Adaption in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[25] Wim Hordijk and Bernard Manderick. The usefulness of recombination. volume 929, pages 908–919, 06 1995.

[26] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[27] C Koulamas, SR Antony, and R Jaen. A survey of simulated annealing applications to operations research problems. *Omega*, 22(1):41 – 56, 1994.

[28] K. F. Man, K. S. Tang, and S. Kwong. Genetic algorithms: concepts and applications [in engineering design]. *IEEE Transactions on Industrial Electronics*, 43(5):519–534, 1996.

[29] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, and Augusta H. Teller. Equation of state calculations by fast computing machines. *Chemical Physics*, 21(6):1087–1092, 1953.

[30] Seyedali Mirjalili. *Genetic Algorithm*, pages 43–55. Springer, Cham, 2018.

[31] Elizabeth Montero, María-Cristina Riff, and Bertrand Neveu. A beginner's guide to tuning methods. *Applied Soft Computing*, 17:39–51, 2014.

[32] Ingo Rechenberg. *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. PhD thesis, Technical University of Berlin, 1971.

[33] Ingo Rechenberg. *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart : Frommann-Holzboog, 1973.

[34] Colin R. Reeves. Introduction to evolutionary computing. *INFORMS Journal on Computing*, 9(3), 1997.

[35] Colin R. Reeves. *Genetic Algorithm*, pages 109–139. Springer, Boston, MA, 2010.

[36] G. Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Transactions on Neural Networks*, 5(1):96–101, 1994.

[37] Hans-Paul Schwefel. Kybernetische evolution als strategie der exprimentellen forschung in der stromungstechnik. Master's thesis, Technical University of Berlin, 1965.

[38] Hans-Paul Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*. Birkhäuser Basel, 1977.

[39] Hans-Paul Schwefel. *Evolution and Optimum Seeking: The Sixth Generation*. John Wiley & Sons, Inc., USA, 1993.

[40] Adam Slowik and Halina Kwasnicka. Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications*, 32:12363–12379, 2020.

[41] M. Srinivas and L. M. Patnaik. Genetic algorithms: a survey. *Computer*, 27(6):17–26, 1994.

[42] K. S. Tang, K. F. Man, S. Kwong, and Q. He. Genetic algorithms and their applications. *IEEE Signal Processing Magazine*, 13(6):22–37, 1996.

Material complementario

Índice general

Capítulo 1

Búsqueda Tabú (BT)

1.1. Problema de la mochila binario

El problema de la mochila binario (0/1 Knapsack problem) considera un conjunto finito de n objetos, donde cada objeto i tiene un valor p_i y un peso w_i, y una mochila que puede soportar hasta un peso determinado c. El objetivo es encontrar el subconjunto de objetos que puedan ser transportados en la mochila, maximizando el valor de la mochila. Formalmente se define como sigue:

$$\begin{array}{rll} \text{maximizar:} & f(\vec{x}) = \sum_{i=1}^{n} p_i \cdot x_i & \\ \text{tal que} & g_1(\vec{x}) = \sum_{i=1}^{n} w_i \cdot x_i \leq c & \\ & x_i \in \{0,1\} & i \in \{1,\ldots,n\} \end{array} \tag{1.1}$$

Para este ejemplo, consideremos que se tienen $n = 5$ objetos. Donde cada objeto tiene los siguientes valores $p = [5, 14, 7, 2, 23]$ y los siguientes pesos $w = [2, 3, 7, 5, 10]$. Además la mochila tiene una capacidad $c = 15$.

1.1.1. Principales componentes de BT

Librerías de **Python** que vamos a utilizar.

```
In [69]: import numpy
         import math
```

Datos de entrada.

```
In [70]: n=5
         p = [5,14,7,2,23]
         w = [2,3,7,5,10]
         c = 15
```

Representación de una solución

Cada solución es una cadena binaria de tamaño 5. Para facilitar la implementación en **Python**, lo tomaremos como una lista de 0's y 1's. Por ejemplo, la solución x='11001' es almacenada como:

```
In [71]: x = [1,1,0,0,1]
```

Solución inicial

Mientras no se exceda la capacidad de la mochila, se van introduciendo de manera aleatoria objetos a la mochila.

```
In [72]: def getInitialSolution(n, p, w, c):
             #Ningún objeto está en la mochila
             x = [0 for i in range(n)]
             weight_x = 0

             #Aleatoriamente elegimos el orden en el que intentaremos
             #introducir los objetos a la mochila
             objects = list(range(n))
             numpy.random.shuffle(objects)

             for o in objects:
                 #Intentamos introducir el objeto "o" a la mochila
                 if weight_x + w[o] <= c:
                     x[o] = 1
                     weight_x += w[o]

             return x

In [73]: x_0 = getInitialSolution(n, p, w, c)
         x_0

Out[73]: [1, 1, 0, 0, 1]
```

Función objetivo

```
In [74]: def f(p, x):
             profits = 0
             for i in range(len(x)):
                 profits += p[i]*x[i]

             return profits

In [75]: print ("f(x): \t", f(p, x))
         print ("f(x_0): ", f(p, x_0))
```

```
f(x): 42
f(x_0): 42
```

Restricción

```
In [76]: def g1(w, x):
             weight = 0
             for i in range(len(x)):
                 weight += w[i]*x[i]

             return weight
```

```
In [77]: print ("g1(w, x): \t", g1(w, x))
         print ("g1(w, x_o): \t", g1(w, x_0))
```

```
g1(w, x): 15
g1(w, x_o): 15
```

Movimiento

Altera el valor del elemento ubicado en la posición i de nuestra solución x.

```
In [78]: def bitflip(x, i):
             x_new = x.copy()
             if x[i] == 0:
                 x_new[i] = 1
             else:
                 x_new[i] = 0

             return x_new
```

```
In [79]: print("x: \t", x)
         x_new = bitflip(x, 2)
         print("x_new: \t", x_new)
```

```
x: [1, 1, 0, 0, 1]
x_new: [1, 1, 1, 0, 1]
```

Vecindario

El vecindario está definido como movimientos "bitflip" en cada una de las variables de la solución, teniendo en cuenta que solamente las soluciones factibles pueden ser parte del vecindario. En **Python** vamos a implementar el vecindario como una lista de listas, donde cada lista interna almacena una solución y su valor tanto de la función objetivo como de la restricción.

```
In [80]: def getNeighborhood(p, w, x, c):
             neighborhood = []
             for i in range(len(x)):
                 x_new = bitflip(x, i)
                 g1_x_new = g1(w, x_new)
                 #Si la solución creada es factible,
                 #la metemos al vecindario
                 if g1_x_new <= c:
                     neighborhood.append([x_new, f(p, x_new), g1_x_new, i])

             return neighborhood

In [81]: print("Vecindario de ", x)
         getNeighborhood(p, w, x, c)

Vecindario de  [1, 1, 0, 0, 1]


Out[81]: [[[0, 1, 0, 0, 1], 37, 13, 0],
          [[1, 0, 0, 0, 1], 28, 12, 1],
          [[1, 1, 0, 0, 0], 19, 5, 4]]
```

Es importante mencionar que para instancias grandes del problema (n's grandes) se debe cuidar la eficiencia. Por ejemplo, no es necesario recalcular completamente los valores de la función objetivo y la restricción de cada solución nueva se puede obtener a partir de la solución base y el cambio realizado.

Lista tabú

La lista tabú va a almacenar los valores que no son permitidos en ciertas posiciones de la solución y su "tiempo tabú". En **Python** utilizaremos un diccionario para guardar los elementos tabú. La llave del diccionario será la posición p y va estar asociada con el valor v que está prohibido en esa posición y con su tiempo tabú t. El siguiente ejemplo indica que en la posición 1 no puede haber un valor de 0 durante las próximas 2 iteraciones.

```
In [82]: tabu_list = {}
         tabu_list[1] = [0, 2]
         print (tabu_list)

{1: [0, 2]}
```

La lista tabú se va a modificar cada iteración actualizando el tiempo tabú de los elementos que ya se encontraban en ella y añadiendo un nuevo elemento.

```
In [83]: def updateTabuList(new_element, tabu_list):
             aux = []

             #Disminuimos en 1 el tiempo tabú de los elementos
             #que ya estaban en la lista tabú
             for key in tabu_list:
                 tabu_list[key][1] -= 1
                 if tabu_list[key][1] == 0:
                     aux.append(key)

             #Sacamos de la lista tabú los elementos con
             #tiempo tabú igual con 0
             for key in aux:
                 tabu_list.pop(key)

             #Agregamos el nuevo elemento a la lista tabú
             tabu_list[new_element[0]] = [new_element[1], new_element[2]]
```

```
In [84]: print(tabu_list)
         tabu_element = [3,1,2]
         updateTabuList(tabu_element, tabu_list)
         print(tabu_list)
         tabu_element = [2,0,2]
         updateTabuList(tabu_element, tabu_list)
         print(tabu_list)
```

```
{1: [0, 2]}
{1: [0, 1], 3: [1, 2]}
{3: [1, 1], 2: [0, 2]}
```

Vecindario reducido

El vecindario reducido se define como el conjunto de soluciones resultante de quitar del vecindario las soluciones generadas a partir de movimientos que se encuentran en la lista tabú.

```
In [85]: def getReducedNeighborhood(x, tabu_list, p, w, c):
             neighborhood = []
             for i in range(len(x)):
             #Si no está prohibido cambiar el valor de la posición i,
             #creamos una nueva solución haciendo bitflip en la posición i
                 if i not in tabu_list:
                     x_new = bitflip(x, i)
                     g1_x_new = g1(w, x_new)
                     #Si la solución creada es factible,
```

```
                #la metemos al vecindario
                if g1_x_new <= c:
                    neighborhood.append([x_new, f(p, x_new), g1_x_new, i])

        return neighborhood
```

1.1.2. Algoritmo completo de una BT simple

```
In [86]: #Obtenemos el índice de la mejor solución en el vecindario
         def getIndexBestNeighbor(neighborhood):
             best = 0
             for i in range(1, len(neighborhood)):
                 if neighborhood[i][1] >= neighborhood[best][1]:
                     best = i

             return best
```

```
In [87]: #Búsqueda tabú simple para el problema de la mochila binario
         def TabuSearch(num_ite, n, p, w, c):
             #Obtenemos la solución inicial
             x = getInitialSolution(n, p, w, c)
             f_x = f(p, x)
             g_x = g1(w, x)

             #Hasta ahora la solución inicial es la mejor
             #solución que se conoce
             x_best, f_best, g_best = x.copy(), f_x, g_x

             #Iniciamos con nuestra lista tabú vacía
             tabu_list = {}

             #Definimos el tiempo tabú
             tabu_time = n//2

             for k in range(num_ite):
                 #Obtenemos nuestro vecindario reducido
                 neighborhood =  getReducedNeighborhood(x, tabu_list, p, w, c)
                 #Si nuestro vecindario está vacío, ya no podemos movernos
                 #a otra solución y debemos terminar la búsqueda
                 if len(neighborhood) == 0:
                     break

                 #Nuestra siguiente solución es la mejor del vecindario
                 best_neighbor = getIndexBestNeighbor(neighborhood)
                 x = neighborhood[best_neighbor][0]
```

```
                f_x = neighborhood[best_neighbor][1]
                g_x = neighborhood[best_neighbor][2]

                #Verificamos si la nueva solución
                #es mejor que lo que conocemos
                if f_x >= f_best:
                    x_best, f_best, g_best = x.copy(), f_x, g_x

                #Actualizamos la lista tabú
                i = neighborhood[best_neighbor][3]
                tabu_element = [i, x[i], tabu_time]
                updateTabuList(tabu_element, tabu_list)

            return x_best, f_best, g_best


In [88]: def printSolution(x, f, g):
            print("f(x) = ", f, "\tg(x) = ", g)
            step = 25
            print("x = \n[", end="")
            for i in range(len(x)-1):
                if i != 0 and i % step == 0:
                    print()
                print(x[i], end=', ')
            print(x[-1], "]")


In [89]: n=5
         p = [5,14,7,2,23]
         w = [2,3,7,5,10]
         c = 15
         num_ite = 50

         x_best, f_best, g_best = TabuSearch(num_ite, n, p, w, c)
         printSolution(x_best, f_best, g_best)

f(x) =  42          g(x) =  15
x = [1, 1, 0, 0, 1 ]
```

Ejecutamos para una instancia aleatoria más grande del problema.

```
In [90]: n=50
         profits = list(range(30, 100))
         weights = list(range(20, 40))
```

```
        p = numpy.random.choice(profits, n)
        print("Arreglo con valores de cada objeto: \n", p)
        w = numpy.random.choice(weights, n)
        print("Arreglo con pesos de cada objeto: \n", w)
        c = numpy.random.randint((30*n)//2, 30*n)
        print("Capacidad de la mochila: ", c)
        num_ite = 1000

        print("Ejecución 1:")
        x_best, f_best, g_best = TabuSearch(num_ite, n, p, w, c)
        printSolution(x_best, f_best, g_best)
```

```
Arreglo con valores de cada objeto:
 [47 59 37 63 68 71 99 38 99 38 78 33 89 63 74 92 32 78 31 89 73 92 38 30
 38 30 66 68 45 30 72 73 41 51 88 65 85 76 65 40 67 62 96 68 30 81 67 42
 53 97]
Arreglo con pesos de cada objeto:
 [38 24 35 34 37 35 29 24 35 34 32 37 34 37 33 34 22 39 36 32 38 26 24 29
 20 32 26 39 29 22 22 33 22 24 21 23 22 32 39 38 38 37 34 27 34 22 29 37
 30 31]
Capacidad de la mochila:  1337
Ejecución 1:
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
```

Realizamos m ejecuciones de nuestro algoritmo y obtenemos la corrida que encontró la mejor solución, la corrida que encontró la peor solución, la media y la desviación estándar de las m corridas con respecto al valor de la función objetivo.

```
In [94]: m=21
         sol = []
         for i in range(m):
             print("Ejecución ", i, ": ")
             x_best, f_best, g_best = TabuSearch(num_ite, n, p, w, c)
             sol.append([f_best, g_best, x_best])
             printSolution(x_best, f_best, g_best)

         sol.sort()
         print("*************** Mejor solución ***************")
         printSolution(sol[-1][2], sol[-1][0], sol[-1][1])

         print("*************** Peor solución ****************")
```

```
        printSolution(sol[0][2], sol[0][0], sol[0][1])

        print("****************** Mediana ******************")
        med = m//2
        printSolution(sol[med][2], sol[med][0], sol[med][1])

        f_sol = [x[0] for x in sol]
        print("Media: ", numpy.mean(f_sol))
        print("Desviación estándar: ", numpy.std(f_sol))
```

```
Ejecución  0 :
f(x) =  2911         g(x) =  1336
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1 ]
Ejecución  1 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  2 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  3 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  4 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  5 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  6 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
```

```
Ejecución  7 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  8 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  9 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  10 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  11 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  12 :
f(x) =  2886         g(x) =  1316
x =
[1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  13 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  14 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  15 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
```

```
Ejecución  16 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  17 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  18 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  19 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  20 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
*************** Mejor solución ***************
f(x) =  2911         g(x) =  1336
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1 ]
*************** Peor solución ****************
f(x) =  2886         g(x) =  1316
x =
[1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
****************** Mediana *******************
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Media:  2891.714285714286
Desviación estándar:  4.4416090728994355
```

Capítulo 2

Recocido Simulado (RS)

2.1. Problema del agente viajero

El problema del agente viajero (Travelling salesman problem) considera un conjunto finito de n ciudades y la distancia entre cada par de ellas. El objetivo es encontrar el camino más corto que visite cada ciudad exactamente una vez y regrese a la ciudad de origen. Formalmente se define como sigue:

$$\begin{array}{rl} \text{minimizar:} & f(x) = d(x_n, x_1) + \sum_{i=1}^{n-1} d(x_i, x_{i+1}) \\ \text{tal que} & x_i \in \{1, 2, \cdots, n\} \end{array} \tag{2.1}$$

donde $d(x_i, x_j)$ es la distancia de ir de la ciudad x_i a la ciudad x_j, n es el número de ciudades y x es una permutación de las n ciudades.

2.1.1. Principales componentes de RS

Librerías de **Python** que vamos a utilizar.

```
In [81]: import numpy
         import math
```

Datos de entrada.

```
In [82]: n = 5
         dist_matrix = [
                         [0,49,30,53,72],
                         [49,0,19,38,32],
                         [30,19,0,41,98],
                         [53,38,41,0,52],
                         [72,32,98,52,0],
                        ]
```

Representación de una solución

Cada solución es una permutación de las n ciudades. En **Python** será una lista de n elementos. Nombraremos a las ciudades como 0, 1, $\cdots$, $n-1$ para que coincidan con los índices de nuestra lista. El primer elemento siempre será 0 porque es la ciudad de partida.

```
In [83]: x = [0,4,2,1,3]
```

Solución inicial

Utilizaremos una estrategia voraz: Empezamos en la ciudad 0, posteriormente revisamos las 4 ciudades restantes y elegimos la que tenga una distancia menor a la ciudad 0. Repitemos el proceso hasta tener nuestra permutación.

```
In [84]: def getNextCity(x, dist_matrix, n):
             current_city = x[-1]
             min_dist = max(dist_matrix[current_city])

             for i in range(1, n):
                 if (i not in x) and (dist_matrix[current_city][i] < min_dist):
                     min_city = i
                     min_dist = dist_matrix[current_city][i]

             return min_city

         def getInitialSolution(n, dist_matrix):
             x = [0]

             while len(x) != n:
                 x.append(getNextCity(x, dist_matrix, n))

             return x

In [85]: x_0 = getInitialSolution(n, dist_matrix)
         print ("x_0: ", x_0)

x_0:  [0, 2, 1, 4, 3]
```

Función objetivo

```
In [86]: def f(n, dist_matrix, x):
             cost = dist_matrix[x[-1]][0]

             for i in range(n-1):
```

```
                cost += dist_matrix[x[i]][x[i+1]]

            return cost
```

```
In [87]: print("f(x)\t= ", f(n, dist_matrix, x))
         print("f(x_0)\t= ", f(n, dist_matrix, x_0))
```

```
f(x)        =  280
f(x_0)        =  186
```

Vecindario

El vecindario se genera eligiendo aleatoriamente una posición i de la permutación. Posteriormente, se generan $N-2$ soluciones moviendo la ciudad que está en la posición p a cualquiera de las otras posiciones posibles. En **Python** utilizaremos una lista de listas para almacenar el vecindario. Cada lista va a tener una permutación y su valor en la función objetivo.

```
In [88]: def getNeighborhood(x, n, dist_matrix):
             aux_x = x.copy()
             #Elegimos una posición aleatoria
             i = numpy.random.randint(1, len(x))
             print ("Posición aleatoria: ", i)
             city = aux_x.pop(i)

             neighborhood = []
             for j in range(1, len(x)):
                 if j != i:
                     new_sol = aux_x.copy()
                     new_sol.insert(j, city)
                     neighborhood.append([new_sol, f(n, dist_matrix, new_sol)])

             return neighborhood
```

```
In [90]: print ("Solución actual: ", x_0)
         neighborhood = getNeighborhood(x_0, n, dist_matrix)
         print ("Vecindario:")
         for e in neighborhood:
             print(e)
```

```
Solución actual:  [0, 2, 1, 4, 3]
Posición aleatoria:  4
Vecindario:
[[0, 3, 2, 1, 4], 217]
[[0, 2, 3, 1, 4], 213]
```

```
[[0, 2, 1, 3, 4], 211]
```

Dado que elegiremos de manera aleatoria una solución del vecindario, no es necesario generar todo el vecindario. Por lo tanto, crearemos una función que nos permita obtener una única solución.

```
In [91]: def nextSolution(x, n, dist_matrix):
             aux_x = x.copy()
             #Elegimos una posición aleatoria
             i = numpy.random.randint(1, len(x))
             #print ("Posición aleatoria: ", i)
             city = aux_x.pop(i)

             #Elegimos una nueva posición
             j = numpy.random.randint(1, len(x))
             while i == j:
                 j = numpy.random.randint(1, len(x))

             aux_x.insert(j, city)

             return aux_x, f(n, dist_matrix, aux_x)
```

Temperatura

Para este ejemplo vamos a utilizar una función lineal para disminuir la temperatura.

```
In [92]: def updateTemperature(t):
             return 0.9*t
```

```
In [93]: t = 1000
         print ("Temperatura: ", t)
         new_t = updateTemperature(t)
         print ("Temperatura: ", new_t)
```

```
Temperatura:  1000
Temperatura:  900.0
```

2.1.2. Algoritmo completo de un RS simple

```
In [94]: #Recocido simulado simple para el problema del agente viajero
         def SimulatedAnnealing(t_0, t_f, n, dist_matrix):
             x_0 = getInitialSolution(n, dist_matrix)
             x = x_0
             fx = f(n, dist_matrix, x)
```

```
            t = t_0
            x_best = x.copy()
            f_best = fx

            print("Solución inicial: ", x, fx)
            while t >= t_f:
                new_x, new_f = nextSolution(x, n, dist_matrix)

                if new_f <= f_best:
                    x_best = new_x.copy()
                    f_best = new_f

                if new_f < fx or numpy.random.random() < math.exp(-1.0*(new_f-fx)/t):
                    x = new_x
                    fx = new_f

                t = updateTemperature(t)
                #print(x, fx)

            return x_best, f_best
```

Ejecutamos el algoritmo.

```
In [99]: n = 5
         dist_matrix = \
         [\
         [0,49,30,53,72],\
         [49,0,19,38,32],\
         [30,19,0,41,98],\
         [53,38,41,0,52],\
         [72,32,98,52,0],\
         ]

         x, fx = SimulatedAnnealing(10000, 0.1, n, dist_matrix)
         print("Solución encontrada: ", x, fx)
```

```
Solución inicial:  [0, 2, 1, 4, 3] 186
Solución encontrada:  [0, 2, 1, 4, 3] 186
```

Ejecutamos para una instancia más grande del problema.

```
In [102]: n = 10
          dist_matrix = \
          [\
          [0,49,30,53,72,19,76,87,45,48],\
```

```
            [49,0,19,38,32,31,75,69,61,25],\
            [30,19,0,41,98,56,6,6,45,53],\
            [53,38,41,0,52,29,46,90,23,98],\
            [72,32,98,52,0,63,90,69,50,82],\
            [19,31,56,29,63,0,60,88,41,95],\
            [76,75,6,46,90,60,0,61,92,10],\
            [87,69,6,90,69,88,61,0,82,73],\
            [45,61,45,23,50,41,92,82,0,5],\
            [48,25,53,98,82,95,10,73,5,0],\
            ]

            x, fx = SimulatedAnnealing(100000, 0.01, n, dist_matrix)
            print("Solución encontrada ", x, fx)
```

```
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Solución encontrada  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
```

Realizamos m ejecuciones de nuestro algoritmo y obtenemos la corrida que encontró la mejor solución, la corrida que encontró la peor solución, la media y la desviación estándar de las m corridas con respecto al valor de la función objetivo.

```
In [103]: m=21
          t_0 = 100000
          t_f = 0.01
          sol = []
          for i in range(m):
              print("Ejecución ", i, ": ")
              x_best, f_best, = SimulatedAnnealing(t_0, t_f, n, dist_matrix)
              sol.append([f_best, x_best])
              print(x_best, f_best)

          sol.sort()
          print("*************** Mejor solución ***************")
          print(sol[0][0], sol[0][1])

          print("*************** Peor solución ****************")
          print(sol[-1][0], sol[-1][1])

          print("****************** Mediana *******************")
          med = m//2
          print(sol[med][0], sol[med][1])

          f_sol = [x[0] for x in sol]
          print("Media: ", numpy.mean(f_sol))
          print("Desviación estándar: ", numpy.std(f_sol))
```

```
Ejecución  0 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  1 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  2 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  3 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  4 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  5 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  6 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  7 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  8 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 1, 4, 7, 2, 6, 9, 8, 3, 5] 248
Ejecución  9 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  10 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  11 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  12 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  13 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  14 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
```

```
Ejecución  15 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  16 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  17 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  18 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  19 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  20 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 7, 2, 6, 9, 8, 3, 4, 1, 5] 271
*************** Mejor solución ***************
248 [0, 1, 4, 7, 2, 6, 9, 8, 3, 5]
*************** Peor solución ****************
271 [0, 7, 2, 6, 9, 8, 3, 4, 1, 5]
****************** Mediana *******************
271 [0, 5, 3, 8, 9, 6, 2, 7, 1, 4]
Media:  269.9047619047619
Desviación estándar:  4.89805366499954
```

Recordemos que esta es la versión más simple de un RS, se pueden hacer cambios en el diseño para mejorar la eficacia del algoritmo.

Capítulo 3

Programación Evolutiva (PE)

3.1. Función de Beale

La función de Beale es una función de dos variables y está definida como sigue:

$$\text{mín } f(x_1, x_2) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2 \quad (3.1)$$

donde $-4.5 \leq x_1, x_2 \leq 4.5$. El mínimo global está en $x^* = (3, 0.5)$ y $f(x^*) = 0$.

3.1.1. Principales componentes de PE

Librería de **Python** que vamos a utilizar.

```
In [1]: import numpy as np
```

Datos de entrada.

```
In [2]: n = 2 #Número de variables de decisión
```

Representación de un individuo

Un arreglo de valores flotantes de tamaño 4. Las primeras dos componentes corresponden a las dos variables del problema. Las siguentes dos componentes corresponden a los tamaños de paso para la mutación de cada una de las variables del problema.

```
In [3]: i = np.array([-1.5, 2.8, 0.3, 0.15])
```

Aptitud

La aptitud de un individuo i la definimos como $F_a(i) = -f(x_1^{(i)}, x_2^{(i)})$, donde f es la función de Beale.

```
In [4]: def f(x):
            x1, x2 = x[0], x[1]
            term1 = (1.5 - x1 + x1*x2)**2
            term2 = (2.25 - x1 + x1*(x2**2))**2
            term3 = (2.625 - x1 + x1*(x2**3))**2

            return term1 + term2 + term3

        def fitness(x):
            return -f(x)

In [5]: print ("x = (%s, %s)\t-f(x) = %s\t"%(i[0], i[1], fitness(i[:2])))

x = (-1.5, 2.8)         -f(x) = -895.2129089999994
```

Población inicial

Por cada individuo requerido, se utiliza una distribución uniforme para generar números aleatorios. Primero se generan dos números en el intervalo $[-4.5, 4.5]$ para crear las variables de decisión. Posteriormente, se generan dos números en el intervalo $(0, 1)$ para crear los tamaños de paso.

```
In [6]: def getInitialPopulation(mu, n):
            parents = []
            for i in range(mu):
                #Generamos un individuo
                p = np.concatenate((np.random.uniform(-4.5, 4.5, n),
                np.random.uniform(0, 1, n)))
                #Calculamos la aptitud del individuo
                p = [p, fitness(p[:n])]
                #Agregamos al individuo a la población "parents"
                parents.append(p)

            #Regresamos la población generada
            return parents

In [7]: mu = 100 #Tamaño de la población
        population = getInitialPopulation(mu, n)
        print("Tamaño de la población:", len(population))
        print("Primer individuo generado: ", population[0])

Tamaño de la población: 100
Primer individuo generado:
[array([-3.57073909,  4.48033254,  0.04512549,  0.43092328]), -103643.736219691]
```

Mutación y autoadaptación

Para mutar al individuo, se necesita primero mutar los valores de los tamaño de paso de la siguiente forma:

$$\sigma_1' = \sigma_1 \cdot (1 + \alpha \cdot N(0,1))$$
$$\sigma_2' = \sigma_2 \cdot (1 + \alpha \cdot N(0,1))$$

verificamos que los valores de σ_1' y σ_2' no sean menores que ε_0, en caso de que sí, se dejarán en ε_0. Posteriormente, utilizamos los nuevos σ para mutar las variables de decisión como sigue:

$$x_1' = x_1 + \sigma_1' \cdot N(0,1)$$
$$x_2' = x_2 + \sigma_2' \cdot N(0,1)$$

Finalmente, construimos el individuo hijo:

$$i' = [x_1', x_2', \sigma_1', \sigma_2']$$

Cuando se mutan las variables de decisión, se debe revisar que estén dentro del rango que indica el problema. En caso de que no, se le asignará el valor del límite que rebasó.

```
In [10]: def mutation(i, n, alpha, epsilon):
             #Mutamos los últimos dos componentes de nuestro arreglo
             #(tamaños de paso)
             mutation_sigma = i[n:]*(1+(alpha*np.random.normal(0, 1, n)))
             #Verificamos que los nuevos valores no sean menores a épsilon
             mutation_sigma[mutation_sigma < epsilon] = epsilon

             #Mutamos las variables de decisión a partir de las mutaciones
             mutation_x = i[:n] + (mutation_sigma*np.random.normal(0, 1, n))

             #Revisamos que estén dentro de los límites
             mutation_x[mutation_x < -4.5] = -4.5
             mutation_x[mutation_x >  4.5] =  4.5

             #Creamos el nuevo Individuo y lo devolvemos
             return [np.concatenate((mutation_x, mutation_sigma)),
             fitness(mutation_x)]
```

```
In [11]: #Parámetros para la mutación
         alpha = 0.2
         epsilon = 0.01

         child = mutation(population[0][0], n, alpha, epsilon)
         print(child)
```

```
[array([-3.46492722,  3.83356143,  0.06230124,  0.43211314]), -37879.09011226479]
```

3.1.2. Algoritmo completo de PE simple

A partir de 100 padres, vamos a generar 100 hijos. Unimos ambas poblaciones y seleccionamos los 100 mejores individuos de acuerdo a su aptitud.

```
In [12]: def EvolutionaryProgramming(n, mu, G, alpha, epsilon):
             parents = getInitialPopulation(mu, n)

             for t in range(num_gen):
                 new_gen = parents.copy()

                 for parent in parents:
                     #Creamos un hijo
                     child = mutation(parent[0], n, alpha, epsilon)
                     #Agregamos al hijo a la nueva generación
                     new_gen.append(child)

                 #Ordenamos del peor individuo al mejor individuo
                 #(del menor valor de aptitud al mayor)
                 new_gen = sorted(new_gen, key=lambda individual: individual[-1])
                 #Nos quedamos con los mu mejores individuos
                 new_gen = new_gen[mu:]

                 parents = new_gen.copy()

             #devolvemos el mejor
             return parents[-1][0], -parents[-1][1]
```

```
In [13]: n = 2
         mu = 100
         G = 200
         alpha = 0.2
         epsilon = .01
```

```
In [14]: sol, fx = EvolutionaryProgramming(n, mu, G, alpha, epsilon)
         print(f"x: {sol[:n]}\nsigma: {sol[n:]}\nf(x): {fx}")
```

```
x: [2.99992434 0.50001577]
sigma: [0.01       0.01466159]
f(x): 2.8418807729833617e-08
```

Realizamos m ejecuciones de nuestro algoritmo y obtenemos la corrida que encontró la mejor solución, la corrida que encontró la peor solución, la corrida que se encuentra en la mediana de las m corridas, la media y la desviación estándar de las m corridas con respecto al valor de la función objetivo.

```
In [15]: m = 21
         sol =[]
         for m in range(m):
             x, fx = EvolutionaryProgramming(n, mu, num_gen, alpha, epsilon)
             sol.append([x, fx])

         sol = sorted(sol, key=lambda individuo: individuo[-1])
         print("*************** Mejor solución ***************")
         print(f"x: {sol[0][0][:n]}\nsigma: {sol[0][0][n:]}\n
         f(x): {sol[0][-1]}")

         print("\n*************** Peor solución ****************")
         print(f"x: {sol[-1][0][:n]}\nsigma: {sol[-1][0][n:]}\n
         f(x): {sol[-1][-1]}")

         print("\n****************** Mediana *******************")
         print(f"x: {sol[m//2][0][:n]}\nsigma: {sol[m//2][0][n:]}\n
         f(x): {sol[m//2][-1]}")

         f_sol = [x[-1] for x in sol]
         print("\nMedia: ", np.mean(f_sol))
         print("\nDesviación estándar: ", np.std(f_sol))
```

```
*************** Mejor solución ***************
x: [2.99971315 0.49991913]
sigma: [0.02772399 0.01081308]
f(x): 1.5359162146571287e-08

*************** Peor solución ****************
x: [2.999368   0.49990788]
sigma: [0.03042325 0.02163403]
f(x): 1.6027470532192182e-07

****************** Mediana *******************
x: [2.99950765 0.49988047]
sigma: [0.01059131 0.01625244]
f(x): 3.897568350940374e-08

Media:  5.586687828982277e-08

Desviación estándar:  3.49564794851159e-08
```

Recordemos que esta es la versión más simple de PE, se pueden hacer cambios en el diseño para mejorar la eficacia del algoritmo.

Capítulo 4

Estrategias Evolutivas (EE)

4.1. Función de Ackley

La función de Ackley es una función de n variables y está definida como sigue:

$$\text{mín } f(\vec{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n}x_i^2}\right) - exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e \qquad (4.1)$$

donde $30 \leq x_i \leq 30$. El mínimo global está en $x_i = 0$ y $f(\vec{x}) = 0$.

4.1.1. Principales componentes de EE

Librerías de **Python** que vamos a utilizar.

```
In [3]: import numpy as np
        from copy import copy
        from math import e, exp, sqrt, pi, cos
        from random import choices
```

Datos de entrada.

```
In [4]: n = 2 #Número de variables de decisión
```

Representación de un individuo

Cada individuo será visto como un arreglo de valores flotantes de tamaño $n + 1$. Las primeras n componentes corresponden a las variables de decisión del problema y la siguiente componente corresponde al tamaño de paso para la mutación de las variables del problema.

```
In [5]: #Si n = 2
        i = np.array([-10.3, 7.84, 0.84])
        i

Out[5]: array([-10.3 ,   7.84,   0.84])
```

Aptitud

La aptitud de un individuo i la definimos como $F_a(i) = -f(x_1^{(i)}, x_2^{(i)})$, donde f es la función de Ackley.

```
In [7]: def f(x):
            part1 = 20*exp(-0.2*sqrt(np.average(x**2)))
            part2 = exp(np.average(np.array([cos(i) for i in  2*pi*x])))
            return -part1 - part2 + 20 + e

        def fitness(x):
            return -f(x)
```

```
In [8]: print ("x = (%s, %s)\t-f(x) = %s\t"%(i[0], i[1], fitness(i[:2])))
```

```
x = (-10.3, 7.84)        -f(x) = -18.39186319125107
```

Población inicial

Por cada individuo requerido, se utiliza una distribución uniforme para generar números aleatorios. Primero se generan n números en el intervalo $[-30, 30]$, cada uno corresponde a una variable de decisión. Posteriormente, un número aleatorio en el intervalo $(0, 1)$ que corresponde al tamaño de paso.

```
In [15]: def getInitialPopulation(mu, n):
             population = []
             for i in range(mu):
                 #Generamos un individuo
                 p = np.concatenate((np.random.uniform(-30, 30, n),
                 np.random.uniform(0, 1, 1)))
                 #Calculamos la aptitud del individuo
                 p = [p, fitness(p[:n])]
                 #Agregamos al individuo a la población "parents"
                 population.append(p)

             return population
```

```
In [16]: mu = 100 #Tamaño de la población
         population = getInitialPopulation(mu, n)
         population[0]
```

```
Out[16]: [array([-18.65372343,  -9.19726163,   0.87925505]),
-20.776871049837613]
```

Cruza

Recombinación discreta local. Utilizando una distribución uniforme, generamos un número aleatorio en el intervalo $(0,1)$ por cada componente de nuestro individuo. Recordemos que las primeras n componentes del individuo corresponden a sus variables de decisión y la última componente a su tamaño de paso. Si el valor es menor a 0.5, el hijo se queda con el valor del primer padre. De lo contrario, el hijo se queda con el valor del segundo padre.

```
In [18]: def localDiscreteCrossover(parent_1, parent_2):
             N = len(parent_1)
             #Creamos un vector con N componentes en el intervalo
             # (0,1) con distribución uniforme
             z = np.random.uniform(0, 1, N)
             child = np.array([0.0]*N)
             #Asignamos el valor del padre correspondiente
             for i in range(N):
                 if z[i] < 0.5:
                     child[i] = parent_1[i]
                 else:
                     child[i] = parent_2[i]

             #Regresamos el individuo generado
             return child
```

```
In [21]: i1 = [-10.3, 7.84, 0.84]
         i2 = [2.4, -3.84, 0.98]
         new_i = localDiscreteCrossover(i1,i2)
         print(new_i)
```

```
[-10.3   -3.84   0.84]
```

Mutación

Para mutar un individuo, se necesita mutar primero el tamaño de paso:

$$\sigma' = \sigma \cdot e^{\tau \cdot N(0,1)}$$

Si el valor de σ' es menor a ε_0, hacemos que $\sigma' = \varepsilon_0$. Posteriormente, utilizamos σ' para mutar las variables de decisión. Si $n = 2$, nuestras variables de decisión mutadas quedan de la siguiente forma:

$$x_1' = x_1 + \sigma' \cdot N(0,1)$$
$$x_2' = x_2 + \sigma' \cdot N(0,1)$$

Revisamos que las variables de decisión generadas estén dentro del rango que indica el problema. En caso de que no, se les asigna el valor del límite que rebasó. Finalmente, construimos el individuo hijo:

$$i' = [x_1', x_2', \sigma']$$

```
In [23]: def mutation(i, n, epsilon, tau):
             #Mutamos el tamaño de paso
             mutation_sigma = i[-1]*(e**(tau*np.random.normal(0, 1, 1)))
             #Verificamos que el nuevo valor no sea menor a épsilon
             if mutation_sigma[0] < epsilon:
                 mutation_sigma[0] = epsilon

             #Mutamos las variables de decisión
             mutation_x = i[:n] + (mutation_sigma[0]*np.random.normal(0, 1, n))

             #Revisamos que estén dentro de los límites
             mutation_x[mutation_x < -30] = -30
             mutation_x[mutation_x >  30] =  30

             return np.concatenate((mutation_x, mutation_sigma))

In [24]: tau = 1/sqrt(2) #Tasa de aprendizaje
         epsilon = 0.01 #mínimo valor de sigma
         i = [-10.3, 7.84, 0.84]
         new_i = mutation(i, n, epsilon, tau)
         print(new_i)

[-10.15890813   7.69605627   0.4825125 ]
```

Crear hijo usando cruza y mutación

```
In [25]: def createChild(parent_1, parent_2, n, epsilon, tau):
             new_ind = localDiscreteCrossover(parent_1, parent_2)
             new_ind = mutation(new_ind, n, epsilon, tau)

             #Regresamos el nuevo individuo, junto con su valor de aptitud
             return [new_ind, fitness(new_ind[:n])]

In [26]: createChild(population[0][0], population[1][0], n, epsilon, tau)

Out[26]: [array([  9.79058706, -19.6921706 ,   1.24097226]), -20.876622230156865]
```

4.1.2. Algoritmo completo de (μ, λ) - EE

Se usa una selección (μ, λ) donde μ=100 y λ=700. Se elegirán dos padres al azar para generar un hijo nuevo y este proceso se llevará a cabo 700 veces. De los hijos resultantes, se elegirán a los 100 mejores que pasarán a la siguiente generación.

```
In [27]: def EvolutionStrategies(n, G, epsilon=0.01, tau=None, mu_=100,
lambda_=700):
             #Si el usuario no define tau,
             #se usa el valor recomendado en la literatura
             if tau == None:
                 tau = 1/sqrt(n)

             population = getInitialPopulation(mu_, n)
             #Generamos los índices de los individuos para poder elegirlos
             #aleatoriamente
             population_idx = range(mu_)

             #Repetimos el proceso por G generaciones
             for _ in range(G):
                 offspring = []
                 #Generamos lambda_ hijos
                 for i in range(lambda_):
                     #Seleccionamos a los padres
                     parents_idx = choices(population_idx, k=2)
                     parent_1 = population[population_idx[0]][0]
                     parent_2 = population[population_idx[1]][0]
                     #Creamos el hijo y lo añadimos a la población de hijos
                     offspring.append( createChild(parent_1, parent_2,
                     n, epsilon, tau) )

                 #Ordenamos del mejor individuo al peor individuo
                 offspring.sort(key=lambda x: x[-1], reverse=True)
                 #Seleccionamos a los mu_ mejores
                 population = offspring[:mu_].copy()


             #Regresamos el mejor individuo
             return population[0][0], -population[0][1]
```

```
In [28]: n = 2 #Número de variables/dimensiones
         mu_ = 100 #tamaño de la población
         lambda_ = 700 #Cantidad de hijos que se generan por generación
         G = 100 #número de generaciones
         tau = 1/sqrt(n) #tasa de aprendizaje
         epsilon = 0.01 #valor mínimo de sigma (tamaño de paso)
```

```
In [29]: sol, fx = EvolutionStrategies(n, G, epsilon, tau, mu_, lambda_)
         print(f"x: {sol[:n]}\nSigma: {sol[n:]}\nf(x): {fx}")
```

```
x: [-0.00065657  0.00037216]
Sigma: [0.01]
f(x): 0.002149800797706991
```

Realizamos m ejecuciones de nuestro algoritmo usando 5, 10 y 20 variables de decisión. Para cada instancia del problema, obtenemos la corrida que encontró la mejor solución, la corrida que encontró la peor solución, la corrida que se encuentra en la mediana de las m ejecuciones, la media y la desviación estándar de las m corridas con respecto al valor de la función objetivo.

```
In [30]: def makeMRuns(m, n, G, epsilon, tau):
             sol =[]
             for m in range(m):
                 x, fx = EvolutionStrategies(n, G, epsilon, tau)
                 sol.append([x, fx])

             print(f"Para {n} variables")

             sol.sort(key=lambda individuo: individuo[-1])
             print("*************** Mejor solución ***************")
             print(f"x: {sol[0][0][:n]}\nSigma: {sol[0][0][n:]}\n
             f(x): {sol[0][-1]}")

             print("\n*************** Peor solución ****************")
             print(f"x: {sol[-1][0][:n]}\nSigma: {sol[-1][0][n:]}\n
             f(x): {sol[-1][-1]}")

             print("\n****************** Mediana *******************")
             print(f"x: {sol[m//2][0][:n]}\nSigma: {sol[m//2][0][n:]}\n
             f(x): {sol[m//2][-1]}")

             f_sol = [x[-1] for x in sol]
             print("\nMedia: ", np.mean(f_sol))
             print("\nDesviación estándar: ", np.std(f_sol))
```

```
In [31]: #Instancia con 5 variables
         n = 5
         G = 100
         epsilon = 0.01
         tau = 1/sqrt(n)
         m = 21

         makeMRuns(m, n, G, epsilon, tau)
```

```
Para 5 variables
*************** Mejor solución ***************
x: [ 0.00064673 -0.00386858  0.00048901  0.00019717 -0.00088428]
Sigma: [0.01]
f(x): 0.007429178001150394

*************** Peor solución ****************
x: [ 29.99756535 -24.9972199  -14.9996601  -13.9957628   26.00348506]
Sigma: [0.01047811]
f(x): 19.79532217626284

****************** Mediana *******************
x: [ 1.29892061e+01  2.99840510e+00 -9.99321299e+00  3.37022526e-04
 -8.98909928e+00]
Sigma: [0.01]
f(x): 16.324552661876236

Media:  9.776078373439958

Desviación estándar:  9.336075319808344
```

```
In [32]: #Instancia con 10 variables
         n = 10
         G = 100
         epsilon = 0.01
         tau = 1/sqrt(n)
         m = 21

         makeMRuns(m, n, G, epsilon, tau)
```

```
Para 10 variables
*************** Mejor solución ***************
x: [-0.00355678 -0.00469178  0.00141009  0.00297665  0.00214689 -0.0053407
 -0.00626911 -0.0019492   0.00068978 -0.00434212]
Sigma: [0.01262651]
f(x): 0.015775159267399363

*************** Peor solución ****************
x: [ 28.99420216  28.99560935   8.99479738 -25.99707964 -19.00635308
  21.00779196  27.99865091   5.00792837  -5.99960455  -1.00384787]
Sigma: [0.01]
f(x): 19.650596767560053

****************** Mediana *******************
```

```
x: [ 3.00131095e+00 -1.29979440e+01 -9.99735252e+00 -1.50097207e-03
  1.09964389e+01  3.00359373e+00  3.99191258e+00  8.99595040e+00
  1.99896535e+01  1.29886174e+01]
Sigma: [0.01]
f(x): 17.482527440612373

Media:  12.858952654109073

Desviación estándar:  8.197688889476673
```

```
In [33]: #Instancia con 20 variables
         n = 20
         G = 100
         epsilon = 0.015
         tau = 1/sqrt(n)
         m = 21

         makeMRuns(m, n, G, epsilon, tau)
```

```
Para 20 variables
*************** Mejor solución ***************
x: [-0.00096883 -0.01002416 -0.01067195 -0.00834144  0.00048528  0.01650114
  0.00474224  0.0038431   0.00660964 -0.00306943  0.01213031 -0.0023433
  0.00077898 -0.01847668 -0.00778009 -0.00873538  0.00346705  0.00676743
 -0.01373322  0.01753902]
Sigma: [0.015]
f(x): 0.043170523843248265

*************** Peor solución ****************
x: [ 21.00506099  12.97651264  12.99219435  15.00800424 -10.00504334
 -15.99002891  -0.99609665  21.01679554   4.00619211   3.99273967
  21.00221368 -22.99073379  15.00043309 -22.99548788  20.98875876
 -12.00360914  18.00666284 -13.98518125   8.01997577 -16.98441983]
Sigma: [0.01602363]
f(x): 19.16371926140832

****************** Mediana *******************
x: [-1.49840187e+01  5.99552206e+00 -6.01179188e+00 -2.49974553e+01
 -1.00433095e+00  7.72362131e-03  4.00673058e+00 -1.50045372e+01
 -8.98240037e+00  9.77063376e-01  5.00800423e+00 -1.09996444e+01
  9.01293109e+00 -7.99212644e+00 -6.99015853e+00 -4.98356274e+00
 -1.79853169e+01 -8.99187456e+00  2.29892261e+01  1.39955873e+01]
Sigma: [0.015]
f(x): 18.08099069250812
```

```
Media:  11.55775897319607

Desviación estándar:  8.883558523605728
```

Capítulo 5

Algoritmos genéticos (AG)

5.1. Función de Beale

La función de Beale es una función de dos variables y está definida como sigue:

$$\text{mín } f(x_1, x_2) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2 \quad (5.1)$$

donde $-4.5 \leq x_1, x_2 \leq 4.5$. El mínimo global está en $x^* = (3, 0.5)$ y $f(x^*) = 0$.

5.1.1. Principales componentes de AG

Librerías de **Python** que vamos a utilizar.

```
In [1]: from random import randint, random, shuffle
        from copy import copy, deepcopy
        from math import ceil, log, e, exp, sqrt, pi, cos
        import numpy as np
```

Datos de entrada.

```
In [2]: n = 2 #Número de variables de decisión
        precision = 3 #número de dígitos de precisión
        bounds = [-4.5, 4.5] #límites inferior y superior
```

Representación de la información genética

La fórmula vista para calcular el número de bits que se necesitan por variable es:

$$num\_bits = \lceil \log_2((l_{sup} - l_{inf}) * 10^p) \rceil$$

donde p es la precisión deseada y l_{inf} y l_{sup} son los límites inferior y superior de la variable.

```
In [3]: def getSizeVariable(bounds, p):
            return ceil(log((bounds[1] - bounds[0])*(10**p), 2))

In [4]: num_bits = getSizeVariable(bounds, precision)
        print(num_bits)

14
```

Dado que nuestro problema tiene dos variables y ambas tienen el mismo límite superior e inferior, el tamaño de la cadena binaria que representa el cromosoma de un individuo es de longitud 28. Por ejemplo:

```
In [5]: #Si n = 2
        i = '01110100001000110101011001'
        print(i, len(i))

01110100001000110101011001 28
```

Para poder calcular la aptitud de cada individuo es necesario obtener el valor real de cada una de sus variables.

```
In [6]: def getRealValues(chromosome, num_bits, num_variables, bounds):
            point = []
            for i in range(num_variables):
                b = chromosome[i*num_bits:(i+1)*num_bits]

                #binario a entero
                d = 0
                e = num_bits-1
                for j in b:
                    d += (int(j) * 2**e)
                    e -= 1

                #entero a decimal
                r = bounds[0] + (((bounds[1]-bounds[0])*d)/(2**num_bits-1))
                point.append(r)

            return np.array(point)

In [7]: i_real = getRealValues(i, 14, 2, bounds)
        i_real

Out[7]: array([-0.41723128,  3.00247207])
```

Aptitud

Función de Beale

```
In [8]: def f(x):
            x1, x2 = x[0], x[1]
            term1 = (1.5 - x1 + x1*x2)**2
            term2 = (2.25 - x1 + x1*(x2**2))**2
            term3 = (2.625 - x1 + x1*(x2**3))**2

            return term1 + term2 + term3
```

Dado que estamos resolviendo un problema de minimización, asignamos la aptitud de cada individuo i de la siguiente forma:

$$f_a(i) = \frac{1}{f(x_1^{(i)}, x_2^{(i)}) + 1}$$

```
In [9]: def getFitness(f_x):
            return 1/(f_x+1)
```

```
In [10]: fx = f(i_real)
         print (f"x = {i_real}\nf(x) = {fx}\nAptitud(x) = {getFitness(fx)}")
```

```
x = [-0.41723128  3.00247207]
f(x) = 69.71559241216701
Aptitud(x) = 0.014141152833331061
```

Representación de un individuo

Para almacenar la información de los individuos, vamos a crear una clase con los siguientes atributos:

1. **x**: Cadena binaria con el cromosoma del individuo.
2. **x_real**: Valores reales de cada variable.
3. **f**: Valor en la función objetivo.
4. **fitness**: Aptitud del individuo.

Y tendrá la función **__str__**.

```
In [11]: class Individual:
             def __init__(self, chromosome):
                 self.x = chromosome
                 self.x_real = None
                 self.f = None
                 self.fitness = None
```

```
            def __str__(self):
                #Imprime información del individuo.
                #Este método se manda a llamar a través print
                return "Cadena binaria: {}\nComponentes reales: {}\n
                f(x): {}\nAptitud: {}".format(self.x, self.x_real, self.f,
                self.fitness)
```

Población inicial

Para cada posición de la cadena binaria de un individuo, se utiliza una distribución uniforme para generar un número aleatorio en el intervalo $(0, 1)$. Si el valor resultante es menor a 0.5, toma el valor de cero. De lo contrario, toma el valor de 1.

```
In [12]: def getInitialPopulation(mu, num_bits, n):
             population = []
             for _ in range(mu):
                 chromosome = ""
                 for i in range(n*num_bits):
                     if random()<.5:
                         chromosome+="0"
                     else:
                         chromosome+="1"

                 population.append(Individual(chromosome))
             return population
```

```
In [13]: mu = 100
         population = getInitialPopulation(mu, num_bits, n)
```

Evaluamos cada uno de los individuos:

```
In [14]: for p in population:
             p.x_real = getRealValues(p.x, num_bits, n, bounds)
             p.f = f(p.x_real)
             p.fitness = getFitness(p.f)
```

```
In [15]: print(population[0])
```

```
Cadena binaria: 00110010101000011010100000101
Componentes reales: [-2.72010621 -0.77046329]
f(x): 94.56586667106828
Aptitud: 0.010463987141370645
```

Selección de padres

Universal estocástica

```
In [16]: def getPopulationMean(population, mu):
             mean = 0

             for p in range(mu):
                 mean += population[p].fitness

             return mean/mu
```

```
In [17]: #Regresa un arreglo que contiene los índices de los individuos
         #que serán padres
         def universalStochastic(population, mu):
             r = random()
             mean = getPopulationMean(population, mu)
             cont = 0
             parents = []
             for i in range(mu):
                 cont += population[i].fitness/mean
                 while cont > r and len(parents) < mu:
                     parents.append(i)
                     r += 1

             return parents
```

```
In [18]: parents = universalStochastic(population, mu)
         print(population[parents[0]])
         print(population[parents[10]])
```

```
Cadena binaria: 00110010101000011010100000101
Componentes reales: [-2.72010621 -0.77046329]
f(x): 94.56586667106828
Aptitud: 0.010463987141370645
Cadena binaria: 11100001010001100011111010010
Componentes reales: [3.41997803 0.53753891]
f(x): 0.1092864247827332
Aptitud: 0.9014804271095824
```

Recombinación

Cruza de dos puntos:

1. Generamos dos puntos de cruza aleatorios con distribución uniforme.

2. A partir de la posición 0 y hasta el primer punto de cruza, el primer hijo toma la información del primer padre y el segundo hijo toma la información del segundo padre.

3. A partir del primer punto de cruza y hasta el segundo punto de cruza, el primer hijo toma la información del segundo padre y el segundo hijo toma la información del primer padre.

4. A partir del segundo punto de cruza y hasta el final de la cadena, el primer hijo toma la información del primer padre y el segundo hijo toma la información del segundo padre.

```
In [19]: def twoPointsCrossover(p1, p2):
             index1 = randint(1, len(p2.x)-1)
             index2 = index1
             while index2 == index1:
                 index2 = randint(1,len(p2.x)-1)

             if index2 < index1:
                 index2, index1 = index1, index2

             offspring_1 = p1.x[:index1]+p2.x[index1:index2]+p1.x[index2:]
             offspring_2 = p2.x[:index1]+p1.x[index1:index2]+p2.x[index2:]

             return Individual(offspring_1), Individual(offspring_2)
```

```
In [20]: o1, o2 = twoPointsCrossover(population[parents[0]], population[parents[10]])
         o1.x_real = getRealValues(o1.x, num_bits, n, bounds)
         o2.x_real = getRealValues(o2.x, num_bits, n, bounds)
         o1.f = f(o1.x_real)
         o2.f = f(o2.x_real)
         o1.fitness = getFitness(o1.f)
         o2.fitness = getFitness(o2.f)
         print(o1)
         print(o2)
```

```
Cadena binaria: 00110010101000010010100000101
Componentes reales: [-2.72010621 -1.89553195]
f(x): 680.8091084998717
Aptitud: 0.0014666861846422343
Cadena binaria: 11100001010001101011110100 10
Componentes reales: [3.41997803 1.66260758]
f(x): 305.4954578474802
Aptitud: 0.0032626910918126064
```

Mutación

Por cada elemento en la cadena binaria, generamos un número aleatorio en el intervalo $(0,1)$, utilizando una distribución uniforme. Si el número generado es menor que la probabilidad de mutación, se mutará el elemento: si es 1 se hace 0 y si es 0 se hace 1.

```
In [21]: def mutation(individual, pm):
             for i in range(len(individual.x)):
                 if random() < pm:
                     if individual.x[i] == "1":
                         individual.x = individual.x[:i] + "0" + individual.x[i+1:]
                     else:
                         individual.x = individual.x[:i] + "1" + individual.x[i+1:]
```

```
In [22]: print(f"Antes de mutar\n{o1.x}")
         mutation(o1, 0.1)
         print(f"\nDespués de mutar\n{o1.x}")
```

```
Antes de mutar
00110010101000010010100000101

Después de mutar
00110010101000111010100000101
```

Selección de sobrevivientes

Selección basada en edad aplicando elitismo: Si el peor de los hijos es peor que la mejor solución de la población actual, se sustituye el peor de los hijos por la mejor solución actual. La siguiente generación estará conformada por los hijos generados.

```
In [23]: def getBestIndividual(population):
             best = 0
             for i in range(1, len(population)):
                 if population[i].fitness > population[best].fitness:
                     best = i
             return best

         def getWorstIndividual(population):
             worst = 0
             for i in range(1, len(population)):
                 if population[i].fitness < population[worst].fitness:
                     worst = i
             return worst
```

```
In [24]: def getSurvivors(parents, offspring):
             best_ind = getBestIndividual(parents)
             worst_ind = getWorstIndividual(offspring)

             if parents[best_ind].fitness > offspring[worst_ind].fitness:
                 offspring.pop(worst_ind)
                 offspring.append(copy(parents[best_ind]))

             return offspring
```

5.1.2. Algoritmo Genético

A continuación se muestra el algoritmo genético completo.

```
In [40]: def GeneticAlgorithm(G, mu, pc, pm, n, precision, bounds):
             num_bits = getSizeVariable(bounds, precision)

             population = getInitialPopulation(mu, num_bits, n)
             #Fitness
             for p in population:
                 p.x_real = getRealValues(p.x, num_bits, n, bounds)
                 p.f = f(p.x_real)
                 p.fitness = getFitness(p.f)

             for _ in range(G):
                 parents = universalStochastic(population, mu)
                 shuffle(parents)
                 offspring = []
                 for i in range(0,mu,2):
                     if random() < pc:
                         o1, o2 = twoPointsCrossover(population[parents[i]],
                         population[parents[i+1]])
                     else:
                         o1, o2 = deepcopy(population[parents[i]]),
                         deepcopy(population[parents[i+1]])

                     mutation(o1, pm)
                     o1.x_real = getRealValues(o1.x, num_bits, n, bounds)
                     o1.f = f(o1.x_real)
                     o1.fitness = getFitness(o1.f)

                     mutation(o2, pm)
                     o2.x_real = getRealValues(o2.x, num_bits, n, bounds)
                     o2.f = f(o2.x_real)
                     o2.fitness = getFitness(o2.f)
```

```
                    offspring.extend([o1, o2])

                population = getSurvivors(population, offspring)


            best = getBestIndividual(population)
            return population[best]
```

```
In [41]: G = 100 #Número de generaciones
         mu = 100 #tamaño de la población
         pc = 0.9 #probabilidad de cruza
         pm = 0.0001 #probabilidad de mutación
         n = 2 #número de variables de decision
         precision = 3 #número de dígitos de precisión
         bounds = [-4.5, 4.5] #límites inferior y superior
```

```
In [42]: print(GeneticAlgorithm(G, mu, pc, pm, n, precision, bounds))
```

```
Cadena binaria: 11011110000010100100000000110
Componentes reales: [3.30626259 0.56610511]
f(x): 0.010923175615842709
Aptitud: 0.9891948509250582
```

Realizamos m ejecuciones de nuestro algoritmo y obtenemos la corrida que encontró la mejor solución, la corrida que encontró la peor solución, la corrida que se encuentra en la mediana de las m ejecuciones, la media y la desviación estándar de las m corridas con respecto al valor de la función objetivo.

```
In [31]: def makeMRuns(m, G, mu, pc, pm, n, precision, bounds):
             sol =[]
             for m in range(m):
                 sol.append(GeneticAlgorithm(G, n, pc, pm, n, precision, bounds))


             sol.sort(key=lambda individuo: individuo.f)
             print("*************** Mejor solución ***************")
             print(sol[0])

             print("\n*************** Peor solución ****************")
             print(sol[-1])

             print("\n****************** Mediana *******************")
             print(sol[m//2])
```

```
            f_sol = [x.f for x in sol]
            print("\nMedia: ", np.mean(f_sol))
            print("Desviación estándar: ", np.std(f_sol))
```

```
In [34]: m = 21
         makeMRuns(m, G, mu, pc, pm, n, precision, bounds)
```

```
*************** Mejor solución ***************
Cadena binaria: 11011101100100100100000100001
Componentes reales: [3.28978209 0.58093756]
f(x): 0.020091304576727458
Aptitud: 0.9803044056090018

*************** Peor solución ****************
Cadena binaria: 11000010101010000000110001011
Componentes reales: [ 2.3438015  -4.28300678]
f(x): 35765.95829927722
Aptitud: 2.7958765507331612e-05

****************** Mediana *******************
Cadena binaria: 01010111111100100001011111110
Componentes reales: [-1.40825856  0.42107673]
f(x): 32.408936103276446
Aptitud: 0.029932111483847255

Media:  3412.875542684207
Desviación estándar:  8560.413403038703
```

Es importante mencionar que este es un diseño simple de un AG. Si se revisa el número de copias que se están realizando de los mejores individuos, se observará que es un valor muy alto y esto puede provocar convergencia a soluciones no óptimas.