Universidad Nacional Autónoma de México

Licenciatura en Tecnologías para la Información en Ciencias

Escuela Nacional de Estudios Superiores,
Unidad Morelia

Librería para optimización con metaheurísticas

TESIS

que para obtener el título de

Licenciado en tecnologías para la información en ciencias

presenta

Jesús Armando Ortíz Peñafiel

Directora del trabajo:

Dra. Adriana Menchaca Méndez

Morelia, Michoacán. 6 de octubre del 2023

MTRA. IVONNE RAMÍREZ WENCE
DIRECTORA
DIRECCIÓN GENERAL DE ADMINISTRACIÓN ESCOLAR
P R E S E N T E

Por medio de la presente me permito informar a usted que en la **sesión ordinaria 08** del **Comité Académico** de la **Licenciatura en Tecnologías para la Información en Ciencias** de la Escuela Nacional de Estudios Superiores (ENES) Unidad Morelia celebrada el día **27 de abril de 2022**, se acordó poner a su consideración el siguiente jurado para la presentación del Trabajo Profesional del alumno **Jesús Armando Ortíz Peñafiel** de la Licenciatura en **Tecnologías para la Información en Ciencias,** con número de cuenta **416067657**, con el trabajo titulado: **"Librería para optimización con metaheurísticas"**, bajo la dirección como tutora de la **Dra. Adriana Menchaca Méndez.**

El jurado queda integrado de la siguiente manera:

Presidente:	Dra. María del Río Francos
Vocal:	Dr. Luis Miguel García Velázquez
Secretario:	Dra. Adriana Menchaca Méndez
Suplente:	Dra. Marisol Flores Garrido
Suplente:	Dra. Miriam Pescador Rojas

Sin otro particular, quedo de usted.

A t e n t a m e n t e
"POR MI RAZA HABLARÁ EL ESPÍRITU"
Morelia, Michoacán a 14 de marzo de 2023.

DRA. YUNUEN TAPIA TORRES
SECRETARIA GENERAL

Agradecimientos Institucionales

Gracias al apoyo del Programa UNAM-DGAPA-PAPIME PE102320, la universidad, la licenciatura en tecnologías para la información en ciencias, el personal docente y un agradecimiento a los miembros del jurado que hicieron sus observaciones puntuales:

- Dra. María del Río Francos
- Dr. Luis Miguel García Velázquez
- Dra. Adriana Menchaca Méndez
- Dra. Marisol Flores Garrido
- Dra. Miriam Pescador Rojas

Agradecimientos personales

Este trabajo escrito es prueba de la conclusión de más de 4 años de lecciones personales y profesionales. Escribir esto me hace resucitar todos los personajes que estuvieron involucrados a lo largo de este capítulo. Dedico cada palabra a todos los actores que estuvieron presentes desde que tengo memoria. En especial a quien me ha dado cada aliento de su vida para brindarme un mundo sin igual, quien estuvo conmigo cada día desde que caímos en un encierro sin conocimiento del mañana, gracias mamá. A mis hermanas que han estado conmigo construyendo un sendero por el cual acompañarnos con un sin fin de posibilidades, a mi padre por todas las lecciones que me ha enseñado, Oscar quien ha sido un apoyo en las circunstancias complicadas y Sofia Paulina Maldona por extenderme la mano y abrazarme en todo momento con amor.

Agradezco a todos mis maestros que me han ayudado a quitar la venda que tenía. Me enseñaron la puerta a un mundo que no me imaginaba que estaba presente con un sin fin de desafiantes atractivos y muchas respuestas por donde mires. Quiero destacar a los grandiosos maestros Dra. Marisol, Dra. María del Rio, Dr. Luis Miguel y Dr. Raggi que me han enseñado por donde comenzar a recorrer todo este mundo y me han brindado lecciones dentro y fuera del salón.

Por último, esta aventura concluye con la persona que hizo que la docencia tenga un significado especial, quien me apoyo dentro y fuera de la aula. Muchas gracias por todas las palabras de aliento y las lecciones que siempre estarán presentes Dra. Adriana Menchaca Mendez.

Resumen

Many engineering, science, or industry applications involve solving complex optimization problems where classical methods do not work. For example, NP-hard problems or problems where the objective functions are not differentiable, discontinuous, or cannot be expressed algebraically. One way to deal with these problems is by using metaheuristics. A metaheuristic aims to find good solutions in a reasonable execution time. Some examples of metaheuristics are tabu search, simulated annealing, evolutionary programming, evolutionary strategies, genetic algorithms, differential evolution, and particle clustering.
Currently, there are several implementations of different metaheuristics. However, they could be more efficient, well-documented, and flexible to make them easier to use. In this thesis, we developed a library called Pyristic which has implemented the following. Five metaheuristics: (i) tabu search, (ii) simulated annealing, (iii) evolutionary programming, (iv) evolutionary strategies, and (v) genetic algorithms). Two continuous optimization test problems: (i) Beale function and (ii) Ackley function. And two combinatorial optimization problems: (i) binary knapsack problem and (ii) traveling agent problem implemented. Aiming to show how the Pyristic library works, we describe how to use each metaheuristic in the four test problems implemented. Finally, we compare the five metaheuristics and identify some advantages and disadvantages of applying them to these test problems.

Resumen

Muchas de las aplicaciones en ingeniería, ciencia o industria implican resolver problemas de optimización difíciles donde los métodos clásicos no funcionan. Por ejemplo, problemas NP-difíciles o problemas en los que las funciones objetivo no son diferenciables, son discontinuas o no se pueden expresar de manera algebraica. Una forma de hacer frente a estos problemas, es mediante el uso de metaheurísticas. Una metaheurística tiene como objetivo encontrar soluciones buenas en un tiempo de ejecución razonable. Algunos ejemplos de metaheurísticas son: búsqueda tabú, recocido simulado, programación evolutiva, estrategias evolutivas, algoritmos genéticos, evolución diferencial y cúmulo de partículas.

En la actualidad existen varias implementaciones de diferentes metaheurísticas. Sin embargo, muchas veces no son eficientes, no están bien documentadas o no son flexibles, lo que impide que puedan ser utilizadas fácilmente. En este trabajo de tesis, desarrollamos una librería llamada **Pyristic** que tiene implementadas cinco metaheurísticas (búsqueda tabú, recocido simulado, programación evolutiva, estrategias evolutivas y algoritmos genéticos), dos problemas de prueba de optimización continua (función de Beale y función de Ackley) y dos problemas de optimización combinatoria (problema de la mochila binario, problema del agente viajero). Teniendo como objetivo mostrar el funcionamiento de la librería **Pyristic**, se describe cómo hacer uso de cada una de las metaheurísticas en los cuatro problemas de prueba implementados. Finalmente, se hace una comparación de las cinco metaheurísticas y se identifican algunas ventajas y desventajas de aplicarlas en dichos problemas de prueba.

Índice general

Índice de algoritmos

Índice de figuras

Capítulo 1

Introducción

Desde hace varios años se ha identificado la necesidad de crear estrategias que apoyen en la toma de decisiones. Por ejemplo, elegir el camino para ir de un punto a otro dentro de una ciudad, la manera en que se almacenan los productos dentro de un vehículo repartidor, las posiciones en las que se colocarán un conjunto de bocinas dentro de un recinto, etc. Estas tomas de decisiones se pueden trasladar a un problema de optimización cuyo objetivo es encontrar la mejor opción. Por este motivo, desde hace mucho tiempo se han propuesto diferentes métodos de optimización y a la fecha esta área de investigación permanece activa.
En este trabajo de tesis estudiamos cinco metaheurísticas empleadas para resolver problemas de optimización, continuos y combinatorios, en los que las técnicas clásicas de optimización no resultan adecuadas. Esto puede ser debido a que el espacio de búsqueda es muy grande, la función objetivo no es diferenciable o no es continua, entre otras posibles razones.

1.1. Antecedentes

En la actualidad, existen diversos métodos de optimización para lidiar con diferentes tipos de problemas. Algunos ejemplos de métodos clásicos son el algoritmo simplex, el método de la caminata aleatoria y el método de Newton. El método simplex nos permite resolver problemas de optimización continua, sin embargo, la función y el conjunto de restricciones deben ser lineales. El método de la caminata aleatoria se emplea en problemas de optimización continua, sin embargo, es sensible al punto inicial y converge rápido a un mínimo local. Existen los métodos que dependen del gradiente, uno de ellos es el método de Newton, sin embargo, depende de que la función sea derivable en todo el dominio y se requiere hacer el cálculo de la matriz Hessiana y de su inversa a cada iteración, lo cual genera un costo computacional elevado.

Como podemos observar, los métodos clásicos tienen varias desventajas y muchas veces no pueden ser aplicados. Por esta razón, surgen las llamadas metaheurísticas.

Estas técnicas buscan resolver de una manera más general diferentes tipos de problemas. Su objetivo es encontrar buenas soluciones en un tiempo de ejecución razonable, sin garantizar encontrar la solución óptima. Algunos ejemplos de metaheurísticas son: búsqueda tabú [5, 6], recocido simulado [14, 11], programación evolutiva [2], estrategias evolutivas [17], algoritmos genéticos [9], evolución diferencial [20] y cúmulo de partículas [10]. Debido a la amplia aplicación de metaheurísticas en diversos problemas del mundo real, existen varias implementaciones de las mismas. Por ejemplo, en el repositorio "metaheuristic github[1]"podemos encontrar las siguientes:

- **Moe**[2]. Librería desarrollada en el lenguaje de programación C++. No tiene dependencias externas (Boost, Armandillo, Libtorch, etc.). Los algoritmos que tiene implementados son:
  - Evolución diferencial
  - Cúmulo de partículas
  - Recocido simulado.
  - Algoritmos genéticos
- **Heurisko**[3]. Librería implementada en el lenguaje de programación C++. Los algoritmos implementados están paralelizados utilizando *OpenMP* y son los siguientes:
  - Evolución diferencial
  - Algoritmos genéticos
  - Cúmulo de partículas.
  - Optimizador "Grey Wolf"
  - Algoritmo de optimización ballenas
- **Optimal**[4]. Librería en fase Beta desarrollada en el lenguaje de programación Python. Los algoritmos implementados son:
  - Algoritmos genéticos
  - Algoritmo de búsqueda gravitacional
  - Entropía cruzada
  - Aprendizaje incremental basado en poblaciones

Algunos inconvenientes que hemos identificado en las implementaciones disponibles son: ausencia de documentación, dificultad para adaptar los algoritmos al problema que se desea resolver o muchas veces no son eficientes en tiempo de ejecución.

[1] https://github.com/topics/metaheuristic
[2] https://github.com/tonykero/Moe
[3] https://github.com/Willtl/heurisko
[4] https://github.com/JustinLovinger/optimal

1.2. Objetivos

1.2.1. Objetivo general

Desarrollar una *framework* que permita utilizar y ajustar diferentes metaheurísticas para la búsqueda de buenas soluciones en problemas de optimización difíciles.

1.2.2. Objetivos particulares

1. Implementar al menos cinco metaheurísticas.
2. Implementar al menos dos problemas de optimización continua y dos problemas de optimización combinatoria.
3. Unir las implementaciones de los puntos anteriores para crear una librería que permita resolver problemas de optimización utilizando metaheurísticas.
4. Resolver al menos uno de los problemas implementados con cada metaheurística.
5. Crear la documentación de la librería.

1.3. Contribuciones

Las contribuciones de esta tesis son:

1. Implementación de las siguientes metaheurísticas:
   a) Búsqueda Tabú
   b) Recocido Simulado
   c) Programación Evolutiva
   d) Estrategias Evolutivas
   e) Algoritmos Genéticos
2. Implementación de dos problemas de optimización continua:
   a) Función de Beale
   b) Función de Ackley
3. Implementación de dos problemas de optimización combinatoria:
   a) Problema del agente viajero
   b) Problema de la mochila binario

4. Construcción de la librería "Pyristic", la cual está ubicada en el repositorio público: https://github.com/JAOP1/pyristic.

5. Desarrollo de notebooks, utilizando la herramienta *Jupyter*, ejemplificando el uso de la librería en los cuatro problemas antes mencionados.

6. Creación del sitio de consulta de las clases y funciones de "Pyristic" en la siguiente liga: https://jaop1.github.io/pyristic/

7. Tabla comparativa de las cinco metaheurísticas aplicadas en los cuatro problemas de optimización.

Todas las implementaciones se hicieron utilizando el lenguaje de programación **Python**.

1.4. Organización de la tesis

Capítulo 2. Optimización: En este capítulo se definen los conceptos básicos de optimización y el conjunto de problemas que se utilizarán en los capítulos siguientes para ejemplificar cómo se emplea cada metaheurística.

Capítulo 3. Búsqueda Tabú: Este capítulo describe cada componente de la metaheurística de *búsqueda tabú* y se muestra cómo aplicarla a problemas de optimización combinatorios.

Capítulo 4. Recocido Simulado: En este capítulo se describe la metaheurística de *recocido simulado* y se muestra cómo utilizarla para resolver problemas de optimización combinatorio.

Capítulo 5. Programación Evolutiva: Este capítulo describe el paradigma de computación evolutiva conocido como *programación evolutiva* y se muestra cómo aplicarlo en problemas de optimización continuos.

Capítulo 6. Estrategias evolutivas: Al igual que la *programación evolutiva*, las *estrategias evolutivas* son uno de los principales paradigmas de computación evolutiva. En este capítulo se describe cada una de sus componentes y se muestra cómo aplicarlo en problemas de optimización combinatoria.

Capítulo 7. Algoritmos genéticos: En este capítulo se describe el tercer paradigma de la computación evolutiva, conocido como *algoritmos genéticos*. Se explican cada una de sus componentes y se muestra cómo utilizarlo tanto en problemas de optimización continua como combinatorios.

Capítulo 8. Resultados: En este capítulo se estudia el comportamiento de las metaheurísticas estudiadas en tres problemas de optimización: función de Ackley, problema de la mochila binario y problema del agente viajero. Para ello

se realiza un estudio experimental y se muestran los resultados obtenidos. Finalmente, se discuten las ventajas y desventajas de cada metaheurística en los problemas estudiados.

Capítulo 9. Conclusiones y trabajo futuro: Este capítulo señala las contribuciones de este trabajo de tesis y menciona algunas líneas de trabajo futuro.

Apéndice: En este apartado se explica el funcionamiento de la librería desarrollada. Se describen las clases, métodos y atributos de cada metaheurística. Además se muestra cómo fueron aplicadas a los diferentes problemas de optimización estudiados. Finalmente, se incluye una breve explicación sobre cómo emplear *pyristicLab*.

Capítulo 2

Optimización

Tanto en las ingenierías como en las áreas afines la toma de decisiones es un factor importante en el diseño de elementos tecnológicos o administrativos. En el sentido más amplio, el objetivo es minimizar costos y esfuerzos o maximizar beneficios.
La optimización se puede caracterizar por abstraer problemas complejos en términos matemáticos o simulaciones, para posteriormente aplicar métodos que busquen las mejores soluciones a dichos problemas. Es decir, encontrar el máximo o mínimo de una o más funciones. En la literatura, existen varios métodos de optimización disponibles [16]. Sin embargo, es importante considerar que ninguno de ellos puede resolver cualquier tipo de problema. Una heurística puede ser definida como una técnica de optimización que encuentra soluciones buenas a un problema, sin garantizar que sean las soluciones óptimas, en un tiempo razonable. Es importante mencionar, que este tipo de técnicas son utilizadas cuando los problemas no pueden ser resueltos con alguna técnica clásica. Existen heurísticas que permiten resolver diferentes tipos de problemas de una manera más general. Este tipo de heurísticas son conocidas como metaheurísticas y son en las que nos enfocaremos en este trabajo de tesis.

2.1. Definición de un problema de optimización mono-objetivo

Cuando estamos trabajando con un problema de optimización, se define un espacio de posibles soluciones al problema llamado *espacio de búsqueda*. Cada solución se denota por un vector, $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]^T$, conocido como *vector de diseño*. Cada componente x_i del vector es conocida como *variable de decisión*. Las variables de decisión son las variables de las que depende la o las funciones objetivo. Resolver un problema de optimización implica encontrar la solución $\boldsymbol{x}^* = [x_1, x_2, \ldots, x_n]^T$ que optimiza (minimiza o maximiza) la o las funciones objetivo $f_i(\boldsymbol{x})$ y satisface m condiciones de desigualdad y p condiciones de igualdad.

Dado que maximizar $f_i(\boldsymbol{x})$ es equivalente a minimizar $-f_i(\boldsymbol{x})$, ver figura 2.1, en este trabajo de tesis vamos a asumir siempre un problema de minimización. Cuando se trabaja con una única función objetivo se conoce como problema de optimización *mono-objetivo* y formalmente se define como sigue:

$$\begin{aligned} \text{minimizar:} \quad & f(\boldsymbol{x}) \\ \text{Sujeto a:} \quad & \\ & g_i(\boldsymbol{x}) \leq 0, \quad \text{donde } i = 1, \ldots, m \\ & h_j(\boldsymbol{x}) = 0, \quad \text{donde } j = 1, \ldots, p \end{aligned} \tag{2.1}$$

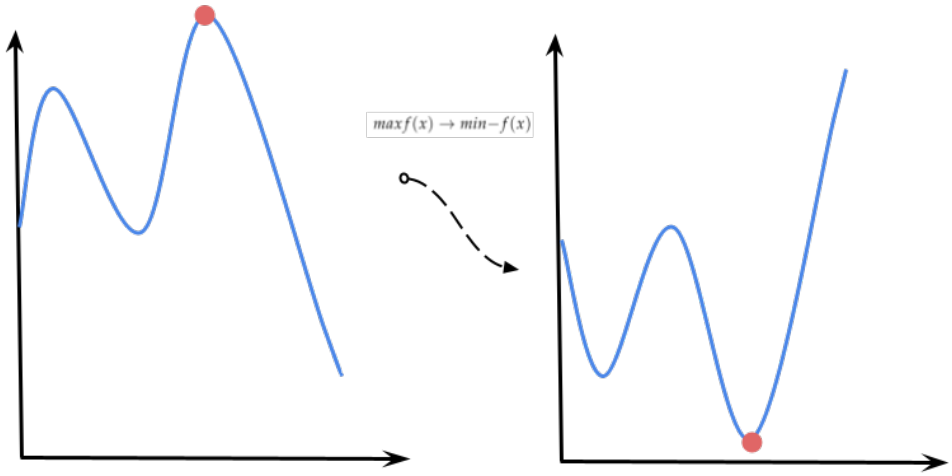


Figura 2.1: Transformación de un problema de maximizar a uno de minimizar.

El conjunto de soluciones que cumplen con todas las restricciones del problema es conocido como *región factible* y regularmente se denota como Ω. La solución óptima encontrada, $\boldsymbol{x}^*$, es conocida como **mínimo** y puede ser:

- Un *mínimo global*, si y solo si, $f(\boldsymbol{x}^*) \leq f(\boldsymbol{y})$ para todo $\boldsymbol{y} \in \Omega$.
- Un *mínimo local*, si y solo si, $f(\boldsymbol{x}^*) \leq f(\boldsymbol{y})$ para todo $\boldsymbol{y} \in N(\boldsymbol{x}^*) \cap \Omega$, donde, $N(\boldsymbol{x}^*)$ es el vecindario de $\boldsymbol{x}^*$.

Si una función tiene varios óptimos, ya sean locales o globales, se dice que la función es *multimodal*.
Existen problemas que no tienen restricciones, $m = p = 0$, y problemas en los que se desea optimizar más de una función objetivo a la vez. Estos últimos son conocidos como *problemas de optimización multi-objetivo*. Es importante señalar que en este trabajo solo se estudiarán problemas de optimización mono-objetivo con y sin restricciones.
Los problemas de optimización también se pueden clasificar dependiendo del tipo de valores que toman las variables de decisión. En esta tesis vamos a considerar dos: 1) problemas de optimización continua y 2) problemas de optimización combinatoria.

Si $\boldsymbol{x} \in \mathbb{R}^n$, se trata de un problema de optimización continua. En los problemas de optimización combinatoria, las variables de decisión toman valores discretos.

Región factible

El conjunto de soluciones $\boldsymbol{x}$ que satisfacen las restricciones $g_i(\boldsymbol{x})$ y $h_e(\boldsymbol{x})$ son conocidas como *soluciones factibles*. En el caso de no cumplir al menos una de las restricciones, será considerada como una *solución infactible*.

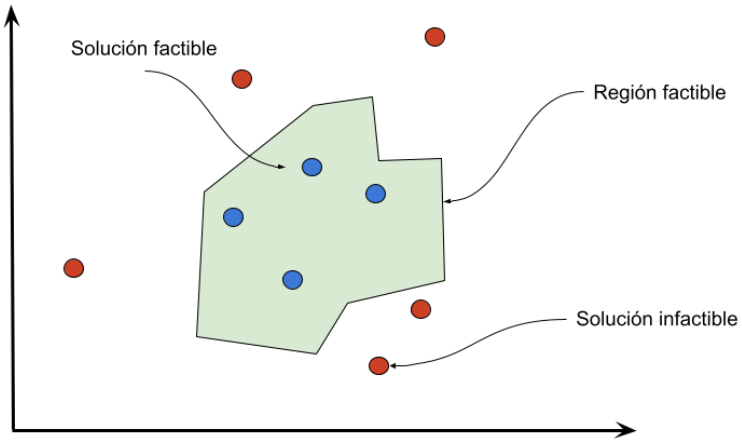


Figura 2.2: Región factible

2.2. Problemas de optimización combinatoria

En esta tesis vamos a abordar dos problemas de optimización combinatoria muy conocidos: problema de la mochila binario y problema del agente viajero. A continuación se describe cada uno de ellos.

2.2.1. El problema de la mochila binario (0/1 Knapsack problem)

Supongamos que tenemos que realizar una práctica de campo y tenemos una mochila en la que podemos guardar objetos. Dado que los trayectos de caminata serán largos, no resulta conveniente llevar más de un peso determinado en la mochila y por lo tanto debemos decidir qué objetos llevar. El objetivo es llevar los artículos más apropiados a la práctica de campo. Ver figura 2.3. El problema de la mochila binario, también conocido como *0/1 Knapsack problem* [1], consiste en seleccionar dentro de un conjunto A de n objetos, un subconjunto B ($B \subseteq A$) de objetos, que se introducirán en una mochila con capacidad c. Cada objeto tiene asignado un valor p_i y un peso w_i. Formalmente, lo definimos como:

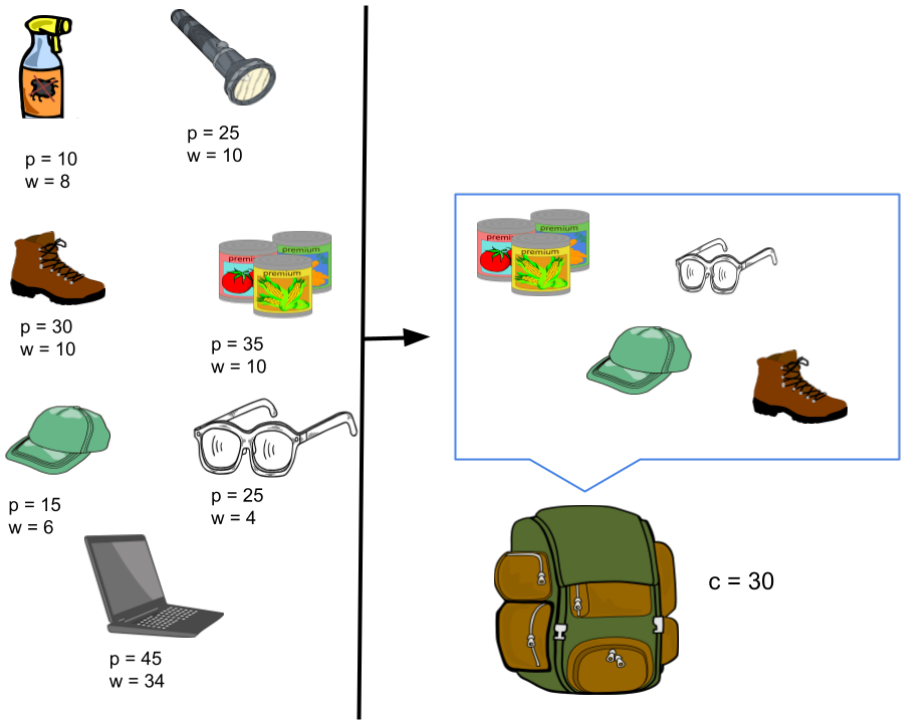


Figura 2.3: Instancia del problema de la mochila binario. En este caso se tienen 7 objetos con sus respectivos valores y pesos, la mochila tienen una capacidad de 30. Los objetos que maximizan el valor de la mochila, sin sobrepasar su capacidad, son la comida, los lentes, los zapatos y la gorra.

$$\begin{aligned} \text{maximizar:} \quad & f(\boldsymbol{x}) = \textstyle\sum_{i=1}^{n} p_i \cdot x_i \\ \text{tal que} \quad & g_1(\boldsymbol{x}) = \textstyle\sum_{i=1}^{n} w_i \cdot x_i \leq c \\ & x_i \in \{0,1\} \qquad\qquad i \in \{1,\ldots,n\} \end{aligned} \tag{2.2}$$

Como podemos ver, el problema de la mochila es un problema de encontrar máximos. Sin embargo, lo podemos transformar en un problema de minimización, de la siguiente forma:

$$\begin{aligned} \text{minimizar:} \quad & f(\boldsymbol{x}) = -1 \cdot \textstyle\sum_{i=1}^{n} p_i \cdot x_i \\ \text{tal que} \quad & g_1(\boldsymbol{x}) = \textstyle\sum_{i=1}^{n} w_i \cdot x_i \leq c \\ & x_i \in \{0,1\} \qquad\qquad i \in \{1,\ldots,n\} \end{aligned} \tag{2.3}$$

En este caso, una solución es un vector $\boldsymbol{x} = [x_1, \ldots, x_n]^T$, donde cada componente x_i puede tomar un valor de 1 ó un valor de 0. El 1 indica que el objeto i está presente en la mochila, mientras que el valor 0 representa que el objeto i no está dentro de la mochila.

2.2.2. El problema del agente viajero (TSP problem)

El problema del agente viajero conocido como TSP [13], por sus siglas en inglés (travellling salesman problem), consiste en visitar un conjunto de nodos (ciudades, negocios, localidades, etc.), regresando siempre a la ciudad de origen, con el menor costo posible. Cada nodo puede ser visitado una única vez. El problema considera que cada nodo está conectado con todos los nodos restantes. Si se tiene el mismo costo de ir del nodo i al nodo j que del nodo j al nodo i, para cualesquiera par de nodos i, j, se trata de la versión *simétrica* del problema. En caso contrario, se trata de la versión *asimétrica*. En este trabajo siempre nos vamos a referir a la versión simétrica. Ver figura 2.4. A continuación se describe formalmente:

$$\begin{array}{rll} \text{minimizar:} & f(\boldsymbol{x}) = \sum_{i=1}^{n-1} d(x_i, x_{i+1}) + d(x_n, x_1) & \\ \text{tal que} & \boldsymbol{x} = [x_1, \ldots, x_n], & x_i \neq x_j \\ & x_i \in \{1, 2, \cdots, n\} & i \in \{1, \ldots, n\} \end{array} \tag{2.4}$$

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

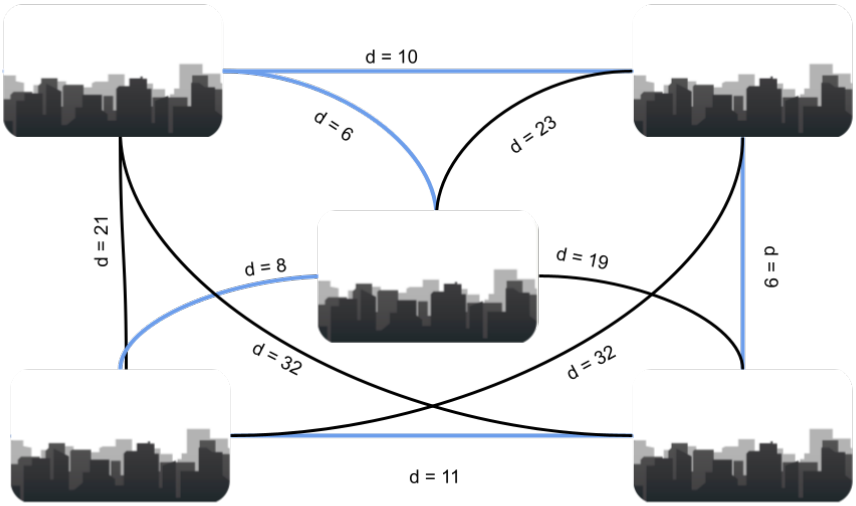


Figura 2.4: Instancia simétrica del problema del agente viajero. La línea azul denota el camino óptimo. Es decir, el camino que visita todos los nodos, regresando al origen, y tiene el menor costo.

La implementación en *Python* de este problema se puede consultar en el apéndice A y

forma parte de nuestra librería *pyristic*.

2.3. Problemas de optimización continuos

2.3.1. Beale

La función de Beale [15] es una función multimodal de dos variables y está definida como sigue:

$$\text{minimizar:} \quad f(x_1, x_2) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2 \tag{2.5}$$

donde $-4.5 \leq x_1, x_2 \leq 4.5$. Ver figura 2.5. El mínimo global está en $x^* = (3, 0.5)$ y $f(x^*) = 0$.

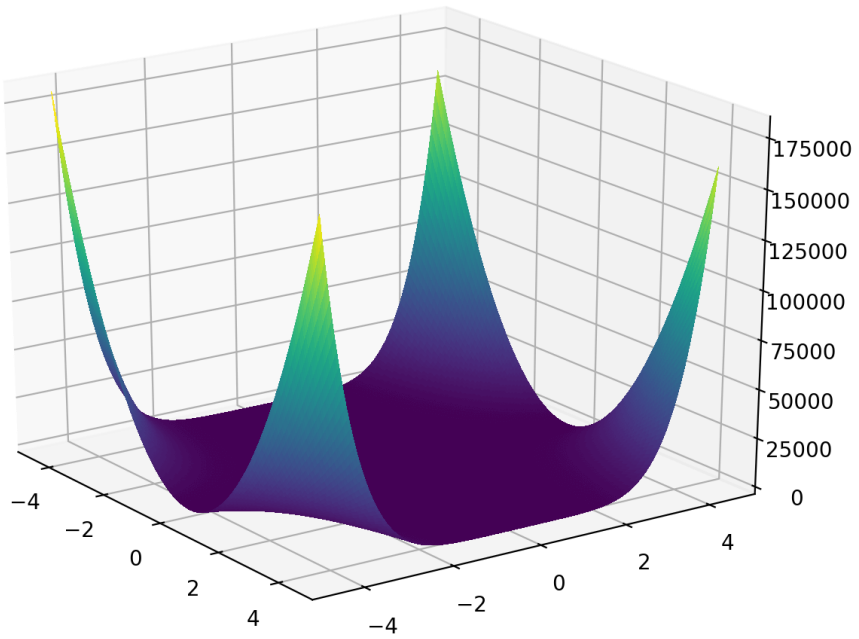


Figura 2.5: Función de Beale

La implementación en *Python* de este problema se puede consultar en el apéndice A y forma parte de nuestra librería *pyristic*.

2.3.2. Ackley

La función de Ackley [15] es una función multimodal de n variables y está definida como sigue:

$$\text{minimizar:} \quad f(\boldsymbol{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e \tag{2.6}$$

donde $30 \leq x_i \leq 30$. Ver figura 2.6. El mínimo global está en $x_i = 0$ y $f(\boldsymbol{x}) = 0$.

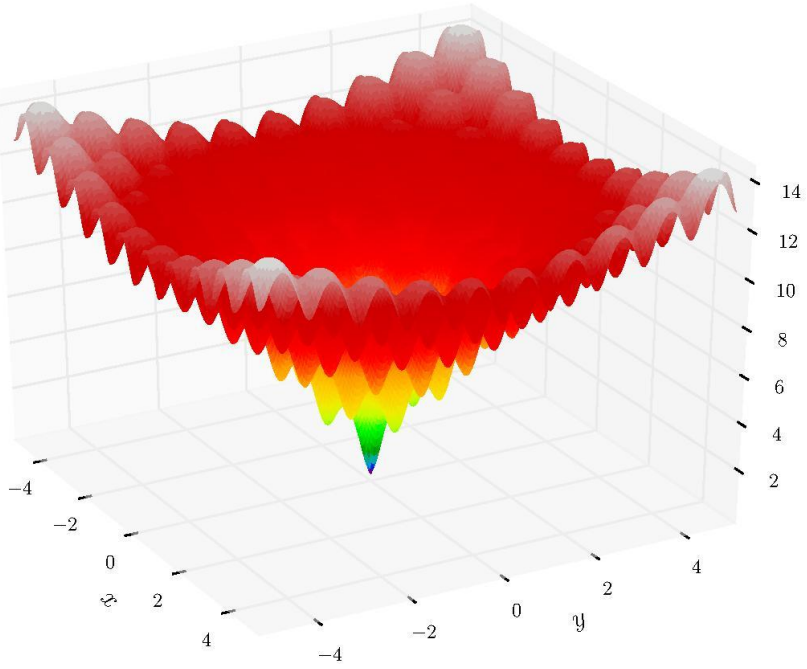


Figura 2.6: Función de Ackley

La implementación en *Python* de este problema se puede consultar en el apéndice A y forma parte de nuestra librería *pyristic*.

Nota:

- En los capítulos siguientes se presentaran los resultados obtenidos por los algoritmos, donde los valores obtenidos serán truncados a dos decimales. Por ejemplo, sea $x = 0.23545$ se mostrará únicamente $x = 0.23$

Capítulo 3

Búsqueda Tabú

> "Algorithms are conceived in analytic purity in the high citadels of academic research, heuristics are midwife by expediency in the dark corners of the practitioners lair . . . and are accorded lower status"
>
> *Fred Glover*, 1997

Búsqueda tabú (BT) es una metaheurística de búsqueda local que fue propuesta por Fred Glover [4, 5, 6]. BT comienza con una solución e iterativamente la va modificando, a través de la definición de un vecindario. La característica principal de BT es la creación de una estructura de datos llamada *lista tabú*, la cual almacena información durante la búsqueda. El objetivo de la *lista tabú* es evitar permanecer en óptimos locales y reducir costos computacionales al tener información sobre zonas exploradas.
BT fue aplicada originalmente en problemas de optimización combinatoria [7]. Sin embargo, han surgido propuestas para resolver problemas de optimización continua, ver por ejemplo [19]. En este capítulo se explicarán brevemente las principales componentes de BT y se aplicará en dos problemas de optimización combinatoria.

3.1. Descripción del algoritmo

Como mencionamos antes, BT define una estructura de datos llamada *lista tabú*. Esta lista almacena soluciones o características de dichas soluciones que serán evitadas por un lapso de tiempo. El tiempo de permanencia de un elemento en la lista tabú es conocido como *tiempo tabú*.
Sea S el espacio de búsqueda del problema que se desea resolver y x la solución actual. BT define *movimientos* que permiten crear un vecindario alrededor de x, $N(x) \subset S$, a cada iteración. Un movimiento genera nuevas soluciones a partir de pequeños cambios en la solución actual. El vecindario de x es reducido de acuerdo a la información

que se tiene almacenada en la lista tabú T, $N^*(x) = N(x) \setminus T$. Finalmente, BT elige una solución de $N^*(x)$ para sustituir a la solución actual x y así pasar a la siguiente iteración. Regularmente se selecciona a la mejor solución del vecindario. BT repite este procedimiento hasta cumplir una condición de paro. El algoritmo 1 muestra de manera muy general cómo opera BT. En las siguientes secciones presentamos una propuesta de algoritmos basados en BT para resolver dos problemas de optimización combinatoria: 1) problema de la mochila y 2) problema del agente viajero [1].

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 x_{best}.
Generar solución inicial x_0;
Actualizar mejor solución encontrada: $x_{best} \leftarrow x_0$, $f_{best} \leftarrow f(x_0)$;
Colocar el contador de iteraciones en 0: $k \leftarrow 0$;
Definir la solución actual: $x \leftarrow x_0$;
Iniciar la lista tabú como un conjunto vacío: $T = \emptyset$;
while $k < I_{max}$ **do**
    Determinar el vecindario $N(x)$;
    Determinar el vecindario reducido $N^*(x) = N(x) - T(x)$;
    Escoger la mejor solución y entre las soluciones que están en $N^*(x)$;
    $x \leftarrow y$;
    if $f(x)$ *es mejor que* $f(x_{best})$ **then**
        $x_{best} = x$;
        $f_{best} = f(x)$;
    $k \leftarrow k + 1$;
    Actualizar la lista tabú T considerando el tiempo tabú t_{tabu};
Regresar x_{best}, f_{best};

3.2. Aplicaciones

3.2.1. Problema de la mochila

Consideremos la siguiente instancia del problema de la mochila.

- Número de objetos: $n = 5$
- Valor de cada objeto: $p = [2, 4, 3, 5, 1]$
- Peso de cada objeto: $w = [1, 2, 3, 1, 2]$
- Capacidad de la mochila: $c = 5$

[1] La descripción de los problemas se puede consultar en el capítulo 2

A continuación se describe cada uno de los componentes que tendrá nuestra BT y se ejemplificará su funcionamiento utilizando esta instancia del problema.

Representación de la solución

La solución se representa como un arreglo de longitud n, donde, cada posición de nuestro arreglo es 1 si el objeto está presente en la mochila y 0 en caso opuesto. Por ejemplo:

- La solución $x = [1, 0, 0, 1, 1]$ indica que los objetos 0, 3 y 4 están contenidos en la mochila.
- Mientras que una solución $y = [0, 0, 0, 0, 0]$ representa una mochila vacía.

Solución inicial

La forma de generar nuestra solución inicial es inicializar x con $[0, 0, 0, 0, 0]$. Posteriormente, se realiza una permutación de un arreglo que contiene los índices de nuestra solución x. En este caso el arreglo de índices es $i = [0, 1, 2, 3, 4]$. Finalmente, se recorre el arreglo i ya permutado y se va actualizando x, colocando un 1 en el orden de dicha permutación. Antes de actualizar x se verifica que la suma de los w_i no exceda la capacidad c de la mochila. A continuación se muestra un ejemplo:

- Inicializamos nuestra solución, $x = [0, 0, 0, 0, 0]$, y su peso actual, $w = 0$.
- Se genera una permutación de los índices i de nuestra solución: $order = [3, 2, 4, 1, 0]$.
- Se van añadiendo los elementos verificando la suma de los w_i, donde, comenzamos con $w = 0$:

  1. Validamos si es posible ingresar el objeto de la posición 3, donde, $w_i = 1$ y $w' = w + w_i = 1$. Se cumple la condición $w' \leq 5$. Nuestra nueva solución es $x = [0, 0, 0, 1, 0]$ y nuestro peso actual es $w = 1$.
  2. Validamos si es posible ingresar el objeto de la posición 2, donde, $w_i = 3$ y $w' = w + w_i = 1 + 3 = 4$. Se cumple la condición $w' \leq 5$. Nuestra nueva solución es $x = [0, 0, 1, 1, 0]$ y nuestro peso actual es $w = 4$.
  3. Validamos si es posible ingresar el objeto de la posición 4, donde, $w_i = 2$ y $w' = w + w_i = 4 + 2 = 6$. No se cumple la condición $w' \leq 5$. Nuestra solución permanece igual $x = [0, 0, 1, 1, 0]$.
  4. Validamos si es posible ingresar el objeto de la posición 1, donde, $w_i = 2$. El objeto no se introduce a la mochila.
  5. El último objeto se introduce, porque, cumple la restricción dejando el peso en $w = 5$. Nuestra solución es $x = [1, 0, 1, 1, 0]$

Función objetivo y restricciones

El cálculo de la función objetivo se muestra en la ecuación (2.3) y recibe como entrada la solución x. Sea $x = [1, 0, 1, 1, 0]$, calculamos $f(x)$ como sigue:

- $f(x) = 1 \cdot 2 + 0 \cdot 4 + 1 \cdot 3 + 1 \cdot 5 + 0 \cdot 1 = 10$

La restricción del problema, g_1, recibe la misma solución x. Sea $x = [1, 0, 1, 1, 0]$, calculamos $g_1(x)$ como sigue:

- $g_1(x) = 1 \cdot 1 + 0 \cdot 2 + 1 \cdot 3 + 1 \cdot 1 + 0 \cdot 2 = 5$

En este caso, la solución x es una solución factible porque $g_1(x) \leq c$, donde, $c = 5$.

Vecindario

Para construir el vecindario $N(x)$, vamos a definir el siguiente movimiento. Dado el índice i, con $i \in \{0, 1, 2, 3, 4\}$, se cambia el valor de $x[i]$. Si $x[i] = 0$ hacemos $x[i] = 1$, de lo contrario, hacemos $x[i] = 0$. Sea $x = [1, 0, 1, 1, 0]$,

- Si $i = 0$, se realiza el siguiente movimiento: $[1, 0, 0, 1, 1] \rightarrow [0, 0, 0, 1, 1]$
- Si $i = 1$, se realiza el siguiente movimiento: $[1, 0, 0, 1, 1] \rightarrow [1, 1, 0, 1, 1]$

El vecindario $N(x)$ se construye a partir de n movimientos, donde n es el número de objetos. Sea $x = [1, 0, 1, 1, 0]$, tenemos que:

$$N(x) = \{[0, 0, 1, 1, 0], [1, 1, 1, 1, 0], [1, 0, 0, 1, 0], [1, 0, 1, 0, 0], [1, 0, 1, 1, 1]\}$$

Las soluciones $x' \in N(x)$ que no cumplen la restricción $g_1(x')$ se descartan.

Lista Tabú

Como se describió al inicio del capítulo, la lista tabú almacena soluciones o características de soluciones que queremos evitar por un lapso de tiempo T. En este caso vamos a almacenar características. Cada elemento de la lista tabú será una lista con tres elementos: $[p, v, t]$. Donde, p es una posición del arreglo x, v el valor que se puso en la posición p y t el número de iteraciones que seguirá en la lista tabú. Un elemento de la lista indica que no se puede actualizar la posición p con el valor v.
Supongamos que el tiempo de permanencia de los elementos de la lista tabú es $T = 5$. Sea $x = [1, 0, 0, 1, 1]$ la solución actual y $x' = [1, 1, 0, 1, 1]$ la solución vecina que se seleccionó para sustituir a x. Ingresaremos a la lista tabú el elemento $[1, 1, 5]$. Esto significa que en la posición con el índice 1 no podemos colocar un 1 durante 5 iteraciones.

Estudio experimental y resultados

En este trabajo de tesis, se construyó la clase *TabuSearch* que forma parte de la librería *Pyristic*. Esta clase permite abordar problemas de optimización basados en BT. En el apéndice A.1.2, se puede consultar la documentación de la clase y la implementación que se realizó para este problema.
El primer experimento consistió en resolver la instancia descrita en las secciones anteriores. Dado que se tienen 5 objetos, el espacio de búsqueda es de tamaño 2^5, es decir, todas las combinaciones de $0's$ y $1's$. Para esta instancia podemos emplear fuerza bruta para revisar cada una de las soluciones y encontrar el óptimo global. En este caso es $x = [1, 1, 0, 1, 0]$, $f(x) = -11$ y $w = 5$.
Para estudiar el comportamiento de nuestra metaheurística es necesario realizar un estudio estadístico. Para ello se ha ejecutado 30 veces con los siguientes parámetros:

- Número de iteraciones: 30
- Tiempo en la lista tabú: 3
- Solución inicial: Generada de manera aleatoria (descrita antes).

Los resultados obtenidos son:

- *Valor de la función objetivo obtenido por la mejor solución*: -11
- *Valor promedio*: -11
- *Desviación estándar*: 0.0
- *Valor de la función objetivo obtenido por la peor solución*: -11

Como podemos observar, para esta instancia pequeña nuestro algoritmo basado en BT logra obtener siempre la solución óptima. Para probar nuestra metaheurística en instancias más grandes, se generó una instancia aleatoria, utilizando una distribución uniforme, con las siguientes características:

- Número de objetos: 50
- $p_i \in [30, 100)$
- $w_i \in [20, 40)$
- La capacidad de la mochila $c = 870$

A diferencia de la instancia anterior, el tamaño del espacio de búsqueda es de $2^{50} = 1,125,899,906,842,624$ y no es posible realizar una búsqueda exhaustiva para conocer el óptimo global. Los parámetros empleados para la búsqueda con nuestra propuesta de BT son:

- Número de iteraciones: 500
- Tiempo en la lista tabú: 250
- Solución inicial: Generada de manera aleatoria (descrita antes).

Los resultados obtenidos son:

- *Valor de la función objetivo obtenido por la mejor solución*: -2309
- *Valor promedio*: -2244.52
- *Desviación estándar*: 38.09
- *Valor de la función objetivo obtenido por la peor solución*: -2180

Como podemos observar, esta instancia es más complicada para nuestro algoritmo y existe una variedad en las soluciones encontradas por cada ejecución. Sin embargo, la desviación estándar no es muy elevada considerando que los valores de los objetos están en el intervalo $[30, 100]$. Por lo anterior, podemos decir que la BT propuesta es robusta.

3.2.2. Problema del agente viajero

Consideremos la siguiente instancia del problema del agente viajero:

- Número de ciudades: 4
- Matriz de distancias entre las ciudades es:

$$M = \begin{bmatrix} 0 & 2 & 1 & 4 \\ 2 & 0 & 3 & 4 \\ 1 & 3 & 0 & 4 \\ 4 & 4 & 4 & 0 \end{bmatrix}$$

A continuación se describe cada uno de los componentes que tendrá nuestra BT y se ejemplificará su funcionamiento utilizando esta instancia del problema.

Representación de la solución

La solución se representa como un arreglo de longitud n, donde, n es el número de ciudades. Cada posición del arreglo indica la ciudad que será visitada. El orden de visita se considera de izquierda a derecha. Las ciudades están enumeradas desde el 0 y hasta $n - 1$. Por ejemplo:

- La solución $x = [3, 1, 2, 0]$ indica que el agente viajero comienza su recorrido en la ciudad con la etiqueta 3, después viaja a la ciudad 1 y sigue su recorrido hasta llegar a la ciudad con la etiqueta 0.

Solución inicial

Para este problema, vamos a utilizar la siguiente estrategia voraz (en inglés conocido como greedy) para generar la solución inicial.

1. Asumimos que la ciudad de origen es la ciudad 0 [2]. Por lo anterior, inicializamos x_0 con 0.
2. Revisamos entre todas las ciudades restantes por visitar cuál se puede visitar con el menor costo partiendo de la ciudad en la que nos encontramos.
3. Colocamos la ciudad del punto anterior en la siguiente posición del arreglo x y asignamos esta ciudad como la ciudad en la que nos encontramos.
4. Repetimos el proceso desde el punto 2 hasta que no haya más ciudades por visitar.

Ejemplo:

1. Inicializamos nuestra solución $x = [0]$ y el arreglo de ciudades no visitadas $c = [1, 2, 3]$.
2. Seleccionamos la ciudad con el menor costo desde la ciudad 0 que aún no se ha visitado. En este caso la ciudad 2: $d(0, 2) = 1$.
3. Insertamos la ciudad 2 en el arreglo x y retiramos esa ciudad del arreglo c. La nueva solución es $x = [0, 2]$ y $c = [1, 3]$
4. Seleccionamos la ciudad con el menor costo desde la ciudad 2 que aún no se ha visitado. En este caso la ciudad 1: $d(2, 1) = 3$.
5. Insertamos la ciudad 1 en el arreglo x y retiramos esa ciudad del arreglo c. La nueva solución es $x = [0, 2, 1]$ y $c = [3]$.
6. Seleccionamos la ciudad con el menor costo desde la ciudad 1 que aún no se ha visitado. En este caso la ciudad 3: $d(2, 1) = 4$.
7. Insertamos la ciudad 3 en el arreglo x y retiramos esa ciudad del arreglo c. La nueva solución es $x = [0, 2, 1, 3]$ y $c = []$

Función objetivo y restricciones

La función objetivo recibe como entrada una solución x y su definición se puede consultar en la ecuación (2.4).

- $f([0, 2, 1, 3]) = d(x_0, x_1) + d(x_1, x_2) + d(x_2, x_3) + d(x_3, x_0) = d(0, 2) + d(2, 1) + d(1, 3) + d(3, 0) = 1 + 3 + 4 + 4 = 12$

[2] Dado que el problema del viajero asume que el viajero regresa siempre a la ciudad de origen, no afecta qué ciudad sea considerada como la ciudad de origen.

Vecindario

El vecindario se va a generar a partir del siguiente movimiento. Elegir una posición aleatoria de x, mover este elemento a cualquiera de las otras $n-2$ posiciones, donde n es el número de ciudades. Recordemos que la ciudad de origen no puede moverse y que la posición actual no se puede considerar. Sea $x = [0,2,1,3]$,

- Si la posición aleatoria es 3 y la nueva posición es 1, se realiza el siguiente movimiento: $[0,2,1,3] \rightarrow [0,3,2,1]$
- Si la posición aleatoria es 1 y la nueva posición es 3, se realiza el siguiente movimiento: $[0,2,1,3] \rightarrow [0,1,3,2]$

Sea $x = [0,2,1,3]$ y la posición aleatoria 2, el vecindario de x queda como sigue:

$$N(x) = \{[0,1,2,3],[0,2,3,1]\}$$

Lista Tabú

En este caso vamos a almacenar características. Cada elemento de la lista tabú será una lista con dos elementos: [v,t]. Donde, v es la ciudad que se ha seleccionado para ser desplazada y t el número de iteraciones que aún permanecerá en la lista. Un elemento de la lista indica que no se puede desplazar el valor v durante t iteraciones. Supongamos que el tiempo de permanencia de los elementos de la lista tabú es $T = 2$. Sea $x = [0,2,1,3]$ la solución actual y $x' = [0,3,2,1]$ la solución vecina que se seleccionó para sustituir a x. Ingresaremos a la lista tabú el elemento $[3,2]$. Esto significa que la ciudad 3 no se puede desplazar durante 2 iteraciones.

Estudio experimental y resultados

En el apéndice A.1.3, se puede consultar la implementación utilizada para este problema. El primer experimento consiste en resolver la instancia pequeña, descrita al inicio de esta sección, donde, el espacio de búsqueda es $\frac{(4-1)!}{2}$, es decir, todas las permutaciones existentes al seleccionar para la primera posición la ciudad 0, para la segunda posición cualquiera de los 3 elementos restantes, para la tercera posición los 2 elementos restantes y en la última posición la ciudad restante. Además, dado que estamos resolviendo la versión simétrica es lo mismo ir de izquierda a derecha que de derecha a izquierda. En este caso es posible revisar las 12 posibles permutaciones. El óptimo global obtenido es en $x = [0,2,3,1]$ y $f(x) = 11$.
Para estudiar el comportamiento de nuestra metaheurística se ha ejecutado 30 veces con los siguientes parámetros:

- Número de iteraciones: 5
- Tiempo en la lista tabú: 2

- Solución inicial: Generada por la estrategia voraz (descrita antes).

Los resultados obtenidos son:

- *Valor de la función objetivo obtenido por la mejor solución*: 11
- *Valor promedio*: 11
- *Desviación estándar*: 0
- *Valor de la función objetivo obtenido por la peor solución*: 11

Finalmente, se creó la siguiente instancia aleatoria de mayor tamaño, utilizando una distribución uniforme:

- Número de ciudades: 10
- Matriz de distancias entre las ciudades:

$$M = \begin{bmatrix} 0 & 49 & 30 & 53 & 72 & 19 & 76 & 87 & 45 & 48 \\ 49 & 0 & 19 & 38 & 32 & 31 & 75 & 69 & 61 & 25 \\ 30 & 19 & 0 & 41 & 98 & 56 & 6 & 6 & 45 & 53 \\ 53 & 38 & 41 & 0 & 52 & 29 & 46 & 90 & 23 & 98 \\ 72 & 32 & 98 & 52 & 0 & 63 & 90 & 69 & 50 & 82 \\ 19 & 31 & 56 & 29 & 63 & 0 & 60 & 88 & 41 & 95 \\ 76 & 75 & 6 & 46 & 90 & 60 & 0 & 61 & 92 & 10 \\ 87 & 69 & 6 & 90 & 69 & 88 & 61 & 0 & 82 & 73 \\ 45 & 61 & 45 & 23 & 50 & 41 & 92 & 82 & 0 & 5 \\ 48 & 25 & 53 & 98 & 82 & 95 & 10 & 73 & 5 & 0 \end{bmatrix}$$

El tamaño del espacio de búsqueda es $\frac{(10-1)!}{2}$. Es decir, 181440 posibles caminos.
Los parámetros empleados en nuestra metaheurística son:

- Número de iteraciones: 100
- Tiempo en lista tabú: 50
- Solución inicial: Generada por la estrategia voraz (descrita antes).

Los resultados obtenidos son:

- *Valor de la función objetivo por la mejor solución:* 248
- *Valor promedio:* 248
- *Desviación estándar:* 0.0
- *Valor de la función objetivo obtenido por la peor solución:* 248

Como podemos observar, nuestra metaheurística siempre converge a la misma solución. Esto se debe a que la solución inicial se crea usando un algoritmo voraz y dado que BT es un buscador local, es posible que se quede atrapado en un óptimo local.

Capítulo 4

Recocido Simulado

Recocido simulado (RS) es una metaheurística de búsqueda local basada en el proceso físico de enfriamiento de los metales: Los metales son expuestos a temperaturas elevadas, y posteriormente, se enfrían lentamente con la finalidad de obtener una configuración de energía mínima. El algoritmo de *metrópolis* [14], es el pionero de estos métodos. RS ha sido aplicado en diferentes problemas, por ejemplo, Kirkpatrick et. al. [11] lo utilizaron para resolver el problema del agente viajero. En [12] muestran varias aplicaciones de RS. En este capítulo, se explicarán brevemente las principales componentes de RS y se aplicará en dos problemas de optimización combinatoria.

4.1. Descripción del algoritmo

El algoritmo de RS es un proceso iterativo que comienza con una temperatura alta y va disminuyendo a cada iteración. Sea S el espacio de búsqueda del problema que se desea resolver y x la solución actual, RS define un vecindario alrededor de x, $N(x) \subset S$, a cada iteración. Posteriormente, selecciona una solución vecina $y \in N(x)$. Finalmente, RS decide reemplazar x con y utilizando los siguientes criterios: i) si y es mejor que x con respecto a la función objetivo, $f(y) \leq f(x)$, o bien, ii) si $Random(0,1) < e^{\frac{-(f(y)-f(x(t)))}{T(t)}}$, donde, $T(t)$ es una función decreciente que indica la temperatura en el tiempo t. Regularmente $T(t)$ es una función lineal. Es decir, la probabilidad de seleccionar a y, siendo que y es peor que x, es alta al inicio de la búsqueda (temperaturas elevadas) y va decreciendo conforme avanza la búsqueda (temperaturas bajas). RS repite este procedimiento hasta cumplir una condición de paro, por ejemplo, llegar a una temperatura determinada. El algoritmo 2 muestra de manera general como opera RS. En las siguientes secciones presentamos una propuesta de algoritmos basados en RS para resolver dos problemas de optimización combinatoria:

1) problema de la mochila y 2) problema del agente viajero [1].

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 x_{best}.
Generar solución inicial x_0;
$T(0) \leftarrow T_0$;
$x(0) \leftarrow x_0$;
Actualizar la información de la mejor solución encontrada: $x_{best} \leftarrow x_0$, $f_{best} \leftarrow f(x_{best})$;
$t \leftarrow 0$;
while $T(t) > T_f$ **do**
  Elegir aleatoriamente una solución y del vecindario, $y \in N(x(t))$;
  if $f(y)$ *es mejor que* $f(x_{best})$ **then**
    $x_{best} = y$;
    $f_{best} = f(y)$;
  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
    $x(t+1) \leftarrow x(t)$;
  $T(t+1) \leftarrow$ Actualizar($T(t)$);
  $t \leftarrow t+1$;

4.2. Aplicaciones

4.2.1. Problema de la mochila

Consideremos la siguiente instancia del problema de la mochila:

- Número de objetos: $n = 5$
- Valor de cada objeto: $p = [2, 4, 3, 5, 1]$
- Peso de cada objeto: $w = [1, 2, 3, 1, 2]$
- Capacidad de la mochila: $c = 5$

A continuación se describe cada uno de los componentes que tendrá nuestro algoritmo basado en RS y se ejemplificará su funcionamiento utilizando esta instancia del problema.

[1] La descripción de los problemas se puede consultar en el capítulo 2.

Representación de la solución

Cada solución será un arreglo de longitud n, donde, cada posición de nuestro arreglo es 1 si el objeto está presente en la mochila y 0 en el caso opuesto. Por ejemplo:

- La solución $x = [1, 0, 0, 1, 1]$ indica que los objetos 0, 3 y 4 están contenidos en la mochila.
- Mientras que una solución $y = [0, 0, 0, 0, 0]$ representa una mochila vacía.

Solución inicial

Para generar nuestra solución inicial, definimos una medida de relación valor/peso de los objetos: $S(p_i, w_i) = \frac{p_i}{w_i}$. Posteriormente, los objetos son ordenados de manera decreciente con respecto al valor de S. Finalmente, seleccionamos uno a uno los objetos, verificando que la suma de los pesos de los objetos agregados a la mochila no exceda la capacidad de la misma.

- Inicializamos nuestra solución, $x = [0, 0, 0, 0, 0]$, y su peso actual, $w = g(x) = 0$.
- Calculamos S para cada uno de los objetos:
  1. Objeto 0: $S(p_0, w_0) = S(2, 1) = \frac{2}{1} = 2$.
  2. Objeto 1: $S(p_1, w_1) = S(4, 2) = \frac{4}{2} = 2$.
  3. Objeto 2: $S(p_2, w_2) = S(3, 3) = \frac{3}{3} = 1$.
  4. Objeto 3: $S(p_3, w_3) = S(5, 1) = \frac{5}{1} = 5$.
  5. Objeto 4: $S(p_4, w_4) = S(1, 2) = \frac{1}{2} = 0.5$.
- Creamos un arreglo con los valores de S de cada objeto: $s = [2, 2, 1, 5, 0.5]$.
- Ordenamos los objetos en orden decreciente con respecto a s. Para ello utilizamos un arreglo de tuplas, el resultado es $[(3, 5), (0, 2), (1, 2), (2, 1), (4, 0.5)]$. Donde, cada tupla (x, y) representa el valor del índice del objeto (x) y el valor en s (y).
- Construimos la solución x:
  1. Se considera agregar el objeto 3 a la mochila, su valor es de 5 y su peso de 1. Verificamos no exceder la capacidad de la mochila: $w = w + w_3 = 0 + 1 \leq 5$. Al no exceder la capacidad de la mochila agregamos el objeto 3: $x = [0, 0, 0, 1, 0]$ y $w = 1$.
  2. Se considera agregar el objeto 0 a la mochila, su valor es de 2 y su peso de 1. Verificamos no exceder la capacidad de la mochila: $w = w + w_0 = 1 + 1 \leq 5$. Al no exceder la capacidad de la mochila agregamos el objeto 0: $x = [1, 0, 0, 1, 0]$ y $w = 2$.

3. Se considera agregar el objeto 1 a la mochila, su valor es de 4 y su peso de 2. Verificamos no exceder la capacidad de la mochila: $w = w + w_1 = 2 + 2 \leq 5$. Al no exceder la capacidad de la mochila agregamos el objeto 1: $x = [1, 1, 0, 1, 0]$ y $w = 4$.
4. Se considera agregar el objeto 2 a la mochila, su valor es de 3 y su peso de 3. Verificamos no exceder la capacidad de la mochila: $w = w + w_3 = 4 + 3 \nleq 5$. Al exceder la capacidad de la mochila no agregamos el objeto 2.
5. Se considera agregar el objeto 4 a la mochila, su valor es de 1 y su peso de 2. Verificamos no exceder la capacidad de la mochila: $w = w + w_1 = 4 + 2 \nleq 5$. Al exceder la capacidad de la mochila no agregamos el objeto 4.

- Se regresa la solución generada $x = [1, 1, 0, 1, 0]$.

Función objetivo y restricciones

La función objetivo recibe como entrada la solución x y su definición se muestra en la ecuación (2.3).
Sea $x = [1, 1, 0, 1, 0]$, calculamos $f(x)$ como sigue:

- $f(x) = 1 \cdot 2 + 1 \cdot 4 + 0 \cdot 3 + 1 \cdot 5 + 0 \cdot 1 = 11$

La restricción del problema, g_1, recibe la misma solución x que se emplea para calcular la función objetivo de la ecuación (2.3)
Sea $x = [1, 1, 0, 1, 0]$, calculamos $g_1(x)$ como sigue:

- $g_1(x) = 1 \cdot 1 + 1 \cdot 2 + 0 \cdot 3 + 1 \cdot 1 + 0 \cdot 2 = 4$

En este caso, la solución x es una solución factible porque $g_1(x) < c$, donde, $c = 5$.

Vecindario

Sea x la solución actual, construimos la solución vecina $y \in N(x)$ de la siguiente manera. Se selecciona aleatoriamente una posición i del arreglo x. Si $x[i] = 0$ hacemos $x[i] = 1$, de lo contrario, hacemos $x[i] = 0$. Si la solución y resultante no es factible se realiza lo siguiente:

1. **Fase de reparación:** En esta fase se retiran objetos de la mochila, mientras el peso total exceda la capacidad de la mochila. Los objetos con un menor valor en la medida S son los que se van retirando.
2. **Fase de mejora:** En esta fase se introducen objetos a la mochila siempre y cuando no excedan la capacidad. Los objetos son priorizados de acuerdo a la medida S.

Ejemplo:

- Sea $x = [1, 0, 1, 1, 0]$ la solución actual.

- Se selecciona un índice $i \in \{0, 1, 2, 3, 4\}$ de manera aleatoria. Supongamos que $i = 1$.
- Dado que $x[1] = 0$, lo actualizamos con $x[i] = 1$. La solución vecina es $y = [1, 1, 1, 1, 0]$.
- Revisamos si la solución y es factible: $w = g(y) = 1 + 2 + 3 + 1 + 0 = 7$. Dado que $g(y) \nleq 5$, se aplican las fases de reparación y mejora antes descritas.

  Fase de reparación:

  - Consultamos el arreglo s y el objeto con menor valor en S que se encuentra en la mochila es el objeto 2. Lo retiramos y obtenemos $y = [1, 1, 0, 1, 0]$ y $g(y) = 4$.
  - Dado que $g(y) \leq 5$, dejamos de sacar objetos y pasamos a la fase de mejora.

  Fase de mejora:

  - La solución actual es $y = [1, 1, 0, 1, 0]$ con un peso $w = g(y) = 4$. Seleccionamos uno de los objetos que no se encuentra en la mochila ($i \in \{2, 4\}$) y que tiene el mayor valor en S. En este caso es el objeto con índice $i = 2$ y construimos la solución $y' = [1, 1, 1, 1, 0]$. Revisamos la condición $w' = w + w_2 = 4 + 3 \nleq 5$. Dado que se excede la capacidad de la mochila, no se añade el objeto con el índice $i = 2$ y seguimos con la solución $y = [1, 1, 0, 1, 0]$.
  - Revisamos el siguiente objeto que no está en la mochila y tiene el segundo valor más grande en S. En este caso es el objeto con índice $i = 4$ y construimos la solución $y = [1, 1, 0, 1, 1]$. Revisamos la condición $w' = w + w_4 = 4 + 2 \nleq 5$. Dado que se excede la capacidad de la mochila, no se añade el objeto con el índice $i = 4$ y seguimos con la solución $y = [1, 1, 0, 1, 0]$.
  - Dado que ya no hay más objetos por revisar, se termina la fase de reparación.

- La nueva solución es $y = [1, 1, 0, 1, 0]$.

Temperatura

En este trabajo se ha implementado una función lineal decreciente para definir el cambio de temperatura. Sea T_i la temperatura inicial y T_f la temperatura final, definimos $T(t)$ como sigue:

- $T(t + 1) = 0.9T(t)$, donde, t es una variable de tiempo, indica la iteración en la que se encuentra nuestro algoritmo.

- $T(0) = T_i$, representa la temperatura en el tiempo 0.
- Dada la temperatura T_i y la temperatura T_f podemos obtener el número de iteraciones t. Esto se logra al despejar t de la siguiente expresión $0.9^t T_i = T_f$.

Estudio experimental y resultados

En este trabajo se implentó la clase *SimulatedAnnealing* la cual forma parte la librería *pyristic*. En el apéndice A.2.3, se puede consultar la implementación utilizada en este problema.
El primer experimento consiste en resolver la instancia descrita en el apartado 4.2.1. Dado que se tienen 5 objetos, el espacio de búsqueda es de tamaño 2^5 (todas las combinaciones de $0's$ y $1's$). Para esta instancia se puede emplear fuerza bruta para obtener la solución óptima global: $x = [1, 1, 0, 1, 0]$, $f(x) = -11$ y $w = 5$. Recordemos que estamos usando la función objetivo de minimización.
Para estudiar el comportamiento de nuestra metaheurística es necesario realizar un estudio estadístico. Para ello se ha ejecutado 30 veces con los siguientes parámetros:

- Solución inicial: Estrategia voraz (descrita en el apartado 4.2.1)
- Temperatura inicial: 1000
- Temperatura final: 0.01

Los resultados obtenidos son:

- *Valor de la función objetivo obtenido por la mejor solución*: -11
- *Valor promedio*: -11
- *Desviación estándar*: 0.0
- *Valor de la función objetivo obtenido por la peor solución*: -11

Como podemos observar, la metaherística propuesta logra obtener el óptimo global en todas las ejecuciones. Sin embargo, es necesario probarla en instancias más grandes. Para ello, hemos generado una instancia aleatoria utilizando una distribución uniforme como sigue:

- Número de objetos: 1000
- $p_i \in [50, 100]$
- $w_i \in [5, 100]$
- La capacidad de la mochila $c = 9786$

A diferencia de la instancia anterior, el tamaño del espacio de búsqueda es de 2^{1000} y no es posible realizar una búsqueda exhaustiva para conocer el óptimo global.
Los parámetros empleados para la búsqueda con nuestra propuesta de RS son:

- Solución inicial: Estrategia voraz (descrita en el apartado 4.2.1)
- Temperatura inicial: 1000
- Temperatura final: 0.01

Los resultados obtenidos son:

- *Valor de la función objetivo por la mejor solución:* -31986.0
- *Valor promedio:* -31986.0
- *Desviación estándar*: 0.0
- *Valor de la función objetivo obtenido por la peor solución:* -31986.0

Como podemos observar, nuestra propuesta obtiene la misma solución en todas las ejecuciones. Esto nos indica que el algoritmo es robusto y no depende de la aleatoriedad. Sin embargo, no es posible asegurar que nuestro algoritmo obtuvo el óptimo global. Para entender mejor el comportamiento de nuestra metaheurística se podría hacer un estudio para ajustar los parámetros de entrada del algoritmo y ver si se puede llegar a una mejor solución. Por otro lado, se puede pensar en otras formas de generar la solución inicial, incluso hacer un estudio comparativo generando la solución inicial de forma totalmente aleatoria.

4.2.2. Problema del agente viajero

Consideremos la siguiente instancia del problema del agente viajero:

- Número de ciudades: 4
- Matriz de distancias entre las ciudades es:

$$M = \begin{bmatrix} 0 & 2 & 1 & 4 \\ 2 & 0 & 3 & 4 \\ 1 & 3 & 0 & 4 \\ 4 & 4 & 4 & 0 \end{bmatrix}$$

A continuación se describe cada uno de los componentes que tendrá nuestra versión de RS y se ejemplificará su funcionamiento utilizando la instancia antes descrita. En este ejemplo expresaremos la temperatura como una función lineal, descrita en la sección anterior.

Representación de la solución

La solución se representa como un arreglo de longitud n, donde, n es el número de ciudades. Cada posición del arreglo indica la ciudad que será visitada. El orden de visita se considera de izquierda a derecha. Las ciudades están enumeradas desde el 0 y hasta $n-1$. Por ejemplo:

- La solución $x = [3, 1, 2, 0]$ indica que el agente viajero comienza su recorrido en la ciudad con la etiqueta 3, después viaja a la ciudad 1 y sigue su recorrido hasta llegar a la ciudad con la etiqueta 0.

Solución inicial

Para este problema, vamos a utilizar la misma estrategia descrita en el apartado 3.2.2.

Vecindario

Para construir una solución vecina $y \in N(x)$, donde x es la solución actual, se seleccionan dos índices $i, j \in \{0, \ldots, n-1\}$ tales que $i \neq j$. El valor i corresponderá al índice en el arreglo x de la ciudad que vamos a mover. El valor j será la nueva posición en la que se encontrará la ciudad x_i.
Ejemplo:

1. La solución actual es $x = [0, 2, 3, 4, 1]$ y los índices aleatorios generados son: $i = 2$ y $j = 1$. Es importante recalcar que i debe ser diferente de j.

2. Ingresamos la ciudad $x_2 = 3$ en la posición y_1. El resto de las posiciones de y se llenan con -1, indicando que aún no se define la ciudad a visitar. Nuestra solución vecina actual es $y = [-1, 3, -1, -1, -1]$.

3. El resto de las ciudades se ingresan en el mismo orden que se encuentran en x, de izquierda a derecha:

4. $i = 0$, se ingresa la ciudad x_0 en la posición más a la izquierda que no contenga una ciudad. La solución vecina ahora es $y = [0, 3, -1, -1, -1]$.

5. $i = 1$, se ingresa la ciudad x_1 en la posición más a la izquierda que no contenga una ciudad. La solución vecina ahora es $y = [0, 3, 2, -1, -1]$.

6. $i = 2$, esta ciudad ya se encuentra en la solución y, por lo que continuamos con la siguiente posición.

7. $i = 3$, se ingresa la ciudad en la posición x_3 en la posición más a la izquierda que no contenga una ciudad. La solución vecina ahora es $y = [0, 3, 2, 4, -1]$.

8. $i = 4$, se ingresa la ciudad en la posición x_4 en la posición más a la izquierda que no contenga una ciudad. La solución vecina ahora es $y = [0, 3, 2, 4, 1]$.

9. Retornamos la solución vecina generada $y = [0, 3, 2, 4, 1]$.

Una de las diferencias más notables en comparación con BT es el proceso de generación del vecindario $N(x)$. En el caso de RS no es necesario generar todo el vecindario N.

Estudio experimental y resultados

En el apéndice A.2.2, se puede consultar la implementación utilizada para este problema. El tamaño del espacio de búsqueda para la instancia descrita al inicio de esta sección es de $\frac{3!}{2} = 3$. Recordemos que la ciudad de origen se queda fija, tenemos 3 posibles ciudades a visitar en el segundo lugar, 2 en el tercer lugar y 1 en el cuarto lugar. Dado que se trata de la versión simétrica del problema, es lo mismo ir de izquierda a derecha que de derecha a izquierda. El óptimo global está en $x = [0, 2, 3, 1]$ y $f(x) = 8$. Para estudiar el comportamiento de la metaheurística propuesta se ha ejecutado 30 veces con los siguientes parámetros:

- Solución inicial: Estrategia voraz (descrita en el apartado 4.2.2).
- Temperatura inicial: 1000
- Temperatura final: 0.01

Los resultados obtenidos son:

- *Valor de la función objetivo obtenido por la mejor solución*: 11
- *Valor promedio*: 11
- *Desviación estándar*: 0.0
- *Valor de la función objetivo obtenido por la peor solución*: 11

Para probar el desempeño de nuestra metaheurística en instancias más grandes, utilizamos la instancia aleatoria descrita en el apartado 3.2.2.
A diferencia de la instancia anterior, el tamaño del espacio de búsqueda es de $\frac{(10-1)!}{2} = 181440$ y no es posible realizar una búsqueda exhaustiva para conocer el óptimo global.

Los parámetros empleados en nuestra metaheurística son:

- Solución inicial: Estrategia voraz (descrita en el apartado 4.2.2).
- Temperatura inicial: 1000
- Temperatura final: 0.01

Los resultados obtenidos son:

- *Valor de la función objetivo por la mejor solución:* 248
- *Valor promedio:* 258.13
- *Desviación estándar:* 11.03
- *Valor de la función objetivo obtenido por la peor solución:* 271

Como podemos observar, la mejor solución obtenida por nuestra propuesta basada en RS es igual a la obtenida con la propuesta de BT, ver capítulo 3. Aunque, con RS se tiene una desviación estándar diferente de 0, al ser una desviación pequeña, en relación al costo del recorrido obtenido, podemos decir que la propuesta basada en RS es robusta. Por otro lado, es importante recordar que se utilizó la misma estrategia voraz para generar la solución inicial en ambas metaheurísticas. Esto podría provocar que ambas metaheurísticas estén convergiendo a un óptimo local. Al igual que con BT, se podría realizar un ajuste de parámetros para mejorar el comportamiento del RS propuesto y explorar otras alternativas para generar la solución inicial.

Capítulo 5

Programación Evolutiva

> "It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change."
>
> *Charles Darwin*

La *programación evolutiva* (PE) es uno de los principales paradigmas de la *computación evolutiva*[1]. Este grupo de algoritmos trabajan de la siguiente forma: Parten de una población inicial P de tamaño μ. Posteriormente, se elige un conjunto de individuos desde P (**padres**) a los cuales se les aplican **operadores genéticos** para generar un conjunto de λ nuevos individuos (**hijos**, generalmente se utiliza $\lambda \geq \mu$). Finalmente, se seleccionan los individuos que pasarán a la siguiente iteración (**selección de sobrevivientes**). Cada iteración es considerada una **generación** y el proceso se repite durante G generaciones o hasta cumplir una condición de paro. La PE fue propuesta por Lawrence J. Fogel [2] en la década de 1960 con el objetivo de simular la evolución como un proceso de aprendizaje. Cada uno de los individuos de la población es considerado de una especie distinta. Por esta razón, en PE no se considera un operador de cruza. Una característica importante de la PE es su comportamiento *autoadaptativo*. En este capítulo se explicarán brevemente los principales componentes de PE y se aplicarán en dos problemas de optimización continua.

5.1. Descripción del algoritmo

El algoritmo de PE es un proceso iterativo que consiste en la creación de descendientes, conocidos como **hijos**, a partir de la mutación de una población de individuos conocida como **padres**. Cada padre en la población genera un hijo por lo que ambas

[1]Conjunto de técnicas, inspiradas en la evolución biológica, para resolver problemas de optimización.

poblaciones son del mismo tamaño. Cada individuo es representado por dos arreglos. Uno representa la solución en el espacio de búsqueda y el otro corresponde al tamaño de desplazamiento para cada una de las variables del problema. Además, los individuos tienen asociado un valor de aptitud. La función de aptitud se define a partir del problema de optimización que se desea resolver.

Una vez que se tiene tanto a la población de padres como a la de hijos, ambas compiten por sobrevivir. Este proceso es conocido como *selección de sobrevivientes*. Una de las estrategias comúnmente utilizadas en PE es la *selección más*, la cual consiste en juntar las poblaciones de **padres** e **hijos** y seleccionar a los mejores individuos, de acuerdo a su valor de aptitud. El algoritmo 3 muestra de manera muy general cómo opera PE. Las variables x_i corresponden a las variables de decisión del problema, mientras que las variables σ_i corresponden a los tamaños de paso que se utilizan en las mutaciones. En las siguientes secciones presentamos una propuesta de algoritmos basados en PE para resolver dos problemas de optimización continua descritos en el capítulo 2: 1) la función de Ackley y 2) la función de Beale.

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 individuo encontrado $\boldsymbol{i}_{best}$.
padres $\leftarrow$ Población inicial de tamaño μ;
Evaluar cada individuo $\boldsymbol{i} \in$ **padres** con la función de aptitud F_a;
$t \leftarrow 1$;
while $t \leq G$ **do**
  hijos $\leftarrow$ Población vacía;
  foreach $i \in$ *padres* **do**
    foreach $j \in$ *variables de decisión* **do**
      $\sigma_j'^{(i)} \leftarrow \max(\varepsilon_0, \sigma_j^{(i)} \cdot (1 + \alpha \cdot N(0,1)))$;
      $x_j'^{(i)} = x_j^{(i)} + \sigma_j'^{(i)} \cdot N(0,1)$;
    hijo $\leftarrow$ Crear un nuevo individuo con los tamaños de desplazamiento $\sigma_j'^{(i)}$ y las variables de decisión $x_j'^{(i)}$;
    Evaluar al nuevo individuo con la función de aptitud F_a;
    Agregar el nuevo individuo a la población **hijos**;
  padres $\leftarrow$ Mejores μ individuos en **padres** $\cup$ **hijos**;
  $t \leftarrow t + 1$;
$\boldsymbol{i}_{best} \leftarrow$ Mejor individuo de la población actual;
Regresar $\boldsymbol{i}_{best}$;

5.2. Aplicaciones

5.2.1. Función de Beale

Función objetivo

La función objetivo recibe como entrada el arreglo de variables de decisión $\boldsymbol{x}$. La definición de la función objetivo se muestra en la ecuación (2.5). Sea $\boldsymbol{x} = [2.3, 3.4]$, calculamos $f(\boldsymbol{x})$ como sigue:

- $f(x) = (1.5 - 2.3 + 2.3 \cdot 3.4)^2 + (2.25 - 2.3 + 2.3 \cdot 3.4^2)^2 + (2.625 - 2.3 + 2.3 \cdot 3.4^3)^2 = 8984.42$.

Función de aptitud

La función de aptitud F_a mide qué tan bueno es un individuo con respecto al problema de optimización $f(x)$ que se desea resolver. Regularmente, la función de aptitud debe regresar un valor mayor o igual que cero y entre mayor sea el valor de aptitud se considera que se trata de un mejor individuo. Para la función de Beale, vamos a definir la aptitud como sigue:

$$F_a(x) = \frac{1}{f(x) + 1} \tag{5.1}$$

Sea $\boldsymbol{x} = [2.3, 3.4]$, calculamos $F_a(\boldsymbol{x})$ como sigue:

- $F_a(x) = \frac{1}{f([2.3,3.4])+1} = \frac{1}{8984.42+1} \approx 0$

Dada la definición de la función de Beale, sabemos que $f(x) \geq 0$. Por lo anterior, tenemos que $F_a(x) \in (0, 1]$:

1. Cuando $f(x)$ es un valor muy grande, F_a tiende a 0: $\lim_{f(x)\to\infty} F_a(x) = 0$.
2. Mientras que, cuando $f(x)$ es un valor muy pequeño, F_a tiende a 1: $\lim_{f(x)\to 0} F_a(x) = 1$.

Representación

Cada solución se representa con dos arreglos de números reales, $\boldsymbol{x}$ y $\boldsymbol{\sigma}$, ambos de tamaño $n = 2$. $\boldsymbol{x}$ contiene los valores que toman las variables de decisión y $\boldsymbol{\sigma}$ los tamaños de desplazamiento para cada variable de decisión. Por ejemplo:

$$\boldsymbol{x} = [2.3, 3.4]$$

$$\boldsymbol{\sigma} = [0.12, 0.37]$$

Población inicial

La población inicial se genera de la siguiente forma. El arreglo $\boldsymbol{x}$, de cada individuo, se llena con números aleatorios generados con una distribución uniforme en el intervalo $[-4.5, 4.5]$. El arreglo $\boldsymbol{\sigma}$ se llena con números aleatorios generados con una distribución uniforme en el intervalo $[0, 1]$. A continuación se muestra un ejemplo.
Población inicial de tamaño 4:

$$P_{inicial} = \begin{bmatrix} [-3.45, 2.34] & [0.13, 0.78] \\ [2.45, -2.34] & [0.45, 0.67] \\ [-1.29, 0.79] & [0.57, 0.98] \\ [3.49, -3.94] & [0.37, 0.88] \end{bmatrix} \tag{5.2}$$

Mutación y autoadaptación

Para generar la población de hijos, se realiza el proceso de mutación como sigue. Sean

$$\boldsymbol{x} = [x_1, x_2, \cdots, x_n]$$

$$\boldsymbol{\sigma} = [\sigma_1, \sigma_2, \cdots, \sigma_n]$$

los componentes de un individuo. Se genera un nuevo individuo:

$$\boldsymbol{x'} = [x'_1, x'_2, \cdots, x'_n]$$

$$\boldsymbol{\sigma'} = [\sigma'_1, \sigma'_2, \cdots, \sigma'_n]$$

utilizando:

$$\sigma'_j = \sigma_j \cdot (1 + \alpha \cdot N(0,1)) \tag{5.3}$$

$$x'_j = x_j + \sigma'_j \cdot N(0,1) \tag{5.4}$$

Nota:

- $N(0,1)$ retorna un número aleatorio empleando una distribución normal con media 0 y desviación estándar 1.
- α es un parámetro que permite controlar el tamaño del vecindario de la solución actual.

Sean $\boldsymbol{x} = [2.3, 3.4]$, $\boldsymbol{\sigma} = [0.12, 0.37]$ las componentes del individuo actual, $\alpha = 0.2$ y $[0.32, 0.49, 0.12, 0.40]$ la secuencia de números aleatorios generados con $N(0,1)$. Las componentes $\boldsymbol{x'}$ y $\boldsymbol{\sigma'}$ del nuevo individuo, se calculan de la siguiente forma:

- $\sigma'_0 = \sigma_0 \cdot (1 + \alpha \cdot N(0,1)) = 0.12 \cdot (1 + 0.2 \cdot 0.32) = 0.12$
- $\sigma'_1 = \sigma_1 \cdot (1 + \alpha \cdot N(0,1)) = 0.37 \cdot (1 + 0.2 \cdot 0.49) = 0.40$

- $x_1' = x_0 + \sigma_0' \cdot N(0,1) = 2.3 + 0.12 \cdot 0.24 = 2.32$
- $x_2' = x_1 + \sigma_1' \cdot N(0,1) = 3.4 + 0.40 \cdot 0.78 = 3.71$

El individuo resultante es $\boldsymbol{x}' = [2.32, 3.71]$ y $\boldsymbol{\sigma}' = [0.12, 0.40]$.

Soluciones inválidas

Cada vez que se genera un nuevo individuo es importante revisar que los valores tanto en su componente $\boldsymbol{x}$ como en su componente $\boldsymbol{\sigma}$ sean válidos. En el caso de la función de Beale los valores de las variables de decisión x_i deben cumplir: $\forall x_i \in [-4.5, 4.5]$. Para este ejercicio, si alguna de las variables no es válida, se ajusta a la cota superior o inferior. Por ejemplo:

1. $[3.4, -4.6] \rightarrow [3.4, -4.5]$.
2. $[5, 0.43] \rightarrow [4.5, 0.43]$.
3. $[-4.67, 4.87] \rightarrow [-4.5, 4.5]$

La primera solución $x = [3.4, -4.6]$ es inválida porque x_1 excede en 0.1 la cota inferior. La estrategia que se sigue es ajustar a la cota más próxima. De la misma manera ajustamos las soluciones de los ejemplos 2 y 3.

En el caso de los tamaños de desplazamiento σ_i. Se revisa si $|\sigma_i^{'}| < \epsilon_0$, en caso de que se cumpla, se actualiza con $\sigma_i^{'} = \epsilon_0$. Por ejemplo:

- Para $\boldsymbol{\sigma}_i^{'} = [0.13, 0]$ y $\epsilon_0 = 0.01$, ajustamos con $\boldsymbol{\sigma}_i^{'} = [0.13, 0.01]$.

Selección de sobrevivientes

El proceso de selección empleado es una *selección más*. Se une la población de **padres** y la población de **hijos**. Posteriormente, se ordenan los individuos con respecto al valor de aptitud y seleccionamos los μ mejores individuos. Donde μ es el tamaño de las poblaciones de **padres** e **hijos**.

- Sea P_X la población de padres y P_Y la población de hijos, con sus respectivos valores en la función de aptitud.

$$P_X = \begin{bmatrix} [2.3, 1.3] & [0.12, 0.57] & 0.02 \\ [4.3, 0.1] & [0.34, 0.89] & 0.07 \\ [-2.43, -1.23] & [0.33, 0.24] & 0.00 \end{bmatrix} \tag{5.5}$$

$$P_Y = \begin{bmatrix} [0.12, 0.34] & [0.15, 0.67] & 0.07 \\ [3, 0.5] & [0.88, 0.48] & 1 \\ [-2.34, 0.43] & [0.76, 0.29] & 0.02 \end{bmatrix} \tag{5.6}$$

- La nueva población está conformada por los individuos:

$$P_n = \begin{bmatrix} [4.3, 0.1] & [0.34, 0.89] & 0.07 \\ [0.12, 0.34] & [0.15, 0.67] & 0.07 \\ [3, 0.5] & [0.88, 0.48] & 1 \end{bmatrix} \tag{5.7}$$

Estudio experimental y resultados

En este trabajo de tesis se implementó la clase *EvolutionaryProgramming*, la cual forma parte de la librería *pyristic*. La implementación de PE se describe en el apéndice A.3.4. Nuestro estudio experimental consta de 30 ejecuciones con los siguientes parámetros:

- Número de generaciones (iteraciones): 700
- Tamaño de la población en cada generación: $\mu = 100$
- $\alpha = 0.5$

Los resultados obtenidos son:

- *Valor de la función objetivo por la mejor solución:* 0.0
- *Valor promedio:* 3.23e-28
- *Desviación estándar:* 1.26e-27
- *Valor de la función objetivo obtenido por la peor solución:* 5.94e-27

El óptimo global de este problema es en $\boldsymbol{x} = [3, 0.5]$ y $f(\boldsymbol{x}) = 0$. Dado que la media es en $f(x) = 0$ y la desviación estándar obtenida es muy pequeña, podemos concluir que la metaheurística propuesta resuelve la función de Beale exitosamente.

5.2.2. Función de Ackley

Como se describió en el capítulo 2, la función de Ackley no está definida para un número específico de variables de decisión. En este caso usaremos $n = 10$ variables de decisión y reutilizaremos los siguientes componentes descritos en la sección anterior.

- Representación de una solución (con $n = 10$ componentes)
- Población inicial
- Mutación y autoadaptación
- Modificación de soluciones inválidas
- Selección de sobrevivientes

Los únicos elementos que se deben modificar son la función objetivo y la función de aptitud.

Función objetivo

La función objetivo recibe como entrada el arreglo $\boldsymbol{x}$ del individuo. La definición de la función objetivo se muestra en la ecuación (2.6). Sea $x = [-7.05, -12.97, -23.93, 22.78, -11.70, 23.25, -25.49, -29.66, 10.13, 6.30]$, $f(x)$ queda como sigue:

- $f(x) = -20 \exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=0}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=0}^{n}\cos(2\pi x_i)\right) + 20 + e$
  $= 21.11$

Función de aptitud

En este caso, solo vamos a convertir el problema de minimización en uno de maximización: $F_a(\boldsymbol{x}) = -f(\boldsymbol{x})$, donde, $f(\boldsymbol{x})$ es la función de Ackley. Es importante observar que no estamos forzando a que $F_a(\boldsymbol{x}) \geq 0$.
Sea $x = [-7.05, -12.97, -23.93, 22.78, -11.70, 23.25, -25.49, -29.66, 10.13, 6.30]$

- $F_a(x) = -f(x) = -21.11$

Estudio experimental y resultados

La implementación utilizada para este problema se puede consultar en el apéndice A.3.5. Los parámetros utilizados son los siguientes:

- Número de generaciones (iteraciones): 500.
- Tamaño de la población en cada generación: $\mu = 100$.
- $\alpha = 2$.

Los resultados obtenidos al realizar 30 ejecuciones independientes son:

- *Valor de la función objetivo por la mejor solución:* 0.12
- *Valor promedio:* 3.33
- *Desviación estándar:* 3.003
- *Valor de la función objetivo obtenido por la peor solución:* 12.52

Sabemos que el óptimo global está en $\boldsymbol{x} = [0, \ldots, 0]$ y $f(\boldsymbol{x}) = 0$. Dado que la media está alejada del óptimo y la desviación estándar es grande, podemos concluir que la metaheurística propuesta no resuelve exitosamente la función de Ackley en altas dimensiones. Recordemos que esta función es altamente multimodal y es común quedarse atrapado en óptimos locales. Como trabajo a futuro se sugiere realizar un ajuste en los parámetros del algoritmo para tratar de mejorar su comportamiento.

Capítulo 6

Estrategias Evolutivas

Las *estrategias evolutivas* (EE) fueron propuestas por Rechenberg y Schwefel en la Universidad Técnica de Berlín en la década de 1960 y pertenecen a uno de los paradigmas de la *computación evolutiva*. El objetivo de Rechenberg y Schwefel era diseñar una boquilla en un problema de hidrodinámica [17]. Una distinción entre EE y PE es la interpretación de un individuo dentro de la población. En PE cada individuo se considera de una especie distinta, mientras que en EE todos los individuos pertenecen a una misma especie y por lo tanto se les puede aplicar un operador de recombinación (cruza).

La primera versión de EE es conocida como $(1+1)-EE$ y trabaja de la siguiente forma: Se genera un individuo $\boldsymbol{x}$ utilizando una distribución uniforme. Posteriormente, se genera un individuo $\boldsymbol{y}$ tras la mutación del individuo $\boldsymbol{x}$. Es decir, $\boldsymbol{x}$ actúa como padre y $\boldsymbol{y}$ como hijo. El individuo $\boldsymbol{x}$ es mutado de la siguiente forma: $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{z}$, donde, $\boldsymbol{z}$ es un vector con valores aleatorios generado con una distribución normal con media 0 y desviación estándar σ. El valor σ controla qué tanto se altera la solución $\boldsymbol{x}$. Finalmente, los individuos $\boldsymbol{x}$ y $\boldsymbol{y}$ compiten y sobrevive el individuo más apto. Es decir, el individuo con mejor aptitud será quien pase a la siguiente generación y actúe como el individuo padre. Este proceso se repite durante G generaciones. Ver algoritmo 4.

El valor σ no permanece constante a lo largo de las generaciones (iteraciones), se ajusta (**auto-adapta**) utilizando la **regla de éxito** $1/5$ cada k iteraciones. La regla de éxito se define como:

$$\sigma = \begin{cases} \sigma/c & \text{Si } p_s > 1/5, \\ \sigma \cdot c & \text{Si } p_s < 1/5, \\ \sigma & \text{Si } p_s = 1/5 \end{cases} \tag{6.1}$$

donde p_s es el porcentaje de mutaciones exitosas y c es una constante para controlar cuánto crece o decrece σ. En la literatura se aconseja usar $0.817 \leq c \leq 1$ [18]. En la ecuación (6.1) podemos ver que si el porcentaje de éxito es mayor a $1/5$, se decide realizar una *búsqueda más amplia* con la esperanza de no quedar atrapados en óptimos locales. Por el contrario, si el porcentaje de éxito es menor a $1/5$, se define un vecindario más pequeño con el objetivo de mejorar la solución localmente.

Algoritmo 4: Versión: $(1+1)$-EE

Entrada: Valor inicial de σ, constante c, número k de iteraciones para ajustar σ, número máximo de generaciones G y parámetros de entrada para el problema que se quiere resolver.
Salida : Mejor solución encontrada.

$t \leftarrow 0$;
$m_{successful} \leftarrow 0$;
Obtener solución inicial $\boldsymbol{x}$;
Evaluar $f(\boldsymbol{x})$;
while $t \leq G$ **do**
  Mutar la solución $\boldsymbol{x}$: $x_i' = x_i + \sigma \cdot N_i(0,1)$;
  if $\boldsymbol{x}'$ *es mejor que* $\boldsymbol{x}$ **then**
    $\boldsymbol{x} \leftarrow \boldsymbol{x}'$;
    $m_{successful} \leftarrow m_{successful} + 1$;
  $t \leftarrow t + 1$;
  if t **mod** $k == 0$ **then**
    $p_s \leftarrow m_{successful}/k$;
    If $p_s > 1/5$ **then** $\sigma \leftarrow \sigma / c$;
    If $p_s < 1/5$ **then** $\sigma \leftarrow \sigma \cdot c$;
    $m_{successful} \leftarrow 0$;
Regresar $\boldsymbol{x}$;

6.1. Descripción del algoritmo

Las EE al igual que PE son paradigmas de la computación evolutiva. Las diferencias principales con respecto a PE las veremos en los operadores genéticos y la selección de un conjunto de individuos que actuaran como padres. El algoritmo 5 muestra de manera muy general como opera una EE y en las siguientes secciones describimos cada uno de los componentes principales de las EE y se presenta una propuesta de algoritmos basados en EE para resolver los dos problemas de optimización continua abordados en el capítulo anterior: 1) función de Beale y 2) función de Ackley.

Algoritmo 5: Versión general de una EE

Entrada: Número máximo de generaciones G, 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.

$t \leftarrow 0$;
población $\leftarrow$ Obtener población inicial;
while $t \leq G$ **do**
    $cont \leftarrow 0$;
    hijos $\leftarrow \emptyset$;
    while $cont < \lambda$ **do**
        padres $\leftarrow$ Seleccionar aleatoriamente dos individuos de la población actual (**población**);
        hijo $\leftarrow$ Generar un hijo aplicando los operadores de cruza y mutación a los padres (**padres**);
        hijos $\leftarrow$ **hijos** $\cup$ {**hijo**};
        $cont \leftarrow cont + 1$;
    población $\leftarrow$ Seleccionar los individuos que pasarán a la siguiente generación de **población** $\cup$ **hijos**.;
    $t \leftarrow t + 1$;
$x_{best} \leftarrow$ Mejor individuo de la población actual;
Regresar x_{best}, f_{best};

6.2. Representación

En EE cada individuo de nuestra población se representa con dos arreglos $\boldsymbol{x}$ y σ. El arreglo $\boldsymbol{x}$ regularmente contiene valores flotantes asociados a cada variable de decisión del problema. Mientras, el arreglo σ (tamaño de desplazamiento), regula qué tanto son alteradas las variables de decisión al generar un hijo. El vector de desplazamiento σ puede ser de diferentes tamaños. En este trabajo se presentará únicamente $m = 1$ y $m = n$. Si σ es de longitud $m = 1$, nos referimos a un tamaño de desplazamiento para todas las variables de decisión del individuo asociado. Si $m = n$, se emplea un tamaño de desplazamiento distinto para cada variable de decisión. A continuación se muestran estas dos posibles representaciones.

- Cuando es un σ para todas las variables de decisión:

$$[x_1, \cdots, x_n], [\sigma]$$

- Cuando es un σ por variable de decisión:

$$[x_1, \cdots, x_n], [\sigma_1, \cdots, \sigma_n]$$

6.3. Selección de padres

La selección de padres es el proceso de selección de individuos que serán empleados en la fase de recombinación (cruza). En EE regularmente se seleccionan aleatoriamente utilizando una distribución uniforme.

Nota:
Los operadores de cruza que se presentarán a continuación, generan un único hijo H. Por esta razón si deseamos generar λ hijos,
se deben tomar 2λ individuos de la población actual.

6.4. Operadores de recombinación o cruza

Regularmente, los operadores de recombinación o cruza utilizan dos individuos (**padres**) para generar un nuevo individuo (**hijo**). Existen varias propuestas de operadores de cruza, a continuación se describen dos.

6.4.1. Discreta

Este tipo de recombinación genera un nuevo individuo tomando componentes de ambos padres. Para ello, recorre una a una cada componente y de manera aleatoria decide si pertenece al primer padre o al segundo.

6.4.2. Intermedia

El operador de cruza intermedia genera un nuevo individuo calculando el promedio de cada componente, considerando a ambos padres. Sean P_1 y P_2 los individuos **padres**, el nuevo individuo H se genera como sigue:

$$H = \frac{P_1 + P_2}{2}$$

6.5. Operadores de mutación

Los operadores de mutación alteran tanto las componentes correspondientes a las variables de decisión ($\boldsymbol{x}$) como las componentes asociadas a los tamaños de paso (σ). De esta forma, los tamaños de paso se van ajustando durante el proceso de búsqueda. Esto se conoce como **auto-adaptación**.
La mutación en las variables de decisión se realiza de la siguiente forma:

$$x_i' = x_i + \sigma_i' \cdot N(0,1) \tag{6.2}$$

donde $N(0,1)$ genera un número aleatorio utilizando una distribución normal con media 0 y desviación estándar 1. En el apartado 6.5 podemos observar que los valores de σ definen qué tanto podemos alterar la componente de nuestra solución actual. Una σ pequeña se traduce en cambios pequeños en nuestra solución, una σ grande significa cambios grandes en nuestra solución. Como mencionamos anteriormente, en el diseño de EE se ha considerado utilizar un tamaño de paso único asociado a todas las variables de decisión o un tamaño de paso por cada variable de decisión e incluso se han estudiado propuestas que consideran la correlación entre las variables. En las siguientes secciones describimos la mutación utilizada para las primeras dos opciones.

6.5.1. Mutación con un único tamaño de paso

En este caso, σ se actualiza de la siguiente forma:

$$\sigma' = \sigma \cdot e^{\tau \cdot N(0,1)} \tag{6.3}$$

donde $N(0,1)$ es un número aleatorio utilizando una distribución normal con media 0 y desviación estándar 1. τ es un parámetro de entrada conocido como tasa de aprendizaje. Se recomienda emplear $\tau = \frac{1}{n}$, donde, n es el número de variables de decisión.

6.5.2. Mutación con un tamaño de paso por variable de decisión

La motivación de usar un tamaño de paso por variable de decisión es que cada variable puede tener un comportamiento diferente. A continuación se muestra cómo se actualizan los tamaños de paso de cada variable:

$$\sigma_i' = \sigma_i \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)} \tag{6.4}$$

donde $N_i(0,1)$ representa un número aleatorio con una distribución normal con media 0 y desviación estándar 1 para cada σ_i. Los parámetros τ y τ' se conocen como tasas de aprendizaje. Se recomienda emplear $\tau = \frac{1}{\sqrt{2\sqrt{n}}}$ y $\tau' = \frac{1}{\sqrt{2n}}$, donde, n es el número de variables de decisión.

6.6. Esquemas de selección de sobrevivientes

A continuación se describen dos esquemas de selección de sobrevivientes que se utilizan comúnmente.

- $(\mu + \lambda)$. En este esquema se une la población de padres de tamaño μ y la población de hijos de tamaño λ. Posteriormente, se seleccionan a los μ mejores individuos de acuerdo a su valor de aptitud.
- (μ, λ). En este esquema la población de hijos reemplazará la población de padres. Para ello, se seleccionan a los μ mejores hijos considerando su aptitud. Es importante hacer notar que se debe cumplir que $\mu \leq \lambda$.

6.7. Aplicaciones

6.7.1. Función de Beale

Función objetivo

La función objetivo recibe como entrada el arreglo de variables de decisión $\boldsymbol{x}$. El cálculo de la función objetivo se muestra en la ecuación (2.5). Sea $\boldsymbol{x} = [2.3, 3.4]$, calculamos $f(\boldsymbol{x})$ como sigue:

- $f(x) = (1.5 - 2.3 + 2.3 \cdot 3.4)^2 + (2.25 - 2.3 + 2.3 \cdot 3.4^2)^2 + (2.625 - 2.3 + 2.3 \cdot 3.4^3)^2 = 8984.42$.

Función de aptitud

La función de aptitud F_a mide qué tan bueno es un individuo con respecto al problema de optimización $f(x)$ que se desea resolver. Regularmente, la función de aptitud debe

regresar un valor mayor o igual que cero y entre mayor sea el valor de aptitud se considera que se trata de un mejor individuo. Para la función de Beale, vamos a definir la aptitud como sigue:

$$F_a(x) = \frac{1}{f(x)+1} \tag{6.5}$$

Sea $\boldsymbol{x} = [2.3, 3.4]$, calculamos $F_a(\boldsymbol{x})$ como sigue:

- $F_a(x) = \frac{1}{f(x)+1} = \frac{1}{8984.42+1} \approx 0$

Representación de una solución

Cada solución se representa con dos arreglos de números reales, $\boldsymbol{x}$ y $\boldsymbol{\sigma}$, ambos de tamaño $n = 2$. $\boldsymbol{x}$ contiene los valores que toman las variables de decisión y $\boldsymbol{\sigma}$ los tamaños de desplazamiento para cada variable de decisión. Por ejemplo:

$$\boldsymbol{x} = [2.3, 3.4]$$

$$\boldsymbol{\sigma} = [0.12, 0.37]$$

Población inicial

La población inicial se genera de la siguiente forma. El arreglo $\boldsymbol{x}$, de cada individuo, se llena con números aleatorios generados con una distribución uniforme en el intervalo $[-4.5, 4.5]$. El arreglo $\boldsymbol{\sigma}$ se genera de la misma forma pero en el intervalo $[0, 1]$. A continuación se muestra un ejemplo.

Población inicial de tamaño 3:

$$P_{inicial} = \begin{bmatrix} [-3.45, 2.34] & [0.13, 0.78] \\ [2.45, -2.34] & [0.45, 0.67] \\ [-1.29, 0.79] & [0.57, 0.98] \end{bmatrix} \tag{6.6}$$

Cruza

Para generar un nuevo individuo se utilizará:

- **Cruza intermedia** para el arreglo $\boldsymbol{x}$
- **Cruza discreta** para el tamaño de desplazamiento $\boldsymbol{\sigma}$

A continuación se muestra un ejemplo. Sean:

$$P = \begin{bmatrix} P_1 = & [-3.45, 2.34] & [0.13, 0.78] \\ P_2 = & [2.45, -2.34] & [0.45, 0.67] \end{bmatrix}$$

los individuos que actuarán como padres y $r = [1, 0]$ la secuencia de 1's y 0's que simulan el lanzamiento de una moneda. El arreglo $\boldsymbol{x_H} = [-0.5, 0]$ se obtiene como sigue:

1. La componente uno es igual a $\frac{x_{1,1}+x_{2,1}}{2} = \frac{-3.45+2.45}{2} = -0.5$.
2. La componente dos es igual a $\frac{x_{1,2}+x_{2,2}}{2} = \frac{2.34+(-2.34)}{2} = 0$.

El arreglo $\boldsymbol{\sigma_H} = [0.13, 0.67]$ se obtiene como sigue:

1. Para la componente uno, se simula un volado. Si cae 1, se copia la información del padre P_1, de lo contrario se toma del padre P_2. Dado que $r_1 = 1$, agregamos el valor de P_1 en $\boldsymbol{\sigma_H} = [0.13]$.
2. Para la componente dos, repetimos el proceso. Se obtuvo $r_2 = 0$, por lo tanto, $\boldsymbol{H_\sigma} = [0.13, 0.67]$.

Por lo tanto, el hijo obtenido es $H = [[-0.5, 0], [0.13, 0.67]]$.

Mutación

Dado que se está empleando un tamaño de paso por variable de decisión, utilizaremos la ecuación (6.4). Si $H = [[-0.5, 0], [0.13, 0.67]]$ es el hijo que se desea mutar, el proceso de mutación es como sigue. Primero se debe realizar la mutación de los tamaños de paso, las tasas de aprendizaje que se van a utilizar son: $\tau^{'} = \frac{1}{\sqrt{2n}} = 0.5$ y $\tau = \frac{1}{\sqrt{2\sqrt{n}}} = 0.59$. Supongamos que $[0.17, 0.23, 0.67]$ es un arreglo de números aleatorios generados con una distribución normal con media 0 y desviación estándar 1, las componentes del vector σ se obtienen de la siguiente forma:

1. La componente uno es igual a $\sigma_1 \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_0(0,1)} = 0.13 \cdot e^{0.5 \cdot 0.17 + 0.59 \cdot 0.23} = 0.16$.
2. La componente dos es igual a $\sigma_2 \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_1(0,1)} = 0.67 \cdot e^{0.5 \cdot 0.17 + 0.59 \cdot 0.67} = 1.08$.

Por lo tanto, $\boldsymbol{\sigma'}_H = [0.16, 1.08]$. Sea 0.38 un número aleatorio generado con una distribución normal con media 0 y desviación estándar 1, el proceso de mutación de las variables de decisión queda de la siguiente forma.

1. La componente uno es igual a $x_1 + \sigma_1' \cdot N(0,1) = -0.5 + 0.16 \cdot 0.38 = -0.43$
2. La componente dos es igual a $x_2 + \sigma_2' \cdot N(0,1) = 0 + 1.08 \cdot 0.38 = 0.41$

Por lo tanto, el hijo mutado queda como $H' = [[-0.43, 0.41], [0.16, 1.08]]$.

Soluciones inválidas

Cada vez que se genera un nuevo individuo es importante revisar que los valores tanto en su componente $\boldsymbol{x}$ como en su componente $\boldsymbol{\sigma}$ sean válidos. En el caso de la función de Beale, los valores de las variables de decisión x_i deben cumplir: $\forall x_i \in [-4.5, 4.5]$. Para este ejercicio, si alguna de las variables no cumple con esta condición, se ajusta a la cota superior o inferior. En el caso de los valores de σ_i. Se revisa si $|\sigma_j^{'}| < \epsilon_0$, en caso de que se cumpla, se actualiza con $\sigma_j^{'} = \epsilon_0$.

Selección de sobrevivientes

El proceso de selección empleado es una *selección más*. Se une la población de **padres** y la población de **hijos**. Posteriormente, se ordenan los individuos con respecto al valor de aptitud y seleccionamos los μ mejores individuos. Donde μ es el tamaño de la población de **padres**.

- Sea P_X la población de padres y P_Y la población de hijos. Cada fila representa un individuo, la primera posición almacena el arreglo $\boldsymbol{x}$, la segunda el arreglo $\boldsymbol{\sigma}$ y la tercera el valor de aptitud.

$$P_X = \begin{bmatrix} [-3.45, 2.34] & [0.13, 0.78] & 0.00 \\ [2.45, -2.34] & [0.45, 0.67] & 0.00 \\ [-1.29, 0.79] & [0.57, 0.98] & 0.04 \end{bmatrix} \tag{6.7}$$

$$P_Y = \begin{bmatrix} [-0.5, 0] & [0.13, 0.67] & 0.04 \\ [0.58, -0.77] & [0.45, 0.98] & 0.11 \\ [-2.37, 1.56] & [0.57, 0.78] & 0.05 \end{bmatrix} \tag{6.8}$$

- La nueva población está conformada por los individuos:

$$P_n = \begin{bmatrix} [-1.29, 0.79] & [0.57, 0.98] & 0.04 \\ [-2.37, 1.56] & [0.57, 0.78] & 0.05 \\ [0.58, -0.77] & [0.45, 0.98] & 0.11 \end{bmatrix} \tag{6.9}$$

Estudio experimental y resultados

En este trabajo de tesis se implementó la clase *EvolutionStrategy*, la cual forma parte de la librería *pyristic*. En el apéndice A.4.4, se puede consultar la documentación. Para estudiar el comportamiento de nuestra metaheurística, en este problema, se ejecutó 30 veces con los siguientes parámetros:

- Número de generaciones: 200
- Tamaño de la población μ: 80
- Tamaño de la población λ: 160
- Esquema de selección: $(\mu + \lambda)$

Los resultados obtenidos son:

- *Valor de la función objetivo por la mejor solución:* 7.65172802912719e-24
- *Valor promedio:* 4.2654367173737946e-13
- *Desviación estándar:* 1.9074148651240894e-12

- *Valor de la función objetivo obtenido por la peor solución:* 8.956762249866645e-12

El óptimo global de este problema es en $\boldsymbol{x} = [3, 0.5]$ y $f(\boldsymbol{x}) = 0$. Dado que la media es en $f(x) = 0$ y la desviación estándar obtenida es muy pequeña, podemos concluir que la metaheurística propuesta resuelve la función de Beale exitosamente.

6.7.2. Función de Ackley

A continuación se presenta una propuesta para resolver la función de Ackley con 10 variables de decisión. Para este ejemplo, reutilizaremos los siguientes componentes descritos en la sección anterior.

- Representación de una solución
- Generación de la población inicial
- Recombinación para la solución y el tamaño de desplazamiento
- Mutación de la solución y tamaño de desplazamiento
- Selección de sobrevivientes

Función objetivo

La función objetivo recibe como entrada el arreglo $\boldsymbol{x}$ del individuo. La definición de la función objetivo se muestra en la ecuación (2.6).
Sea $x = [-7.05, -12.97, -23.93, 22.78, -11.70, 23.25, -25.49, -29.66, 10.13, 6.30]$

- $f(x) = -20 \exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=0}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=0}^{n}\cos(2\pi x_i)\right) + 20 + e$
  $= 21.11$

Función de aptitud

En este caso, solo vamos a convertir el problema de minimización en uno de maximización: $F_a(\boldsymbol{x}) = -f(\boldsymbol{x})$, donde, $f(\boldsymbol{x})$ es la función de Ackley. Es importante observar que no estamos forzando a que $F_a(\boldsymbol{x}) \geq 0$. Sea $x = [-7.05, -12.97, -23.93, 22.78, -11.70, 23.25, -25.49, -29.66, 10.13, 6.30]$

- $F_a(x) = -f(x) = -21.11$

Estudio experimental y resultados

En el apéndice A.4.6, se puede consultar la documentación. Este problema se abordó utilizando dos EE: $(\mu + \lambda) - EE$ y $(1 + 1) - EE$. A continuación se muestran los parametros de entrada utilizados.

- $(\mu + \lambda) - EE$
  - Número de generaciones: 250
  - Tamaño de población μ: 100
  - Tamaño de población λ: 200
- $(1 + 1) - EE$
  - Número de generaciones: 10,000
  - Valor c: 0.83
  - Valor k: 20

Para cada EE se hicieron 30 corridas independientes. En la siguiente tabla se muestran los resultados obtenidos.

	$(1+1)-EE$	$(\mu+\lambda)-EE$
Valor de la función objetivo por la mejor solución	16.959	0.008
Valor promedio	19.547	0.081
Desviación estándar	0.664	0.079
Valor de la función objetivo obtenido por la peor solución	20.235	0.327

Recordemos que el óptimo global de este problema está en $\boldsymbol{x} = [0, \ldots, 0]$ y $f(\boldsymbol{x}) = 0$. Como podemos observar, la EE más simple obtiene los peores resultados y se queda estancada en mínimos locales. Por el contrario, la versión poblacional con un tamaño de paso por variable de decisión y una selección más, logra llegar muy cerca del mínimo. Además, podemos decir que su comportamiento es robusto dado que la desviación estándar con respecto al resultado de cada ejecución es pequeña. Para intentar mejorar los resultados, se recomienda realizar un ajuste de parámetros.

Capítulo 7

Algoritmos Genéticos

Los *algoritmos genéticos* (AG) fueron propuestos por John H. Holland [9] y forman parte de los tres paradigmas de la *computación evolutiva*. Estos están inspirados en la evolución de los seres vivos, donde, los individuos compiten por los recursos limitados. Regularmente, los individuos más fuertes logran sobrevivir y reproducirse, pasando así, su información genética a las nuevas generaciones. El proceso evolutivo de los individuos, en las primeras versiones de los AG, es visto a nivel genotipo: cada individuo de la población se representa con una cadena binaria conocida como *cromosoma*. Posteriormente, se aplican operadores genéticos a los cromosomas de los individuos para generar nuevos individuos. Una diferencia entre EE y AG es el proceso de selección de padres. En EE cada individuo tiene la misma probabilidad de reproducirse. En los AG, la aptitud[1] determina la probabilidad que tiene un individuo para reproducirse, es decir, los más aptos tendrán más posibilidades de replicar su información genética.
En este capítulo se explicarán brevemente los principales componentes de un AG y se aplicarán en dos problemas de optimización continua y uno de optimización discreta.

7.1. Descripción del algoritmo

Los AG son un proceso iterativo. Se inicia con un conjunto de individuos que conforman la población P. Cada individuo es visto como una solución a nuestro problema y tiene asociado un valor de aptitud. Posteriormente, se selecciona un conjunto S de la población P empleando un *método de selección de padres*. A los individuos que están en S se les aplica un *operador de recombinación o cruza*, este último puede ser sexual o asexual. El modo de recombinación se decide definiendo una probabilidad p_c de ser sexual y una probabilidad $1 - p_c$ de ser asexual. En la recombinación sexual, se emparejan dos individuos (padres) de la población S para generar dos nuevos individuos (hijos). Una vez generados los hijos a través del operador de cruza, se les aplica un

[1] La aptitud refleja qué tan buena es una solución en comparación con las demás soluciones, considerando el problema que se está optimizando.

operador de mutación. Este operador realiza pequeñas alteraciones en la información genética de los hijos. La población de hijos es etiquetada como D. Finalmente, se seleccionan los individuos que se mantendrán en la siguiente generación (sobrevivientes). Para ello hay diferentes esquemas de selección en los que se puede utilizar únicamente la población de hijos D o la unión de las poblaciones D y P. Finalmente, se repite este proceso durante un determinado número de generaciones. El algoritmo 6 muestra de manera general como opera un AG. En las siguientes secciones, se explicará de manera detallada los principales componentes de un AG.

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.
pob_actual $\leftarrow$ Población inicial de tamaño μ;
Asignar aptitud a cada individuo de **pob_actual** considerando $f(\boldsymbol{x})$;
$t \leftarrow 1$;
while $t <= G$ **do**
  padres $\leftarrow$ Utilizar un método de selección de padres para determinar los μ individuos, de **pob_actual**, que actuarán como padres;
  hijos $\leftarrow \emptyset$;
  foreach $i, j \in$ **padres do**
    if $random(0,1) < p_c$ **then**
      Aplicar operador de cruza a i y j y generar dos hijos h_1 y h_2;
    else
      Generar dos hijos h_1 y h_2 de manera asexual: $h_1 \leftarrow i$, $h_2 \leftarrow j$;
    Aplicar operador de mutación a h_1 y h_2, usando p_m;
    hijos $\leftarrow$ **hijos** $\cup\{h_1, h_2\}$;
  pob_actual $\leftarrow$ Seleccionar los μ individuos sobrevivientes;
  $t \leftarrow t + 1$;
$x_{best} \leftarrow$ Mejor individuo de la población actual;
Regresar x_{best}, $f(x_{best})$;

7.2. Representación

Los primeros AG representaban cada individuo con una cadena binaria y tenían mecanismos para transformar dicha cadena en los valores asociados a cada variable del problema. Es decir pasar del genotipo al fenotipo. Esto implicaba costos computacionales adicionales, lo cual motivó el surgimiento de nuevas propuestas. A continuación se explican las representaciones empleadas en este trabajo de tesis.

- **Representación real para problemas de optimización continua**. Cada solución

es representada por un arreglo con valores reales, donde, cada componente del arreglo está asociado a una variable de decisión del problema.

- **Representación de números naturales para permutaciones**. Cada solución es un acomodo de n valores naturales, comenzando del número 0 hasta el número $n-1$.

7.3. Métodos de selección de padres

Actualmente, existen varios métodos de selección de padres. En este trabajo solo se abordan cuatro de ellos: ruleta, universal estocástica, muestreo determinístico y selección por torneo. Estos métodos de selección de padres calculan el valor esperado de cada individuo de la siguiente forma: $E_i = \frac{f_i}{f_m}$, donde f_i es la aptitud del individuo i y f_m es la aptitud promedio de la población. Sea $p_i = \frac{f_i}{\sum_j f_j}$ la probabilidad de seleccionar al individuo i, el valor esperado E_i de un individuo indica cuántas copias se esperan obtener de ese individuo, si selecciono aleatoriamente n individuos considerando su probabilidad de selección.

7.3.1. Método de la ruleta

Este método se inspira en girar una ruleta repetidamente, donde, cada segmento de la ruleta representa la probabilidad de selección de un individuo. Ver algoritmo 7. El algoritmo recibe un vector con la aptitud de cada uno de los individuos, la aptitud en la posición i está asociada al individuo i. Con este método es posible que los peores individuos actúen como padres e incluso podrían participar en la generación de más

de un hijo.

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**
        $i \leftarrow i + 1$;
        $suma \leftarrow suma + E_i$;
    Seleccionar al individuo i como padre:
    padres $\leftarrow$ **padres** $\cup\{i\}$;
Regresar **padres**;

7.3.2. Universal estocástica

Este método tiene como objetivo que el número de copias que se generan de cada individuo (veces que actúa como padre) sea muy cercano a su valor esperado. Para ello calcula el valor esperado de cada individuo E_i y genera un valor aleatorio r en el intervalo $[0, 1)$ utilizando una distribución uniforme. Posteriormente, para cada individuo se crean al menos $\lfloor E_i \rfloor$ copias y el valor aleatorio r decidirá si se realiza una copia más.

Ver algoritmo 8.

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** $\cup\{i\}$;
        $r \leftarrow r + 1$;
Regresar **padres**;

7.3.3. Muestreo determinístico

El objetivo del muestreo determinístico es acercarse a los valores esperados calculados a partir de la aptitud de los individuos. Sea E_i el valor esperado de cada individuo i. Se toma la parte entera del valor esperado del individuo i y será el número de copias del individuo i. Para completar el número de padres solicitado, se seleccionan los individuos restantes considerando la expansión decimal del valor esperado de los individuos. Los individuos con un mayor valor son seleccionados primero. Ver algo-

ritmo 9.

Algoritmo 9: Muestreo determinístico

Entrada: Población P de tamaño N.
Salida : Población de N 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 $\leftarrow \emptyset$;
foreach $i \in P$ **do**
    padres $\leftarrow$ **padres** $\cup$ {Parte entera de E_i copias de i};
if *tamaño de* **padres** $< N$ **then**
    Ordenar de manera decreciente la población P por la expansión decimal ;
    i $\leftarrow 0$;
    while *tamaño de* **padres** $< N$ **do**
        padres $\leftarrow$ **padres** $\cup$ { P[i] };
        i $\leftarrow$ i + 1;
Regresa **padres**

7.3.4. Selección por torneo

Existen dos versiones de este método: determinística y probabilística. La versión determinística genera una presión de selección fuerte. El peor individuo tiene una probabilidad 0 de ser seleccionado como padre. El método consiste en crear k grupos, donde cada grupo tiene s individuos. Posteriormente, se realiza un torneo por grupo. La versión determinística selecciona siempre al mejor individuo de cada torneo y repite el proceso hasta completar el número de padres requerido. En la versión probabilística se define una probabilidad p de seleccionar al mejor individuo en los torneos. En cada torneo, se simula un volado pesado considerando p y se decide si tomar al mejor o al peor individuo de cada grupo. Ver algoritmo 10.

Algoritmo 10: Selección por torneo

Entrada: Población P de tamaño N, tamaño del torneo s y probabilidad de seleccionar al mejor p.
Salida : Población de N individuos seleccionados.
Sea Fa_i la aptitud del individuo i;
padres $\leftarrow \emptyset$;
for $i \leftarrow 1$ *To* s **do**
  Generar K conjuntos de tamaño s;
  for $k_i \in K$ **do**
    if *rand(0,1)* $\leq p$ **then**
      padres $\leftarrow$ **padres** $\cup$ { Individuo del conjunto k_i con mayor Fa};
    else
      padres $\leftarrow$ **padres** $\cup$ { Individuo del conjunto k_i con menor Fa};
Regresar **padres**;

7.4. Operadores de recombinación o cruza

Los operadores de recombinación simulan el proceso de cruza efectuado en los sistemas biológicos. Para ello generan soluciones hijas intercambiando segmentos de información de las soluciones padre. A continuación se describen algunos operadores de cruza.

7.4.1. Cruza de n puntos

Este operador es una generalización de la cruza de un punto propuesta por Holland [8]. Originalmente, se aplicó a individuos con representación binaria. Dados dos padres, se crean dos nuevos individuos. Para ello se seleccionan de manera aleatoria n puntos de cruza. Los hijos van copiando posición a posición la información de uno de los padres, cada vez que se encuentra un punto de cruza, intercambian el padre del cual están realizando la copia. El primer hijo comienza con la información del primer padre y el segundo hijo comienza con la información del segundo padre. En la figura 7.1 se muestra un ejemplo cuando $n = 2$.

7.4.2. Cruza uniforme

Fue propuesta por Ackley en 1987, originalmente se aplicaba en individuos con representación binaria. Sean P_1 y P_2 los padres y p_c la probabilidad de intercambio. Este operador genera dos hijos H_1 y H_2 como sigue: copia la información del padre $P1$ en $H1$ y la información del padre $P2$ en $H2$. Después, se recorre cada posición de la cadena binaria y se genera un valor aleatorio r con distribución uniforme entre $[0, 1)$. Si el valor aleatorio $r \leq pc$, se intercambian los padres.

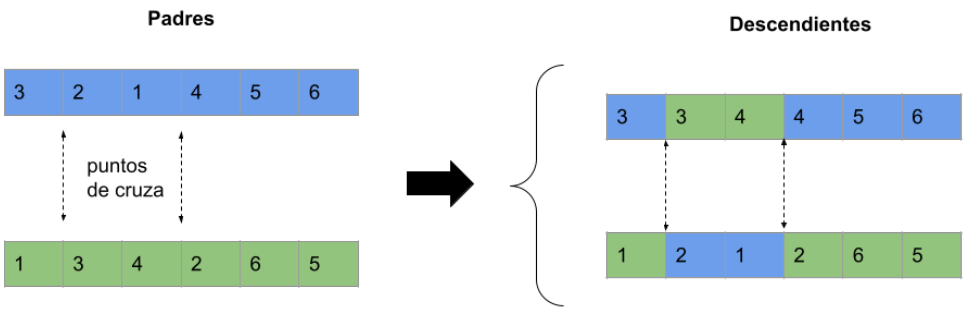


Figura 7.1: Cruza de 2 puntos

$$H_{1,i}, H_{2,i} = \begin{cases} P_{2,i}, P_{1,i} & \text{Si } R_i \leq p_c \\ P_{1,i}, P_{2,i} & \text{Si } R_i > p_c \end{cases} \tag{7.1}$$

7.4.3. Cruza intermedia completa

Este tipo de cruza se aplica cuando se está trabajando con representación real. Sean P_1 y P_2 las soluciones padres y α un valor entre $(0, 1)$, obtenemos dos soluciones hijas de la siguiente forma:

$$H_1 = \alpha \cdot P_1 + (1 - \alpha) \cdot P_2$$
$$H_2 = \alpha \cdot P_2 + (1 - \alpha) \cdot P_1$$

7.4.4. Cruza binaria simulada (simulated binary crossover)

Este tipo de cruza fue propuesta para representación real. Dadas dos soluciones padres P_1 y P_2 se generan dos soluciones hijas H_1 y H_2 de la siguiente forma:

$$H_1 = 0.5[(P_1 + P_2) - \beta|P_2 - P_1|]$$
$$H_2 = 0.5[(P_1 + P_2) + \beta|P_2 - P_1|]$$

Donde β se define como sigue:

$$\beta = \begin{cases} (2u)^{\frac{1}{n_c+1}} & \text{Si } u \leq 0.5, \\ \left(\frac{1}{2(1-u)}\right)^{\frac{1}{n_c+1}} & \text{Si } u > 0.5 \end{cases} \tag{7.2}$$

Regularmente, n_c es igual a 1 ó 2 y $u \in [0, 1]$.

7.4.5. Order crossover

Fue propuesta por Davis en 1985 y se utiliza en individuos con representación entera para permutaciones. Dado dos padres P_1 y P_2, se generan dos individuos H_1 y H_2 como sigue: Para el hijo H_1, selecciona un segmento de longitud variable de manera aleatoria del padre P_1, este segmento es copiado a H_1 en las mismas posiciones. Las posiciones restantes son completadas con la información del padre P_2, de izquierda a derecha, sin considerar los elementos que aparecen en el segmento copiado del padre P_1. Para el segundo hijo H_2, se realiza el mismo procedimiento pero intercambiando a los padres. La figura 7.2 muestra un ejemplo.

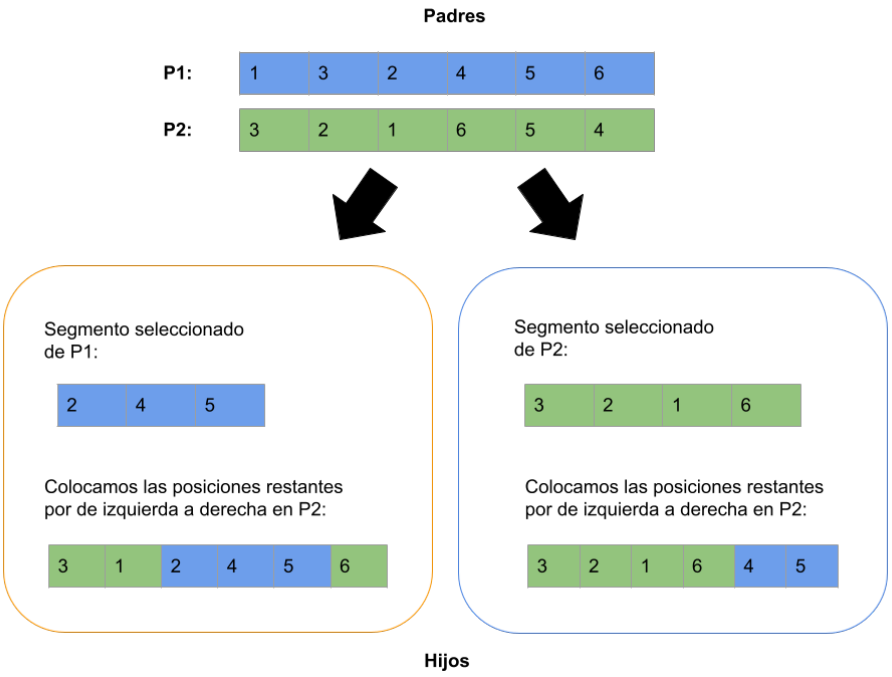


Figura 7.2: Ejemplo de cruza por orden

7.5. Operadores de mutación

La mutación se considera como un operador secundario en los AG y simula los errores de copiado que existen en los sistemas biológicos. A continuación se describen algunos operadores de mutación.

7.5.1. Mutación para representación binaria

Sea X un arreglo con una secuencias de 0's y 1's que representa la información genética de la solución hija. El arreglo mutado $X^{'}$ se genera como sigue: Visitamos cada posición del arreglo X y se modifica su valor con una probabilidad p_m. Si el arreglo en dicha posición contiene un 0, este se sustituye por un 1, de lo contrario, es reemplazado por un 0.

7.5.2. Mutación para representación real: de límite

Sea X un arreglo de longitud n con la información de la solución hija, donde n es el número de variables de decisión. El arreglo mutado $X^{'}$ se obtiene como sigue: Seleccionamos una de las n posiciones de X de manera aleatoria y la ajustamos al límite superior con una probabilidad de 0.5, de lo contrario, al límite inferior.

$$X_k^{'} = \begin{cases} LB & \text{Si } R \leq 0.5 \\ UB & \text{Si } R > 0.5 \end{cases} \tag{7.3}$$

donde $X_k^{'}$ representa la variable de decisión seleccionada, R es un número aleatorio en el intervalo $[0, 1)$, generado con una distribución uniforme, LB y UB son los límites inferior y superior, respectivamente, de la variable $X_k^{'}$.

7.5.3. Mutación para representación real: uniforme

Sea X un arreglo de longitud n con la información de la solución hija, donde, n es el número de variables de decisión. El arreglo $X^{'}$ tras la mutación se obtiene como sigue: Seleccionamos una de las n posiciones de X de manera aleatoria y reemplazamos el valor que tiene por un valor aleatorio con distribución uniforme entre los límites permisibles de la variable de decisión seleccionada.

7.5.4. Mutación para representación real: no uniforme

La mutación no uniforme agrega un ruido δ a cada variable de decisión. El valor δ se puede generar con una distribución normal con media 0 y desviación estándar 1. Sea X el arreglo con la información del hijo, el hijo mutado se genera de la siguiente forma:

$$\boldsymbol{X_i'} = \boldsymbol{X_i} + \delta_i \quad \forall i \in [0, \ldots, n] \tag{7.4}$$

donde, n es el número de variables de decisión. Existen otras propuestas de mutación no uniforme. Por ejemplo, Michalewicz propuso en 1992 ajustar el grado de perturbación según la generación. En las primeras generaciones, la perturbación a las variables es mayor y en las últimas generaciones es muy pequeña.

7.5.5. Mutación para permutaciones: por desplazamiento

Es una generalización de la mutación por inserción. La mutación por inserción consiste en seleccionar aleatoriamente un elemento de la permutación y una nueva posición. Posteriormente, coloca el elemento en la nueva posición y desplaza el resto de los elementos hacia la derecha. La mutación por desplazamiento realiza k inserciones, donde k es un parámetro que define el usuario. Ver figura 7.3.

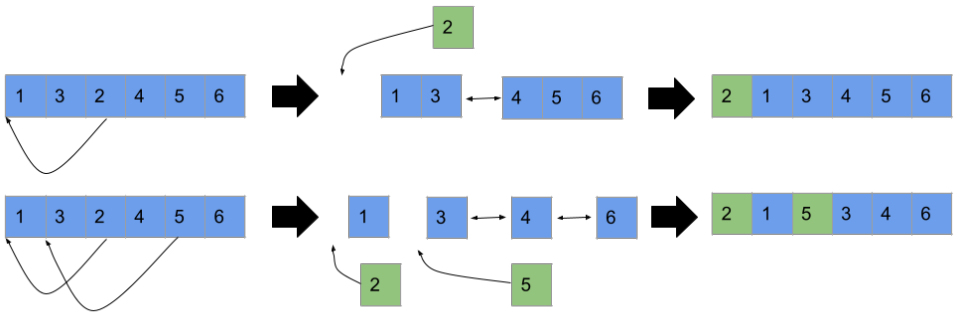


Figura 7.3: La imagen superior muestra un ejemplo de mutación por inserción. La imagen inferior es un ejemplo de mutación por desplazamiento con $k = 2$.

7.5.6. Mutación para permutaciones: por intercambio

Dada una permutación, intercambia dos posiciones seleccionadas de manera aleatoria. Las posiciones restantes permanecen igual. Ver figura 7.4.

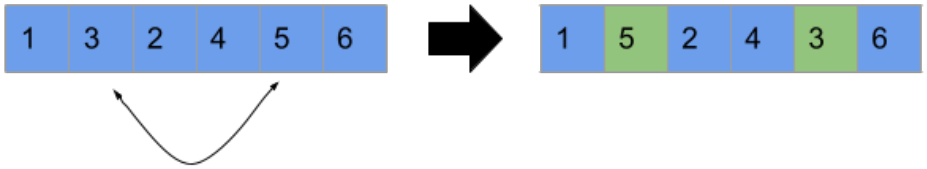


Figura 7.4: Mutación por intercambio.

7.6. Esquemas de selección de sobrevivientes

Al igual que en las estrategias evolutivas los dos esquemas de sobrevivientes más utilizados son:

- $(\mu + \lambda)$. En este esquema se une la población de padres de tamaño μ y la población de hijos de tamaño λ. Posteriormente, se seleccionan a los μ mejores individuos de acuerdo a su valor de aptitud.

- (μ, λ). En este esquema la población de hijos reemplazará la población de padres. Para ello, se seleccionan a los μ mejores hijos considerando su aptitud. Es importante hacer notar que se debe cumplir que $\mu \leq \lambda$.

7.7. Aplicaciones

7.7.1. Función de Beale

En este caso usaremos la misma representación y función de aptitud descritas en los capítulos 5 y 6.

Población inicial

La población inicial se genera de la siguiente forma. El arreglo, de cada individuo, se llena con números aleatorios generados con una distribución uniforme en el intervalo $[-4.5, 4.5]$. A continuación se muestra un ejemplo donde cada fila representa un individuo de la población.

$$P_{inicial} = \begin{bmatrix} [-3.45, 2.34], 0.00 \\ [2.45, -2.34], 0.00 \\ [-1.29, 0.79], 0.04 \\ [3.49, -3.94], 0.00 \\ [1.27, -0.67], 0.21 \\ [2.13, 0.19], 0.74 \end{bmatrix} \tag{7.5}$$

Selección de padres

En este problema, utilizaremos una selección por torneo, donde, cada conjunto es de tamaño $k = 3$ y la probabilidad de seleccionar al individuo con mayor aptitud dentro de un grupo es $p = \frac{1}{2}$, si no, se selecciona al individuo con menor aptitud en ese grupo. Por ejemplo:

Sea:

$$P_{inicial} = \begin{bmatrix} [-3.45, 2.34], 0.00, 0 \\ [2.45, -2.34], 0.00, 1 \\ [-1.29, 0.79], 0.04, 2 \\ [3.49, -3.94], 0.00, 3 \\ [1.27, -0.67], 0.21, 4 \\ [2.13, 0.19], 0.74, 5 \end{bmatrix} \tag{7.6}$$

la población actual (la primera posición es la solución $\boldsymbol{x}$, la segunda la aptitud y la tercera el índice del individuo) y $r = [1, 1, 0, 0, 1, 0]$ la secuencia de 1's y 0's que simulan el lanzamiento de una moneda. El arreglo con los índices de los padres seleccionados $P_s = [2, 5, 3, 1, 4, 0]$ se obtiene como sigue:

- **Ronda 1:** Se realiza un acomodo de la población, el nuevo acomodo se divide en dos conjuntos de tamaño $k = 3$. Los conjuntos obtenidos son:

$$C1 = \begin{bmatrix} [-3.45, 2.34], 0.00, 0 \\ [-1.29, 0.79], 0.04, 2 \\ [3.49, -3.94], 0.00, 3 \end{bmatrix}$$

$$C2 = \begin{bmatrix} [2.45, -2.34], 0.00, 1 \\ [1.27, -0.67], 0.21, 4 \\ [2.13, 0.19], 0.74, 5 \end{bmatrix}$$

  1. Para seleccionar el primer individuo, se simula un volado. Si cae 1, se selecciona al individuo con el mayor valor de aptitud, de lo contrario, se toma al que tiene el menor valor de aptitud. Dado que $r_0 = 1$, seleccionamos $[[-1.29, 0.79], 0.04, 2]$, agregamos el índice en $P_s = [2]$
  2. Para seleccionar al segundo individuo, repetimos el proceso. Se obtiene $r_1 = 1$, se añade $[[2.13, 0.19], 0.74, 5]$ a $P_s = [2, 5]$

- **Ronda 2:** Se realiza de nuevo un acomodo de la población y dividimos en dos conjuntos de tamaño $k = 3$ la población. Los conjuntos obtenidos son:

$$C1 = \begin{bmatrix} [-3.45, 2.34], 0.00, 0 \\ [1.27, -0.67], 0.21, 4 \\ [3.49, -3.94], 0.00, 3 \end{bmatrix}$$

$$C2 = \begin{bmatrix} [2.45, -2.34], 0.00, 1 \\ [-1.29, 0.79], 0.04, 2 \\ [2.13, 0.19], 0.74, 5 \end{bmatrix}$$

  1. Para seleccionar el tercer individuo[2], se repite el volado. Dado que $r_2 = 0$, seleccionamos $[[3.49, -3.94], 0.00, 3]$ y lo agregamos en $P_s = [2, 5, 3]$
  2. Para seleccionar al cuarto individuo, repetimos el proceso. Se obtiene $r_3 = 0$, se añade $[[2.45, -2.34], 0.00, 1]$ a $P_s = [2, 5, 3, 1]$

[2]Es importante mencionar que el individuo con índice 3 tiene un mejor valor de aptitud. Por cuestiones de espacio, se decidió solo incluir dos décimales en todos los resultados presentados en este trabajo.

- **Ronda 3:** Se realiza de nuevo un acomodo de la población y dividimos en dos conjuntos de tamaño $k = 3$ la población. Los conjuntos obtenidos son:

$$C1 = \begin{bmatrix} [1.27, -0.67], 0.21, 4 \\ [3.49, -3.94], 0.00, 3 \\ [2.45, -2.34], 0.00, 1 \end{bmatrix}$$

$$C2 = \begin{bmatrix} [-3.45, 2.34], 0.00, 0 \\ [2.13, 0.19], 0.74, 5 \\ [-1.29, 0.79], 0.04, 2 \end{bmatrix}$$

1. Para seleccionar el quinto individuo, se repite el volado. Dado que $r_4 = 1$, seleccionamos $[[1.27, -0.67], 0.21, 4]$ y lo agregamos en $P_s = [2, 5, 3, 1, 4]$
2. Para seleccionar el sexto individuo, repetimos el proceso. Se obtiene $r_5 = 0$,entonces, se añade $[2, 5, 3, 1, 4, 0]$.

Cruza

Para generar un nuevo individuo vamos a emplear el operador de *cruza intermedia* un una probabilidad de cruza $p_c = 1$. Es decir, los hijos siempre se van a generar de manera sexual. A continuación se muestra un ejemplo. Sean:

$$P = \begin{bmatrix} P_1 = & [-3.45, 2.34] \\ P_2 = & [2.45, -2.34] \end{bmatrix}$$

los individuos que actuarán como padres.
El arreglo $\boldsymbol{x_H} = [-0.5, 0]$ se obtiene como sigue:

1. La componente uno es igual a $\frac{x_{1,1}+x_{2,1}}{2} = \frac{-3.45+2.45}{2} = -0.5$.
2. La componente dos es igual a $\frac{x_{1,2}+x_{2,2}}{2} = \frac{2.34+(-2.34)}{2} = 0$.

Es importante observar que este operador genera un único hijo por cada par de padres.

Mutación

Para resolver la función de Beale vamos a utilizar una mutación uniforme. Sea $\boldsymbol{x_H} = [-0.5, 0]$ la solución hija y $x_i \in [-4.5, 4.5]$.

- Seleccionamos al azar una variable de decisión. El índice seleccionado es $i = 1$.
- Generamos un número aleatorio entre -4.5 y 4.5, utilizando una distribución uniforme. Supongamos que el valor obtenido es $r = -0.89$.
- Actualizamos la posición $i = 1$ con el valor $r = -0.89$. La solución ahora es $x_H^{'} = [-0.5, -0.89]$

Solución inválida

Dado que vamos a utilizar una cruza intermedia y una mutación uniforme para obtener las soluciones hijas. Nuestra metaheurística no podrá generar soluciones fuera de los límites.

Selección de sobrevivientes

El proceso de selección empleado es una *selección más*. A continuación se muestra un ejemplo.

- Sea P_X la población de padres y P_Y la población de hijos.

$$P_X = \begin{bmatrix} [-3.45, 2.34] & 0.00 \\ [2.45, -2.34] & 0.00 \\ [-1.29, 0.79] & 0.04 \\ [3.49, -3.94] & 0.00 \\ [1.27, -0.67] & 0.21 \\ [2.13, 0.19] & 0.74 \end{bmatrix} \tag{7.7}$$

$$P_Y = \begin{bmatrix} [0.58, -0.77] & 0.11 \\ [1.86, -1.50] & 0.01 \\ [1.27, -0.67] & 0.21 \\ [-3.45, 2.34] & 0.00 \\ [1.86, -1.50] & 0.01 \\ [0.02, -0.80] & 0.06 \end{bmatrix} \tag{7.8}$$

- La nueva población está conformada por los individuos:

$$P_n = \begin{bmatrix} [-1.29, 0.79] & 0.04 \\ [0.02, -0.80] & 0.06 \\ [0.58, -0.77] & 0.11 \\ [1.27, -0.67] & 0.21 \\ [1.27, -0.67] & 0.21 \\ [2.13, 0.19] & 0.74 \end{bmatrix} \tag{7.9}$$

Estudio experimental y resultados

En este trabajo de tesis, se construyó la clase *Genetic* que forma parte de la librería *Pyristic*. Esta clase permite abordar problemas de optimización basados en AG. En el apéndice A.5.4, se puede consultar la documentación de la clase y la implementación que se realizó para este problema. Para estudiar el comportamiento de nuestra metaheurística se realizaron 30 ejecuciones independientes con los siguientes parámetros:

- Número de generaciones: 200

- Tamaño de la población μ: 100
- Probabilidad de cruza: 1
- Probabilidad de mutación: 1

Los resultados obtenidos fueron:

- *Valor de la función objetivo por la mejor solución:* 3.8528422849262045e-06
- *Valor promedio:* 0.0002520283655572658
- *Desviación estándar:* 0.0005704335576591761
- *Valor de la función objetivo obtenido por la peor solución:* 0.003221790920680413

El óptimo global de este problema es en $\boldsymbol{x} = [3, 0.5]$ y $f(\boldsymbol{x}) = 0$. Dado que la media es $f(x) = 0$ y la desviación estándar obtenida es muy pequeña, podemos concluir que la metaheurística propuesta resuelve la función de Beale exitosamente.

7.7.2. Función de Ackley

En este ejemplo vamos a utilizar la función de Ackley con $n = 10$ variables de decisión y reutilizaremos los siguientes componentes descritos en la sección anterior.

- Representación de una solución.
- Generación de la población inicial.
- Esquema de selección de padres.
- Operador de recombinación.
- Selección de sobrevivientes.

Función objetivo

La función objetivo recibe como entrada el arreglo $\boldsymbol{x}$ del individuo y su definición se muestra en la ecuación (2.6).
Sea $x = [-7.05, -12.97, -23.93, 22.78, -11.70, 23.25, -25.49, -29.66, 10.13, 6.30]$

- $f(x) = -20 \exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=0}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=0}^{n}\cos(2\pi x_i)\right) + 20 + e$
  $= 21.11$

Función de aptitud

En este caso, vamos a convertir el problema de minimización en uno de maximización: $F_a(\boldsymbol{x}) = -f(\boldsymbol{x})$, donde, $f(\boldsymbol{x})$ es la función de Ackley. Es importante observar que no estamos forzando a que $F_a(\boldsymbol{x}) \geq 0$. Por lo anterior, no podríamos usar un esquema de selección proporcional de manera directa.

Sea $x = [-7.05, -12.97, -23.93, 22.78, -11.70, 23.25, -25.49, -29.66, 10.13, 6.30]$

- $F_a(x) = -f(x) = -21.11$

Mutación

Para mutar los individuos vamos a utilizar una *mutación no uniforme*. Sea $\boldsymbol{x}$ la solución actual y $\boldsymbol{\delta}$ el vector con las perturbaciones de cada variable:

$$\boldsymbol{x} = [-17.83, -13.50, -3.66, -27.10, -16.40, -8.92, 18.34, -17.57, -12.81, 4.71]$$

$$\boldsymbol{\delta} = [-0.03, -0.05, 0.09, -2.60, 1.24, -0.10, 1.03, -1.27, -0.41, -0.49]$$

El individuo mutado queda como sigue:

$$\boldsymbol{x}' = [-17.86, -13.55, -3.57, -29.7, -15.16, -9.02, 19.37, -18.84, -13.22, 4.22]$$

Estudio experimental y resultados

En el apéndice A.5.5, se puede consultar la documentación de la clase y la implementación que se realizó para este problema. Para estudiar el comportamiento de nuestra metaheurística se realizaron 30 ejecuciones independientes con los siguientes parámetros.

- Número de generaciones: 200
- Tamaño de la población μ: 100
- Probabilidad de cruza: 1
- Probabilidad de mutación: 1

A continuación se presentan los resultados obtenidos:

- *Valor de la función objetivo por la mejor solución:* 4.440892098500626e-16
- *Valor promedio:* 4.440892098500626e-16
- *Desviación estándar:* 0.0

- *Valor de la función objetivo obtenido por la peor solución:* 4.440892098500626e-16

Podemos observar que nuestra metaheurística es robusta. En el problema con diez variables de decisión, logra encontrar el mínimo global que está en $x_i = 0$ y $f(\boldsymbol{x}) = 0$ en todas las ejecuciones.

7.7.3. Problema del Agente viajero

Consideremos la siguiente instancia del problema del agente viajero:

- Número de ciudades: 4
- Matriz de distancias entre las ciudades:

$$M = \begin{bmatrix} 0 & 2 & 1 & 4 \\ 2 & 0 & 3 & 4 \\ 1 & 3 & 0 & 4 \\ 4 & 4 & 4 & 0 \end{bmatrix}$$

A continuación se describe cada uno de los componentes que tendrá nuestro AG y se ejemplificará su funcionamiento utilizando esta instancia del problema. En el caso de la selección de padres y sobrevivientes se utilizará la descrita en el apartado 7.7.1.

Función objetivo

La función objetivo recibe como entrada el arreglo de variables de decisión $\boldsymbol{x}$. El cálculo de la función objetivo se muestra en la ecuación (2.4).
Sea $x = [3, 1, 2, 0]$, calculamos $f(x)$ como sigue:

- $f(\boldsymbol{x}) = \sum_{i=0}^{n-1} d(x_i, x_{i+1}) + d(x_n, x_0) = d(x_0, x_1) + d(x_1, x_2) + d(x_2, x_3) + d(x_3, x_0) = 4 + 3 + 1 + 4 = 12$

Función de aptitud

En este caso, solo vamos a convertir el problema de minimización en uno de maximización: $F_a(\boldsymbol{x}) = -f(\boldsymbol{x})$, donde, $f(\boldsymbol{x})$ es la función del problema del agente viajero. Es importante observar que no estamos forzando a que $F_a(\boldsymbol{x}) \geq 0$. Si $x = [3, 1, 2, 0]$, tenemos que $F_a(x) = -f(x) = -12$.

Representación de la solución

La solución se representa como un arreglo de longitud n, donde, n es el número de ciudades. Cada posición del arreglo indica la ciudad que será visitada. Ninguna de las ciudades puede ser visitada dos veces. El orden de visita se considera de izquierda a derecha. Las ciudades están enumeradas desde el 0 y hasta $n - 1$. Por ejemplo:

- La solución $\boldsymbol{x} = [0, 1, 2, 3]$ indica que el agente viajero comienza su recorrido en la ciudad con la etiqueta 0, después viaja a la ciudad 1 y sigue su recorrido hasta llegar a la ciudad con la etiqueta 3. El recorrido finaliza regresando de la ciudad con la etiqueta 3 a la ciudad con la etiqueta 0.

Población inicial

La población inicial se genera de la siguiente forma. Cada individuo será una permutación aleatoria de los números enteros en $[0, n)$, donde, n es el número de ciudades. La primera posición de la permutación corresponderá a la ciudad con la etiqueta 0. A continuación se muestra un ejemplo para $n = 4$.

$$\boldsymbol{P}_{\boldsymbol{inicial}} = \begin{bmatrix} [0,2,1,3] & -12 \\ [0,1,3,2] & -11 \\ [0,2,3,1] & -11 \\ [0,1,2,3] & -13 \end{bmatrix}$$

Cada fila representa un individuo, la primera columna indica la solución $\boldsymbol{x}$ y la segunda su valor de aptitud correspondiente $F_a(\boldsymbol{x})$.

Cruza

En este problema vamos a utilizar el operador de cruza *order crossover* y una probabilidad de cruza $p_c = 1$. A continuación se muestra un ejemplo.

Sean P_1 y P_2 las soluciones padre:

$$P = \begin{bmatrix} P_1 = & [0,2,1,3] \\ P_2 = & [0,1,2,3] \end{bmatrix}$$

- Para generar $H_1 = [0, 2, 1, 3]$ se realiza lo siguiente:

  1. Iniciamos nuestro individuo como $H_1 = [-1, -1, -1, -1]$ para indicar que no se ha introducido ningún elemento.
  2. Seleccionamos un segmento aleatorio de P_1. Supongamos que el segmento es $S = [1, 3]$. El segmento se introduce en las mismas posiciones que se encuentran en P_1. Nuestro individuo ahora es $H_1 = [-1, -1, 1, 3]$.
  3. En la posición 0 se seleccionará el primer elemento de P_2 que no ocurra en el segmento $S = [1, 3]$. Por lo tanto, nuestro individuo ahora es $H_1 = [0, -1, 1, 3]$.
  4. Para la posición 1 repetimos el mismo proceso. El primer elemento sin ocurrencia en S es el elemento en la posición 2. Por lo tanto, nuestro individuo ahora es $H_1 = [0, 2, 1, 3]$.

- Para generar $H_2 = [0, 1, 2, 3]$ se realiza de la siguiente forma:

  1. Iniciamos nuestro individuo como $H_2 = [-1, -1, -1, -1]$ para indicar que no se ha introducido ningún elemento.
  2. Seleccionamos el segmento aleatorio pero ahora de P_2. Nuestro individuo ahora es $H_2 = [-1, 1, 2, 3]$.
  3. En la posición 0 se seleccionará el primer elemento de P_1 que no ocurra en el segmento $S = [2, 3]$. Por lo tanto, nuestro individuo ahora es $H_1 = [0, -1, 2, 3]$.
  4. Para la posición 1 repetimos el mismo proceso. El siguiente elemento sin aparecer en S es el elemento 1. Por lo tanto, nuestro individuo queda como $H_2 = [0, 1, 2, 3]$.

Mutación

Para mutar las soluciones obtenidas por la recombinación, se realiza una mutación por inserción. A continuación se muestra un ejemplo.

Sea $H = [0, 2, 1, 3]$ nuestra solución a mutar. La solución obtenida tras la mutación $H^{'} = [0, 1, 2, 3]$ se obtiene como sigue:

- Iniciamos con $H^{'} = [-1, -1, -1, -1]$ para mostrar el proceso.
- Seleccionamos el elemento a desplazar. Supongamos que se selecciona el elemento en la posición 2.
- Seleccionamos la posición en la que se introducirá el elemento, supongamos que se selecciona la posición 1. Por lo tanto $H^{'} = [-1, 1, -1, -1]$
- Copiamos todos los elementos que se encontraban antes de la posición 1. Ahora $H^{'} = [0, 1, -1, -1]$.
- Introducimos los elementos que se encuentran después de la posición 1, sin considerar el elemento desplazado. Por lo tanto $H^{'} = [0, 1, 2, 3]$

Estudio experimental y resultados

En el apéndice A.5.6, se puede consultar la documentación de la clase y la implementación que se realizó para este problema. Para probar el desempeño de nuestro AG utilizamos la instancia aleatoria descrita en el apartado 3.2.2. Nuestro experimento constó de 30 ejecuciones independientes,
los parámetros que utilizamos son los siguientes:

- Número de generaciones: 200

- Tamaño de la población μ: 100
- Probabilidad de cruza: 1
- Probabilidad de mutación: 1

Los resultados obtenidos son:

- *Valor de la función objetivo por la mejor solución:* 248
- *Valor promedio:* 248
- *Desviación estándar:* 0
- *Valor de la función objetivo obtenido por la peor solución:* 248

Como podemos observar el AG llega siempre a la misma solución y es la mejor solución encontrada por las diferentes metaheurísticas estudiadas en este trabajo de tesis. Por lo anterior, podemos decir que nuestro AG es robusto y logra encontrar buenas soluciones para esta instancia del problema.

Capítulo 8

Resultados y conclusiones

En este capítulo presentamos un estudio comparativo de las metaheurísticas estudiadas en este trabajo de tesis. En primer lugar usaremos programación evolutiva, estrategias evolutivas y algoritmos genéticos para resolver la función de Ackley. Y en segundo lugar emplearemos búsqueda tabú, recocido simulado y algoritmos genéticos para resolver el problema de la mochila binario y el problema del agente viajero[1] pero con instancias de mayor tamaño. La función de Beale que se describió no se considera, dado que las tres metaheurísticas presentadas (PE, EE y AG) encuentran el óptimo global sin dificultad.

8.1. Función de Ackley

Nuestro estudio experimental consistió en realizar 30 ejecuciones independientes de las metaheurísticas para resolver la función de Ackley con 5, 10 y 15 variables de decisión. En la Tabla 8.1 se muestran los parámetros utilizados.

Parámetros	PE	EE	AG
Número de generaciones	200	200	200
Tamaño de la población padre	100	100	100
Método de selección de padres	×	aleatorio	*Selección por torneo*
Operador de mutación en las soluciones	Ec. 5.4	Ec. 6.5	Ec. 7.4
Operador de mutación en las variables de desplazamiento	Ec. 5.3	Ec. 6.4	×
Operador de cruza en las soluciones	×	*Cruza intermedia*	*Cruza intermedia*
Operador de cruza en variables de desplazamiento	×	*Cruza discreta*	×
Esquema de selección de sobrevivientes	*Selección más*	*Selección más*	*Selección más*

Tabla 8.1: Parámetros empleados en PE, EE y AG.

La metaheurística con los peores resultados es PE, una de las posibles razones es que no tiene un operador de cruza que combine información de diferentes individuos de

[1]El código para los problemas combinatorios lo pueden consultar en https://github.com/JAOP1/pyristic/blob/main/examples/benchmarkCombinatorial.ipynb

Algoritmo	Mejor solución	Peor solución	Solución promedio	Desviación estándar
Programación evolutiva	0.01	17.07	8.34	4.58
Estrategias evolutivas	0.00	0.00	0.00	0.00
Algoritmos genéticos	0.00	0.00	0.00	0.00

Tabla 8.2: Resultados obtenidos con 5 variables de decisión en la función de Ackley.

Algoritmo	Mejor solución	Peor solución	Solución promedio	Desviación estándar
Programación evolutiva	10.23	18.39	14.41	2.39
Estrategias evolutivas	0.00	1.64	0.13	0.40
Algoritmos genéticos	0.00	0.00	0.00	0.00

Tabla 8.3: Resultados obtenidos con 10 variables de decisión en la función de Ackley.

Algoritmo	Mejor solución	Peor solución	Solución promedio	Desviación estándar
Programación evolutiva	11.64	18.67	16.54	1.86
Estrategias evolutivas	0.02	3.22	1.32	1.07
Algoritmos genéticos	0.00	0.00	0.00	0.00

Tabla 8.4: Resultados obtenidos con 15 variables de decisión en la función de Ackley.

la población. Por otro lado, su operador de mutación mueve la solución actual dentro de un vecindario el cual depende de un parámetro α. Dado que la función de Ackley tiene muchos óptimos locales es posible que PE no logré salir de esas regiones. Una alternativa para tratar de mejorar los resultados obtenidos con PE es utilizar técnicas de ajuste de parámetros. Quizás un valor adecuado de α ayudé a la metaheurística a escapar de los óptimos locales. En el caso de las EE, vemos que para 5 variables de decisión logra encontrar la solución óptima en todos los casos. Sin embargo, para 10 y 15 variables empieza a tener dificultades y en algunas corridas no logra obtener el mínimo global. Recordemos que una diferencia entre EE y PE es que EE utiliza un operador de cruza, lo cual sugiere que este operador puede tener un impacto en los resultados. Sin embargo, no es su operador principal y la forma en que se eligen a los padres es completamente aleatoria. Por otro lado, los AG logran encontrar la solución global en las tres instancias. Por lo que podemos decir que los operadores de selección de padres, cruza y mutación permiten explorar y explotar adecuadamente el espacio de búsqueda. Es importante mencionar que los AG tienen una amplia variedad de operadores que se pueden utilizar y por lo tanto su diseño e implementación puede ser más compleja.

8.2. El problema de la mochila binario

Para este problema utilizamos las metaheurísticas de búsqueda tabú, recocido simulado y algoritmos genéticos. Nuestro estudio experimental consistió en resolver 3 instancias aleatorias del problema con 50, 100 y 150 objetos, donde, cada objeto está asociado con un beneficio $p_i \in [10, 100]$ y un peso $w_i \in [15, 80]$. Los arreglos P y W fueron

generados con una distribución uniforme.
En el caso de BT la solución inicial se generó como se indica en el apartado 4.2.1, el vecindario y la lista tabú como se explicó en el apartado 3.2.1. RS utiliza la solución inicial, el vecindario y la función de temperatura que se explicaron en el apartado 4.2.1. Finalmente, AG utiliza una selección por torneo para seleccionar a los padres y una selección más para seleccionar a los sobrevivientes, los operadores genéticos usados son cruza de un punto y mutación para representación binaria. La tabla 8.5 muestran los parámetros utilizados en la tres metaheurísticas.

Parámetros	BT	RS	AG
Evaluaciones de la función objetivo	42075	42075	42075
Tiempo tabú	$\frac{n}{2}$	×	×
Temperatura inicial	×	1000	×
Temperatura final	×	$6.86e^{-50}$	×
Tamaño de la población	×	×	19
Probabilidad de cruza	×	×	0.80
Probabilidad de mutación	×	×	0.60

Tabla 8.5: Parámetros de los algoritmos para el problema de la mochila.

Algoritmo	Mejor solución	Peor solución	Solución promedio	Desviación estándar
Búsqueda tabú	1542	1542	1542	0
Recocido simulado	1555	1555	1555	0
Algoritmos genéticos	1555	1531	1547.10	6.45

Tabla 8.6: Resultados obtenidos con 50 objetos en el problema de la mochila.

Algoritmo	Mejor solución	Peor solución	Solución promedio	Desviación estándar
Búsqueda tabú	2401	2401	2401	0
Recocido simulado	2417	2417	2417	0
Algoritmos genéticos	2363	2255	2307.08	30.47

Tabla 8.7: Resultados obtenidos con 100 objetos en el problema de la mochila.

Algoritmo	Mejor solución	Peor solución	Solución promedio	Desviación estándar
Búsqueda tabú	3768	3768	3768	0
Recocido simulado	3777	3770	3774.90	2.37
Algoritmos genéticos	3540	3188	3412.96	76.01

Tabla 8.8: Resultados obtenidos con 150 objetos en el problema de la mochila.

Las tablas 8.6, 8.7 y 8.8 muestran los resultados obtenidos al realizar 30 ejecuciones independientes de cada metaheurística y se puede observar que los AG obtienen los peores resultados en las tres instancias. Sin embargo, los resultados obtenidos por BT son producto de la técnica para generar una solución inicial, es decir, el algoritmo BT se estanca en un óptimo local sin lograr ninguna mejora, esto es por su selección voraz dentro del vecindario generado. Mientras, RS obtiene los mejores resultados, lo cual indica que su mecanismo permite que cualquier solución del vecindario pueda pasar a la siguiente iteración en las primeras etapas de la búsqueda ayuda a escapar de los óptimos locales. Con respecto a los AG son mucho más flexibles que RS o BT, en el sentido de que se pueden aplicar a una amplia variedad de problemas sin tener que realizar muchos cambios en la implementación. Por otro lado, BT y RS requieren del diseño de los mecanismos para generar las soluciones vecinas y en el caso de BT la forma en la que se utilizará la lista tabú u otras memorias que se quieran incorporar. Es importante mencionar que las tres metaheurísticas pueden ser mejoradas haciendo un análisis de sus parámeros o modificando su diseño. En esta tesis se construyó un diseño simple para ejemplificar su uso y estudiar sus resultados iniciales.

8.3. El problema del agente viajero

En este problema utilizaremos nuevamente búsqueda tabú, recocido simulado y algoritmos genéticos. Las instancias de prueba se crearon para 10, 15 y 20 ciudades, donde, los costos entre ciudades son números enteros generados aleatoriamente en el intervalo $[10, 100]$ utilizando una distribución uniforme. Los parámetros utilizados por las metaheurísticas se muestran en la tabla 8.9.
En el caso de BT se creó la solución inicial, el vecindario y la lista tabú como se explicó en el apartado 3.2.2. RS crea la solución inicial como en el apartado 3.2.2, el vecindario y la función de temperatura que se explicaron en el apartado 4.2.2. Finalmente, AG utiliza una selección por torneo para seleccionar a los padres y una selección más para seleccionar a los sobrevivientes, los operadores genéticos usados son cruza de permutaciones ordenadas y mutación para permutaciones: por desplazamiento. La tabla 8.9 muestran los parámetros utilizados en la tres metaheurísticas.

8.3.1. Instancia I (10 ciudades)

Como podemos observar, nuevamente BT tiene mayores dificultades. Como mencionamos en la sección anterior esto puede deberse a su mecanismo voraz de selección dentro del vecindario. Una forma de intentar mejorar su diseño es definir mecanismos más sofisticados para el uso de la lista tabú e incluso definir otro tipo de memorias que permitan extraer más conocimiento a lo largo de la búsqueda [3]. Por ejemplo, almacenar las características que comparten las mejores soluciones encontradas y ponerlas cómo tabú durante varias iteraciones. Por otro lado vemos que RS sigue siendo competitivo con respecto a AG y a un costo computacional mucho menor.

Parámetros	BT	RS	AG
Evaluaciones de la función objetivo	42075	42075	42075
Tiempo tabú	$\frac{n}{2}$	×	×
Temperatura inicial	×	1000	×
Temperatura final	×	$6.86e^{-50}$	×
Tamaño de la población	×	×	19
Probabilidad de cruza	×	×	0.70
Probabilidad de mutación	×	×	0.70

Tabla 8.9: Parámetros de los algoritmos para el problema de la mochila.

Algoritmo	Mejor solución	Peor solución	Solución promedio	Desviación estándar
Búsqueda tabú	237	271	256.26	16.84
Recocido simulado	221	259	230.46	10.24
Algoritmos genéticos	221	253	225.46	6.89

Tabla 8.10: Resultados obtenidos con 10 ciudades en el problema TSP.

Algoritmo	Mejor solución	Peor solución	Solución promedio	Desviación estándar
Búsqueda tabú	332	332	332	0
Recocido simulado	319	332	331.10	3.12
Algoritmos genéticos	300	354	320.30	13.35

Tabla 8.11: Resultados obtenidos con 15 ciudades en el problema TSP.

Algoritmo	Mejor solución	Peor solución	Solución promedio	Desviación estándar
Búsqueda tabú	408	449	418.73	14.48
Recocido simulado	385	472	447.40	26.37
Algoritmos genéticos	377	475	414.00	27.50

Tabla 8.12: Resultados obtenidos con 20 ciudades en el problema TSP.

8.4. Conclusiones

En este trabajo de tesis se estudiaron 5 metaheurísticas y se aplicaron a 4 problemas de optimización continuos y combinatorios. Los problemas de optimización continua se abordaron con las metaheurísticas de cómputo evolutivo (Programación Evolutiva, Estrategias Evolutivas y Algoritmos Genéticos). Esto debido a que su diseño contempla este tipo de problemas y por lo tanto su aplicación era directa. En el caso de los problemas de optimización combinatoria se utilizaron las metaheurísticas de Búsqueda Tabú, Recocido Simulado y Algoritmos Genéticos. Es importante mencionar que Búsqueda Tabú y Recocido Simulado no pueden aplicarse directamente ya que el diseño de sus componentes depende del problema que se está resolviendo. A diferencia

de los Algoritmos Genéticos que puede aplicarse fácilmente a una amplia variedad de problemas. Con respecto a los resultados obtenidos, podemos decir que los AG son una buena opción para buscar buenas soluciones en problemas de optimización difíciles. Sin embargo, debemos tener en cuenta que se pueden realizar diseños más elaborados de todas las metaheurísticas con el objetivo de mejorar sus resultados.

Apéndice A

Ejemplos de uso de la librería pyristic

A.1. Búsqueda Tabú

La librería **Pyristic** incluye una clase llamada `TabuSearch` que facilita la implementación de una metaheurística basada en Búsqueda Tabú para resolver problemas de minimización. Para poder utilizar esta clase es necesario:

1. Definir:
    - La función objetivo f.
    - La lista de restricciones.
    - Estructura de datos (opcional).
2. Crear una clase que herede de `TabuSearch`.
3. Sobreescribir las siguientes funciones de la clase `TabuSearch`:
    - get_neighbors (requerido)
    - encode_change (requerido)

A continuación se muestran las librerías y elementos que se deben importar. Posteriormente, se resolverán dos problemas de optimización combinatoria usando la clase `TabuSearch`.

```
In [1]: import sys
        import os
        from pyristic.heuristic.Tabu_search import TabuSearch
        from pyristic.utils.helpers import *
        from pprint import pprint
        import numpy as np
        import copy
```

A.1.1. Clase `TabuSearch`

Variables

- ***logger.*** Diccionario con información relacionada a la búsqueda con las siguientes llaves:
  - `best_individual`. Mejor individuo encontrado.
  - `best_f`. El valor obtenido de la función objetivo de `individual`.
  - `current_iter`. Iteración actual de la búsqueda.
  - `total_iter`. Número total de iteraciones.
- ***TL.*** Estructura de datos auxiliar que mantendrá memoria de las soluciones encontradas durante el tiempo especificado en `optimize`, por defecto utiliza `TabuList`.
- ***f.*** Función objetivo.
- ***Constraints.*** Lista de restricciones del problema. Las restricciones deben ser funciones que retornan True o False, indicando si cumple dicha restricción.

Métodos

- ***__init__.*** Constructor de la clase.
  Argumentos:
  - `function`. Función objetivo.
  - `constraints`. Lista con las restricciones del problema.
  - `TabuStruct`. Estructura de datos que almacena información de variaciones que mejoran la solución.

  Valor de retorno:
  - Ninguno.
- ***optimize.*** método principal, realiza la ejecución empleando la metaheurística llamada `TabuSearch`.

  Argumentos:
  - `Init`. Solución inicial, se admite un arreglo de *numpy* o una función que retorne un arreglo de *numpy*.
  - `iterations`. Número de iteraciones.
  - `memory_time`. Tiempo que permanecerá una solución en nuestra estructura llamada `TabuList`.
  - `**kwargs`. Parámetros externos a la búsqueda.

Valor de retorno:

- Ninguno

- ***get_neighbors.*** Función que genera el vecindario de soluciones de la solución x.
  Argumentos:
  - `x`. Arreglo de *numpy* representando a la solución actual.
  - `**kwargs`. Parámetros externos a la búsqueda.

  Valor de retorno:
  - Arreglo bidimensional de *numpy* representando a todas las soluciones generadas desde la solución x.

- ***encode_change.*** Revisa nuestra solución actual x y la solución generada para indicar en dónde sucedió la pequeña variación.
  Argumentos:
  - `neighbor`. Arreglo de *numpy* representando una variación de nuestra solución actual x.
  - `x`. Arreglo de *numpy* representando nuestra solución actual.
  - `**kwargs`. Parámetros externos a la búsqueda.

  Valor de retorno:
  - Lista con dos elementos, donde, la primera componente será la posición i donde sucedió la variación y la segunda componente es el elemento en la componente i de `neighbor`.

Clase `TabuList`

Clase auxiliar que almacenará las soluciones encontradas en la búsqueda, las soluciones permanecerá en la estructura de datos por un tiempo definido en la función `optimize` de `TabuSearch`.

Variables

- ***_TB.*** Lista de listas que representarán las posiciones que fueron modificadas con un contador de tiempo.
- ***timer.*** El tiempo que durará cada solución en la lista.

Métodos

- ***push.*** Introduce los cambios que proporcionaron una mejora en la búsqueda.
  Argumentos:
  - `x`. Arreglo con la siguiente información:
    - Primera componente: posición (índice) donde se encontró una mejora en la función objetivo.
    - Segunda componente: valor por el cual mejoró nuestra solución.
    - Tercera componente: iteración en la que se realizó la mejora.

  Valor de retorno:
  - Ninguno.

- ***find.*** Revisa si la nueva solución sea una de las modificaciones hechas en iteraciones previas almacenadas en `_TB`.
  Argumentos:
  - `x`. Arreglo de *numpy* que representa el cambio realizado en la solución actual de la búsqueda, es decir, recibe el arreglo que retorna la función `encode_change(neighbor,x)`.

  Valor de retorno:
  - Valor booleano que indica si la modificación en dicha solución ya se encontraba en nuestra lista tabú.

- ***reset.*** Borra toda la información almacenada en nuestro contenedor `_TB` y actualiza la variable `timer`. Argumentos:
  - `timer`. Número que representa el tiempo que durarán ahora las soluciones en nuestra lista tabú.

  Valor de retorno:
  - Ninguno.

- ***update.*** Realiza la actualización en el contendor `_TB` modificando el tiempo de cada uno de los individuos almacenados y elimina aquellos individuos que ya expiró su tiempo.
  Argumentos:
  - Ninguno.

  Valor de retorno:
  - Ninguno.

- ***pop_back.*** Elimina el último elemento del contenedor `_TB`.
  Argumentos:

- Ninguno.

Valor de retorno:

- Ninguno.

- ***get_back.*** Regresa el último elemento del contenedor `_TB`.
  Argumentos:
  - Ninguno.

  Valor de retorno:
  - Elemento del contendor `_TB`.

A.1.2. Problema de la mochila

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

Consideremos la siguiente entrada:

- $n = 5$
- $p = \{5, 14, 7, 2, 23\}$
- $w = \{2, 3, 7, 5, 10\}$
- $c = 15$

Donde la mejor solución es: $x = [1, 1, 0, 0, 1]$, $f(x) = 42$ y $g_1(x) = 15$

Función objetivo

Dado que la clase `TabuSearch` considera problemas de minimización, es necesario convertir el problema de la mochila a un problema de minimización. Para esto se multiplica el valor de la función objetivo por -1.

```
In [3]:
        def f(x : np.ndarray) -> float:
            p = np.array([5,14,7,2,23])
            return -1*np.dot(x,p)
```

Restricciones

Las restricciones se definen en funciones diferentes y se agregan a una lista.

```
In [4]:
        def g1(x : np.ndarray) -> bool:
            w = [2,3,7,5,10]
            return np.dot(x,w) <= 15

        constraints_list= [g1]
```

En el problema de la mochila unicamente queremos revisar que no se exceda el peso.

Uso de `TabuSearch`

Para poder hacer uso de la metaheurística de búsqueda tabú implementada en la librería **Pyristic**, es necesario crear una clase que herede de la clase `TabuSearch`.

```
In [5]: class Knapsack_solver(TabuSearch):

            def __init__(self, f_ : function_type , constraints_: list):
                super().__init__(f_,constraints_)


            def get_neighbors(self, x : np.ndarray,**kwargs) -> list:
                neighbors_list = []

                for i in range(len(x)):
                    x[i] ^= 1 #1
                    neighbors_list+=[copy.deepcopy(x)]
                    x[i] ^= 1

                return neighbors_list

            def encode_change(self, neighbor : (list,np.ndarray),\
            x : (list,np.ndarray),**kwargs) -> list: #2

                x_ = [None,None]

                for i in range(len(x)):
                    if x[i] != neighbor[i]:
                        return [i,neighbor[i]]

                return x_
```

La nueva clase es llamada *Knapsack_solver*, donde, se han sobrescrito las funciones `get_neighbors` y `encode_change`. Si no implementamos las funciones mencionadas el algoritmo no va a funcionar.

Ejecución de la metaheurística

Una vez definida la clase *Knapsack_solver*, se crea un objeto de tipo *Knapsack_solver* indicando en los parámetros la función objetivo y las restricciones del problema. En este caso llamamos *Knapsack* al objeto creado.

```
In [6]: Knapsack = Knapsack_solver(f, [g1])
```

Finalmente, se llama a la función `optimize`. Esta función recibe tres parámetros:

- Solución inicial o función generadora de soluciones iniciales.
- El número de iteraciones.
- El tiempo donde evitaremos hacer un cambio en cierta posición (tiempo tabú).

Para este ejemplo usamos una mochila vacía ($x_0 = [0,0,0,0,0]$), 30 iteraciones y un tiempo tabú igual a 3.

```
In [7]: init_backpack_solution = np.zeros(5,dtype=int)
        '''Parameters:
            Initial solution
            Number of iterations
            Tabu time
        '''
        Knapsack.optimize(init_backpack_solution,30,3)
        print(Knapsack)

Tabu search:
 f(X) = -42
 X = [1 1 0 0 1]
```

A continuación resolveremos el mismo problema para una instancia más grande.
Tenemos que definir nuevamente la función objetivo y la restricción para emplearlo para cualquier instancia del problema.
Definiremos las siguientes variables como variables globales:

- n es un número que indicará el tamaño de nuestra instancia.
- p es un arreglo que se refiere al beneficio que proporciona cada uno de los objetos.

- w es un arreglo con el peso de cada uno de los objetos.
- c es el peso máximo que puede tener nuestra mochila.

```
In [8]: n = 50
        p = [60, 52, 90, 57, 45, 64, 60, 45, 63, 94, 44, 90, 66, 64,\
             32, 39, 91, 40, 73, 61, 82, 94, 39, 68, 94, 98, 80, 79, 73,\
             99, 49, 56, 69, 49, 82, 99, 65, 34, 31, 85, 67, 62, 56, 38,\
             54, 81, 98, 63, 48, 83]
        w = [38, 20, 21, 21, 37, 28, 32, 30, 33, 35, 29, 32, 35, 24, 28,\
             29, 22, 34, 31, 36, 36, 28, 38, 25, 38, 37, 20, 23, 39, 31,\
             27, 20, 38, 38, 36, 28, 39, 22, 23, 22, 21, 24, 23, 33, 31,\
             30, 32, 30, 22, 37]
        c = 870

In [9]:
        def f(x : np.ndarray) -> float:
            global p
            return -1* np.dot(x,p)


        def g1(x : np.ndarray) -> bool:
            global w,c
            result = np.dot(x,w)
            g1.__doc__="{} <= {}".format(result,c)
            return result <= c

        constraints_list= [g1]
```

Solución inicial

En el ejemplo anterior, la solución inicial fue una mochila vacía. Ahora crearemos una mochila que introduce objetos de manera aleatoria, mientras no se exceda el peso de la mochila.

```
In [10]: def getInitialSolution(NumObjects=5):
             global n,p,w,c
             #Empty backpack
             x = [0 for i in range(n)]
             weight_x = 0

             #Random order to insert objects.
             objects = list(range(n))
             np.random.shuffle(objects)
```

```
        for o in  objects[:NumObjects]:
            #Check the constraint about capacity.
            if weight_x + w[o] <= c:
                x[o] = 1
                weight_x += w[o]

        return np.array(x)
```

Definiremos nuestro objeto del tipo *Knapsack_solver* y llamaremos el método `optimize` con los siguientes parámetros:

- La función que crea la solución inicial.
- 100 iteraciones.
- El tiempo tabú será $\frac{n}{2}$.

```
In [11]: Knapsack_2 = Knapsack_solver(f, [g1])
         Knapsack_2.optimize(getInitialSolution,100,n//2)
         print(Knapsack_2)
```

```
Tabu search:
 f(X) = -2276
 X = [1 0 1 1 .... 0 0 1]
 Constraints:
 870 <= 870
```

Para revisar el comportamiento de la metaheurística en determinado problema, la librería **Pyristic** cuenta con una función llamada `get_stats`. Esta función se encuentra en **utils.helpers** y recibe como parámetros:

- El objeto creado para ejecutar la metaheurística.
- El número de veces que se quiere ejecutar la metaheurística.
- Los argumentos que recibe la función `optimize` (debe ser una tupla).

La función `get_stats` retorna un diccionario con algunas estadísticas de las ejecuciones.

```
In [12]: args = (getInitialSolution,500,n//2)
         statistics = get_stats(Knapsack_2, 21, args)

In [13]: pprint(statistics)
```

```
{'Best solution': {'f': -2309,
                   'x': array([0, 0, 1, 0, ... , 0, 0, 1])},
 'Mean': -2244.5238095238096,
 'Standard deviation': 38.099642602522,
 'Worst solution': {'f': -2180,
                    'x': array([0, 0, 1, 0, ... , 0, 0, 1])}}
```

A.1.3. Problema del agente viajero

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

Donde:

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

```
In [14]: import random
```

```
In [15]: num_cities = 10
         iterations = 100
         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],\
         ]
```

```
In [16]:
         def f_salesman(x : np.ndarray) -> float:
             global dist_matrix
             total_dist = 0
             for i in range(1,len(x)):
                 u,v = x[i], x[i-1]
```

```
            total_dist+= dist_matrix[u][v]
        total_dist += dist_matrix[x[-1]][0]
        return total_dist
```

```
In [17]:
    def g_salesman(x : np.ndarray) -> bool:
        """
        Xi in {1,2, ... , N}
        """
        size = len(x)
        size_ = len(np.unique(x))
        return size == size_
```

En este ejemplo mostraremos la forma de definir nuestra lista tabú para el problema del agente viajero para emplearla en nuestra búsqueda `TabuSearch`. Es necesario que nuestra lista tabú contenga los siguientes métodos:

- `reset`
- `update`
- `push`
- `find`

```
In [18]: class Tabu_Salesman_list:
            def __init__(self,timer):
                self.__TB = {}
                self.timer = timer

            def reset(self,timer) -> None:
                self.__TB = {}
                self.timer = timer

            def update(self) -> None:
                to_pop = []
                for key in self.__TB:
                    if self.__TB[key]-1 == 0:
                        to_pop.append(key)
                    else:
                        self.__TB[key]-=1
                for key in to_pop:
                    self.__TB.pop(key)
```

```
            #x has [p,v,step], we are only interested in v (value)
            def push(self, x : list ) -> None:
                self.__TB[x[1]] = self.timer


            def find(self, x : list) -> bool:
                return x[1] in self.__TB
```

```
In [19]: class TravellingSalesman_solver(TabuSearch):

            def __init__(self, f_ : function_type ,\
                         constraints_: list, TabuStorage):
                super().__init__(f_,constraints_,TabuStorage)


            def get_neighbors(self, x : np.ndarray,**kwargs) -> list:

                neighbors_list = []

                ind = random.randint(1,len(x)-1)
                while  self.TL.find([-1,x[ind]]):
                    ind = random.randint(1,len(x)-1)
                v = x[ind]
                x_tmp = list(x[v != x])
                for i in range(1, len(x)):
                    if ind == i:
                        continue
                    neighbors_list += [ x_tmp[:i] + [v] + x_tmp[i:]]

                return neighbors_list



            def encode_change(self, neighbor : (list,np.ndarray),\
                              x : (list,np.ndarray),**kwargs) -> list: #2

                x_p ={x[i] : i for i in range(len(x))}
                n_p = {neighbor[i]: i for i in range(len(x))}
                ind = -1
                max_dist = -1
                value = -1
                for i in range(1, len(x)):
                    v = x[i]
```

```
                    dist = abs(x_p[v] - n_p[v])
                    if dist > max_dist:
                        ind = i
                        max_dist = dist
                        value = v

                return [ind , value]
```

Solución inicial

En este caso, creamos la solución inicial utilizando una estrategia voraz.

```
In [20]: def getInitialSolutionTS(distance_matrix, total_cities):
             Solution = [0]
             remaining_cities  = list(range(1,total_cities))

             while len(remaining_cities) != 0:
                 from_ =Solution[-1]
                 to_ = remaining_cities[0]
                 dist = distance_matrix[from_][to_]

                 for i in range(1, len(remaining_cities)):
                     distance = distance_matrix[from_][remaining_cities[i]]
                     if distance < dist:
                         to_ = remaining_cities[i]
                         dist = distance
                 Solution.append(to_)
                 ind = remaining_cities.index(to_)
                 remaining_cities.pop(ind)
             return Solution
```

```
In [21]: TravellingSalesman = TravellingSalesman_solver(f_salesman,[g_salesman],\
                                Tabu_Salesman_list(num_cities//2))
         init_path = np.array(getInitialSolutionTS(dist_matrix,num_cities))
         print("Initialize search with this initial point {} \n f(x) = {}".format(\
                             init_path, f_salesman(init_path)))
```

```
Initialize search with this initial point [0 5 3 8 9 6 2 7 1 4]
 f(x) = 271
```

```
In [22]: TravellingSalesman.optimize(init_path, iterations, num_cities//2)
         print(TravellingSalesman)
```

```
Tabu search:
 f(X) = 248
 X = [0 5 3 8 9 6 2 7 4 1]
 Constraints:

    Xi in {1,2, ... , N}
```

```
In [23]: args = (init_path, iterations, num_cities//2)
         statistics = get_stats(TravellingSalesman, 30, args)
```

```
In [24]: pprint(statistics)
```

```
{'Best solution': {'f': 248, 'x': array([0, 5, 3, 8, 9, 6, 2, 7, 4, 1])},
 'Mean': 248.0,
 'Standard deviation': 0.0,
 'Worst solution': {'f': 248, 'x': array([0, 5, 3, 8, 9, 6, 2, 7, 4, 1])}}
```

A.2. Recocido Simulado

La librería **Pyristic** incluye una clase llamada *SimulatedAnnealing* que facilita la implementación de la metaheurística basada en *Recocido Simulado*. Para poder utilizar esta clase, se debe realizar lo siguiente:

1. Definir:
   - La función objetivo f.
   - La lista de restricciones.
2. Crear una clase que herede de *SimulatedAnnealing*.
3. Sobreescribir las siguientes funciones de la clase SimulatedAnnealing:
   - get_neighbor (requerido)
   - update_temperature (opcional)

A continuación se muestran los elementos que se deben importar. Posteriormente, se resolverán dos problemas de optimización combinatoria usando la clase *SimulatedAnnealing*.

```
In [1]: import sys
        import os
        from pyristic.heuristic.SimulatedAnnealing_search import SimulatedAnnealing
        from pyristic.utils.helpers import *
        from pprint import pprint
        import numpy as np
        import copy
```

A.2.1. Clase `SimulatedAnnealing`

Variables

- ***logger.*** Diccionario con información relacionada a la búsqueda con las siguientes llaves:
  - `best_individual`. Mejor individuo encontrado.
  - `best_f`. El valor obtenido de la función objetivo de `individual`.
  - `temperature`. Temperatura inicial que se actualizará cada iteración.
- ***f.*** Función objetivo.
- ***Constraints.*** Lista de restricciones del problema. Las restricciones deben ser funciones que retornan True o False, indicando si cumple dicha restricción.

Métodos

- ***__init__.*** Inicializa la clase.
  Argumentos:
  - `function`. Función objetivo.
  - `constraints`. Lista con las restricciones del problema.

  Valor de retorno:
  - Ninguno.

- ***optimize.*** método principal, realiza la ejecución empleando la metaheurística llamada `SimulatedAnnealing`.
  Argumentos:
  - `Init`. Solución inicial, se admite un arreglo de *numpy* o una función que retorne un arreglo de *numpy*.
  - `IniTemperature`. Valor de punto flotante que indica con que temperatura inicia la búsqueda. * `eps`. Valor de punto flotante que indica con que temperatura termina la búsqueda.
  - `**kwargs`. Parámetros externos a la búsqueda.

  Valor de retorno:
  - Ninguno.

- ***get_neighbor.*** Genera una solución realizando una variación aleatoria en la solución actual.
  Argumentos:
  - `x`. Arreglo de *numpy* representando a la solución actual.
  - `**kwargs` Parámetros externos a la búsqueda.

  Valor de retorno:
  - Arreglo de *numpy* representando la solución generada.

- ***update_temperature.*** Función que decrementa la temperatura.
  Argumentos:
  - `**kwargs` Parámetros externos a la búsqueda.

  Valor de retorno:
  - La nueva temperatura.

A.2.2. Problema del agente viajero

$$\begin{array}{rl} \text{minimizar:} & f(x) = d(x_n, x_1) + \sum_{i=1}^{n-1} d(x_i, x_{i+1}) \\ \text{tal que:} & x_i \in \{1, 2, \cdots, n\} \end{array} \tag{A.3}$$

Donde:

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

A continuación vamos a definir una instancia de este problema utilizando 10 ciudades.

```
In [3]: import random
        import math

        num_cities = 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],\
        ]
```

Función objetivo

```
In [4]: @checkargs
        def f_salesman(x : (list,np.ndarray)) -> float:
            global dist_matrix
            total_dist = dist_matrix[x[-1]][0]
            for i in range(1,len(x)):
                u,v = x[i], x[i-1]
                total_dist+= dist_matrix[u][v]

            return float(total_dist)
```

Restricciones

Las restricciones se definen como una lista de funciones que retornan valores *booleanos*. Estos valores permitirán verificar si una solución es factible o no.
En el caso del problema del agente viajero queremos comprobar que estamos visitando todas las ciudades exactamente una vez.

```
In [5]: @checkargs
        def g_salesman(x : np.ndarray) -> bool:

            size = len(x)
            size_ = len(np.unique(x))
            return size == size_

        constraints_list= [g_salesman]
```

Solución inicial

La estrategia utilizada para generar la solución inicial es la siguiente:

1. Introducir la ciudad 1 en la primera posición de nuestra permutación y crear un arreglo con las ciudades restantes.
2. Seleccionar la ciudad más cercana desde nuestra ubicación actual.
3. Retirar del arreglo la ciudad seleccionada y asignarla como nuestra ubicación actual. Se repite el punto 2 hasta que no haya más ciudades en el arreglo.

```
In [6]: def getInitialSolutionTS(distance_matrix, total_cities) -> np.ndarray:
            Solution = [0]
            remaining_cities  = list(range(1,total_cities))

            while len(remaining_cities) != 0:
                from_ = Solution[-1]
                to_ = remaining_cities[0]
                dist = distance_matrix[from_][to_]

                for i in range(1, len(remaining_cities)):
                    distance = distance_matrix[from_][remaining_cities[i]]
                    if distance < dist:
                        to_ = remaining_cities[i]
                        dist = distance
                Solution.append(to_)
                ind = remaining_cities.index(to_)
                remaining_cities.pop(ind)
```

```
            return np.array(Solution)
```

```
In [7]: Path = getInitialSolutionTS(dist_matrix,num_cities)
        print(Path)
```

```
[0 5 3 8 9 6 2 7 1 4]
```

Declaración de `SimulatedAnnealing`

Para implementar una metaheurística basada en recocido simulado, utilizando la librería **Pyristic**, es necesario crear una clase que herede de la clase `SimulatedAnnealing`. En este ejemplo, la nueva clase es llamada *TravellingSalesman_solver*.
La nueva clase debe sobreescribir la función `get_neighbor`, de lo contrario el algoritmo no va a funcionar.

```
In [8]: class TravellingSalesman_solver(SimulatedAnnealing):

            @checkargs
            def __init__(self, f_ : function_type , constraints_: list):
                super().__init__(f_,constraints_)

            @checkargs
            def get_neighbor(self, x : np.ndarray) -> np.ndarray:

                x_ = x.copy()
                N = len(x_)
                index1 = random.randint(1, N-1)
                index2 = random.randint(1, N-1)

                while index2 == index1:
                    index2 = random.randint(1, N-1)

                v = x[index1]
                x_ = list(x_[v != x_])
                x_ = x_[:index2] + [v] + x_[index2:]
                return np.array(x_)
```

La función `get_neigbor` debe regresar una solución *vecina* de la solución actual, es decir, una variación de la solución actual. Para nuestro ejemplo, una solución vecina de la solución x, la vamos a definir como sigue:
Se seleccionan dos indices distintos de manera aleatoria llamados i y j, donde, tomaremos el elemento x_i y desplazaremos las otras posiciones de la solución x de modo que x_i se encuentre en la posición j y esta nueva solución será retornada.

Ejecución de la metaheurística

Una vez definida la clase *TravellingSalesman_solver*, se crea una instancia indicando en los parámetros la función objetivo y las restricciones del problema. En este caso llamamos *TravellingSalesman* a la instancia creada.

```
In [9]: TravellingSalesman =\
        TravellingSalesman_solver( f_salesman, constraints_list)
```

Finalmente, se llama la función `optimize` (esta función es la misma para todas las clases en la librería). La función `optimize` recibe tres parámetros:

- Solución inicial o función generadora de soluciones iniciales.
- La temperatura inicial.
- La temperatura final.

Vamos a utilizar los siguientes parámetros:

- Emplearemos la solución obtenida por la función *getInitialSolutionTS*.
- 1000 de temperatura inicial.
- 0.1 de temperatura final.

```
In [10]: TravellingSalesman.optimize(Path, 1000.0 , 0.1)
         print(TravellingSalesman)

Simulated Annealing:
 f(X) = 271.0
 X = [0 5 3 8 9 6 2 7 1 4]
```

Para revisar el comportamiento de la metaheurística en determinado problema, la librería **Pyristic** cuenta con una función llamada `get_stats`. Esta función se encuentra en **utils.helpers** y recibe como parámetros:

- El objeto creado para ejecutar la metaheurística.
- El número de veces que se quiere ejecutar la metaheurística.
- Los argumentos que recibe la función `optimize` deben estar contenidos en una tupla.

La función **get_stats** retorna un diccionario con algunas estadísticas de las ejecuciones.

```
In [11]: #Ejecutamos get_stats 30 veces.
         args = (Path, 1000.0, 0.01)
         statistics = get_stats(TravellingSalesman, 30, args)
```

```
In [12]: pprint(statistics)
```

```
{'Best solution': {'f': 248.0, 'x': array([0, 5, 3, 8, 9, 6, 2, 7, 4, 1])},
 'Mean': 258.1333333333333,
 'Standard deviation': 11.035498277025022,
 'Worst solution': {'f': 271.0, 'x': array([0, 5, 3, 8, 9, 6, 2, 7, 1, 4])}}
```

A.2.3. Problema de la mochila

$$\begin{array}{rll} \text{maximizar:} & f(\boldsymbol{x}) = \sum_{i=1}^{n} p_i \cdot x_i & \\ \text{donde:} & g_1(\boldsymbol{x}) = \sum_{i=1}^{n} w_i \cdot x_i \leq c & \\ & x_i \in \{0,1\} & i \in \{1,\ldots,n\} \end{array} \tag{A.4}$$

Para este problema vamos a crear una instancia de 1000 objetos, donde, cada objeto estará definido de la siguiente manera:

- $p_i \in [50, 100]$
- $w_i \in [5, 100]$
- $C = 9786$

```
In [13]: n = 1000
         p = np.random.randint(50,101,n)
         w = np.random.randint(5,101,n)
         c = 9786
```

A continuación mostraremos dos algorimos basados en recocido simulado. El primero de ellos es un diseño sencillo y el segundo es un diseño más elaborado que obtiene mejores resultados.

Función objetivo

Dado que la clase `SimulatedAnnealing` considera problemas de minimización, es necesario convertir el problema de la mochila a un problema de minimización. Para esto se multiplica el valor de la función objetivo por -1.

```
In [14]:  def f(x : np.ndarray) -> float:
              global p
              return -1.0* np.dot(x,p)
```

Restricciones

La restricción del problema de la mochila es seleccionar un número de objetos sin exceder la capacidad de la mochila.

```
In [15]: def g1(x : np.ndarray) -> bool:
             global w,c
             result = np.dot(x,w)
             g1.__doc__="{} <= {}".format(result,c)
             return result <= c

         constraints_list= [g1]
```

Solución inicial

Nuestra solución inicial es creada introduciendo objetos de manera aleatoria, mientras no se exceda el peso de la mochila.

```
In [16]: def getInitialSolution(NumObjects=15):
             global n,p,w,c
             #Empty backpack
             x = [0 for i in range(n)]
             weight_x = 0

             #Random order to insert objects.
             objects = list(range(n))
             np.random.shuffle(objects)

             for o in  objects[:NumObjects]:
                 #Check the constraint about capacity.
                 if weight_x + w[o] <= c:
                     x[o] = 1
                     weight_x += w[o]

             return np.array(x)
```

Ejecución de la metaheurística

```
In [17]: class Knapsack_solver(SimulatedAnnealing):


             def __init__(self, f_ : function_type , constraints_: list):
                 super().__init__(f_,constraints_)
```

```
    def get_neighbor(self, x : np.ndarray, **kwargs) -> np.ndarray:

        x_ = x.copy()
        N = len(x_)
        while(True):
            ind = random.randint(0, N-1)
            x_[ind] ^= 1

            if(self.is_valid(x_)):
                break

            x_[ind] ^= 1

        return np.array(x_)

    def update_temperature(self, **kwargs) -> float:
        return self.logger['temperature'] * 0.99
```

Parte clave de este algoritmo es la temperatura. La temperatura es una función que varía de acuerdo al tiempo t, donde al inicio de la búsqueda permite aceptar con mayor probabilidad soluciones peores.
La clase `SimulatedAnnealing` define esta función como `update_temperature` que está implementada por defecto. En este ejemplo vamos a definir nuestra función de cambio lineal $T(t+1) = \sigma T(t)$.
La función `get_neighbor` tomará la solución x (arreglo de 0's y 1's), donde el valor de retorno será la solución x con una posición aleatoria i que será reemplazada por el valor 0 si la posición x_i es 1, sino, será 1.

```
In [18]: Knapsack_ = Knapsack_solver(f, [g1])
         Knapsack_.optimize(getInitialSolution,1000.0,0.1)

In [19]: print(Knapsack_)

Simulated Annealing:
 f(X) = -15402.0
 X = [0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
      0 0 1 1 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0
        ...  0 0 1 0 0 0 0 0 0 1 0 0 0 0 0]
 Constraints:
 9785 <= 9786
```

Para realizar un estudio estadístico del comportamiento de la metaheurística, la librería Pyristic cuenta con una función llamada get_stats. Esta función se encuentra en **utils.helpers** y recibe como parámetros:

- Objeto que realiza la ejecución de la metaheurística.
- El número de veces que se quiere ejecutar la metaheurística.
- Tupla con los argumentos que recibe la función optimize.
- Argumentos adicionales a la búsqueda (opcional).

La función get_stats considera las solución regresada por la metaheurística en cada ejecución y retorna un diccionario con la mejor y peor solución encontrada, la media y desviación estándar del valor de la función objetivo.

```
In [20]: args = (getInitialSolution,1000.0,0.01)
         statistics = get_stats(Knapsack_, 30, args)

In [21]: pprint(statistics)

{'Best solution': {'f': -16780.0,
                   'x': array([1, 0, 0, 0, 0, ... , 0, 0, 0])},
 'Mean': -15396.4,
 'Standard deviation': 488.9018715447917,
 'Worst solution': {'f': -14012.0,
                    'x': array([0, 0, 0, 0, 0, ... , 0, 0, 0])}}
```

Se puede observar que los resultados obtenidos no son tan buenos como los obtenidos con la metaheurística basada en Búsqueda Tabú. A continuación se harán algunas modificaciones en el diseño, con el objetivo de mejorar los resultados.

Solución inicial

En el diseño anterior, no se consideró el beneficio de introducir un objeto determinado a la mochila según su valor y peso. Una forma de hacerlo es la siguiente:

1. Asignar a cada objeto su valor por unidad de peso: $\frac{p_i}{w_i}$.
2. Ordenar todos los objetos con respecto a este indicador.
3. Seleccionar los objetos que tenga mayor valor en este indicador sin exceder la capacidad.

```
In [22]: def Init_GISP():
             global n,p,w,c
             Arr_=[]
             for i in range(n):
                 Arr_.append((p[i]/w[i],i))
```

```
        Arr_.sort(reverse=True)
        return Arr_
    GISP_Arr = Init_GISP()

    def getInitialSolution2():
        global n,p,w,c, GISP_Arr
        X = [0 for i in range(n)]
        current_weight = 0
        for i in range(n):
            ind = GISP_Arr[i][1]
            if current_weight+ w[ind] <= c:
                current_weight+=w[ind]
                X[ind] = 1
        return np.array(X)
```

Para crear un nuevo vecino, seguiremos la misma estrategia que en el diseño anterior. Sin embargo, si la nueva solución no es válida vamos a realizar lo siguiente:

1. Fase de reparación.
2. Fase de mejoramiento.

Fase de reparación: En esta fase retiramos objetos que se encuentran en la mochila mientras la capacidad actual exceda el límite definido c. Los objetos son retirados de acuerdo a los valores más pequeños en el indicador definido ($\frac{p_i}{w_i}$).
Fase de mejoramiento: En esta fase se introducen objetos a la mochila siempre y cuando no excedan la capacidad. Estos objetos son introducidos priorizando aquellos con los valores más grandes en el indicador definido ($\frac{p_i}{w_i}$).

```
In [23]: class Knapsack_solver2(SimulatedAnnealing):

            def __init__(self, f_ : function_type , constraints_: list):
                super().__init__(f_,constraints_)


            def generate_neighbor(self, x: np.ndarray) -> np.ndarray:
                x_ = x.copy()
                N = len(x_)
                ind = random.randint(0, N-1)
                x_[ind] ^= 1

                return x_

            def repair_neighbor(self, x: np.ndarray) -> np.ndarray:
```

```
        global GISP_Arr,w,c,n

        #Get the total weight
        total_weight=0
        for i in range(n):
            if x[i]:
                total_weight += w[i]

        for i in range(n):
            ind = GISP_Arr[n - (1+i)][1] #Lowest

            if x[ind]:
                total_weight -= w[ind]
                x[ind] = 0

            if total_weight <= c:
                break

    def improve_neighbor(self, x: np.ndarray) -> np.ndarray:
        global GISP_Arr,w,c,n

        #Get the total weight
        total_weight=0
        for i in range(n):
            if x[i]:
                total_weight += w[i]

        for i in range(n):
            ind = GISP_Arr[i][1]
            if total_weight + w[ind] > c:
                continue

            if x[ind] == 0 :
                total_weight += w[ind]
                x[ind] = 1


    def get_neighbor(self, x : np.ndarray) -> np.ndarray:

        neighbor_ = self.generate_neighbor(x)

        if(self.is_valid(neighbor_)):
            return neighbor_
```

```
                #RI strategy
                self.repair_neighbor(neighbor_)
                self.improve_neighbor(neighbor_)

                return neighbor_
```

Ejecutamos nuestra metaheurística:

```
In [24]: f(getInitialSolution2())
```

```
Out[24]: -29887.0
```

```
In [25]: Knapsack_2 = Knapsack_solver2(f, [g1])
```

```
In [26]: Knapsack_2.optimize(getInitialSolution2,1000.0,0.1)
         print(Knapsack_2)
```

```
Simulated Annealing:
 f(X) = -29901.0
 X = [1 0 0 0 1 0 ..... 0 1 1 0 0 0 0 1 0 0 0 0]
 Constraints:
 9786 <= 9786
```

A.2.4. Resultados

Vamos a comparar las distintas combinaciones entre la forma de generar la solución inicial y la estrategia de generar un nuevo vecino.

1. Solución por Indicador y RI estrategia.
2. Solución por Indicador y estrategia ingenua.
3. Solución ingenua y RI estrategia.
4. Solución ingenua y estrategia ingenua.

Solución por Indicador y RI estrategia.

```
In [27]: args = (getInitialSolution2,1000.0,0.01)
         statistics = get_stats(Knapsack_2, 30, args)
```

```
In [28]: print("f(x*) = {} \nfunción objetivo promedio:
         {} \nfunción objetivo de la peor solución: {}".format(
         statistics['Best solution']['f'],statistics['Mean']
         ,statistics['Worst solution']['f']))
```

```
f(x*) = -29912.0
función objetivo promedio: -29894.8
función objetivo de la peor solución: -29887.0
```

Solución por Indicador y estrategia ingenua.

```
In [29]: args = (getInitialSolution2,1000.0,0.01)
         statistics = get_stats(Knapsack_, 30, args)

In [30]: print("f(x*) = {} \nfunción objetivo promedio:
         {} \nfunción objetivo de la peor solución: {}".format(
         statistics['Best solution']['f'],statistics['Mean']
         ,statistics['Worst solution']['f']))
```

```
f(x*) = -29887.0
función objetivo promedio: -29887.0
función objetivo de la peor solución: -29887.0
```

Solución ingenua y RI estrategia.

```
In [31]: args = (getInitialSolution,1000.0,0.01)
         statistics = get_stats(Knapsack_2, 30, args)

In [32]: print("f(x*) = {} \nfunción objetivo promedio:
         {} \nfunción objetivo de la peor solución: {}".format(
         statistics['Best solution']['f'],statistics['Mean']
         ,statistics['Worst solution']['f']))
```

```
f(x*) = -29768.0
función objetivo promedio: -29577.533333333333
función objetivo de la peor solución: -29331.0
```

Solución ingenua y estrategia ingenua.

```
In [33]: args = (getInitialSolution,1000.0,0.01)
         statistics = get_stats(Knapsack_, 30, args)

In [34]: print("f(x*) = {} \nfunción objetivo promedio:
         {} \nfunción objetivo de la peor solución: {}".format(
         statistics['Best solution']['f'],statistics['Mean']
         ,statistics['Worst solution']['f']))
```

```
f(x*) = -16269.0
función objetivo promedio: -15303.1
función objetivo de la peor solución: -14115.0
```

A.3. Programación Evolutiva

La librería **Pyristic** incluye una clase llamada `EvolutionaryProgramming` inspirada en la metaheurística de *Programación Evolutiva* (PE) para resolver problemas de minimización. Para trabajar con esta clase se requiere hacer lo siguiente:

1. Definir:
    - La función objetivo f.
    - La lista de restricciones.
    - Lista de límites inferiores y superiores.
    - Configuración de operadores de la metaheurística (opcional).
2. Crear una clase que hereda de `EvolutionaryProgramming`.
3. Sobreescribir las siguientes funciones:
    - mutation_operator (opcional)
    - adaptive_mutation (opcional)
    - survivor_selection (opcional)
    - initialize_step_weights (opcional)
    - initialize_population (opcional)
    - fixer (opcional)

A continuación se muestran los elementos que se deben importar.

Librerías externas

```
In [2]: from pprint import pprint
        import math
        import numpy as np
        import copy
```

```
In [4]: from IPython.display import Image
        from IPython.core.display import HTML
```

Componentes de `pyristic`

La estructura que está organizada la librería es:

- Las metaheurísticas están ubicadas en `heuristic`.
- Las funciones de prueba están ubicadas en `utils.test_function`.
- Las clases auxiliares para mantener la información de los operadores que serán empleados para alguna de las metaheurísticas basadas en los paradigmas del cómputo evolutivo están ubicadas en `utils.helpers`.

- Las metaheurísticas basadas en los paradigmas del cómputo evolutivo dependen de un conjunto de operadores (selección, mutación y cruza). Estos operadores están ubicados en `utils.operators`.

```
In [3]: from pyristic.heuristic.EvolutiveProgramming_search import\
        EvolutionaryProgramming
        from pyristic.utils.helpers import EvolutionaryProgrammingConfig,\
                get_stats, ContinuosFixer
        from pyristic.utils.test_function import beale_, ackley_
        from pyristic.utils.operators import mutation, selection
```

A.3.1. Clase `EvolutionaryProgramming`

Variables

- ***logger.*** Diccionario con información relacionada a la búsqueda con las siguientes llaves:
  - `best_individual`. Mejor individuo encontrado.
  - `best_f`. El valor obtenido de la función objetivo al evaluar al individuo en `best_individual`.
  - `current_iter`. Iteración actual de la búsqueda.
  - `total_iter`. Número total de iteraciones.
  - `parent_population_x`. Arreglo bidimensional de numpy. Cada fila representa a un individuo de la población y las columnas representan el número de variables de decisión.
  - `offspring_population_x`. Arreglo bidimensional de numpy. Cada fila representa a un individuo de la población y las columnas representan el número de variables de decisión.
  - `parent_population_sigma`. Arreglo de numpy que representa el desplazamiento de por variable de decisión de cada uno de los individuos.
  - `offspring_population_sigma`. Arreglo de numpy que representa el desplazamiento por variable de decisión de cada uno de los individuos.
  - `parent_population_f`. Arreglo de numpy que contiene el valor de la función objetivo para cada uno de los individuos de la población de parent_population_x.
  - `offspring_population_f`. Arreglo de numpy que contiene el valor de la función objetivo para cada uno de los individuos de la población de offspring_population_x.
- ***f.*** Función objetivo.
- ***Constraints.*** Lista de restricciones del problema. Las restricciones deben

ser funciones que retornan True o False, indicando si cumple dicha restricción.

- ***Bounds.*** Representa los límites definidos para cada una de las variables del problema. Se aceptan las siguientes representaciones:
  - Arreglo de numpy con solo dos componentes numéricas, donde, la primera componente es el límite inferior y la segunda componente es el límite superior. Esto significa que todas las variables de decisión estarán definidas para el mismo intervalo.
  - Arreglo bidimensional de numpy con dos arreglos de numpy, donde, el primer arreglo de numpy representa el límite inferior para cada variable de decisión, mientras, la segunda componente representa el límite superior para cada variable de decisión.
- ***Decision_variables.*** El número de variables de decisión del problema.

Métodos

- ***__init__.*** Constructor de la clase.
  Argumentos:
  - `function`. Función objetivo.
  - `decision_variables`. Número que indica las variables de decisión del problema.
  - `constraints`. Lista con las restricciones del problema.
  - `bounds`. Límites del espacio de búsqueda de cada una de las variables de decisión del problema.
  - `config`. Estructura de datos (`EvolutionaryProgrammingConfig`) con los operadores que se emplearán en la búsqueda.

  Valor de retorno:
  - Ninguno.
- ***optimize.*** método principal, realiza la ejecución de la metaheurística.
  Argumentos:
  - `generations`. Número de generaciones (iteraciones de la metaheurística).
  - `size_population`. Tamaño de la población (número de individuos).
  - `verbose`. Indica si se imprime en qué iteración se encuentra nuestra búsqueda. Por defecto, está en True.

- `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

Valor de retorno:

- Ninguno.

- ***mutatio_operator.*** Muta las variables de decisión que se encuentran almacenadas en el diccionario `logger` con la llave `parent_population_x`.
  Argumentos:
  - `**kwargs` Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

  Valor de retorno:
  - Un arreglo bidimensional de *numpy* representado a los nuevos individuos (se almacenarán en `logger` con la llave `offspring_population_x`).

- ***adaptive_mutation.*** Muta los tamaños de paso que se encuentran almacenados en el diccionario `logger` con la llave `parent_population_sigma`.
  Argumentos:
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

  Valor de retorno:
  - Un arreglo bidimensional de numpy con los tamaños de desplazamiento para cada una de las variables de decisión de los individuos (se almacenarán en `logger` con la llave `offspring_population_sigma`).

- ***survivor_selection.*** Selección de los individuos que pasarán a la siguiente generación.
  Argumentos:
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

  Valor de retorno:
  - Un diccionario con las siguientes llaves:
    - `parent_population_f`. El valor de la función objetivo de cada individuo que pasará a la siguiente generación.

- `parent_population_sigma`. El/los valor(es) de desplazamiento de los individuos seleccionados.
- `parent_population_x`. El vector $\boldsymbol{x}$ de cada uno de los individuos.

Por defecto la metaheurística utiliza el esquema de selección $(\mu + \lambda)$ que se encuentra en utils.operators.selection con el nombre de merge_selector.

- ***initialize_population.*** Crea una población de individuos aleatorios. Para ello se utiliza una distribución uniforme y se generan números aleatorios dentro de los límites indicados para cada variable. Los individuos generados son almacenados en `logger` con la llave `parent_population_x`. Esta función es llamada dentro de la función optimize.
  Argumentos:
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

  Valor de retorno:
  - Un arreglo bidimensional de *numpy*. El número de filas es igual al tamaño de la población y el número de columnas es igual al número de variables que tiene el problema que se está resolviendo.

- ***initialize_step_weights.*** Inicializa el tamaño de desplazamiento de cada una de las variables de decisión pertenecientes a cada individuo de la población. Para ello se generan números aleatorios en el intervalo $[0, 1]$, utilizando una distribución uniforme. Los tamaños de desplazamiento están almacenados en `logger` con la llave `parent_population_sigma`.
  Argumentos:
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

  Valor de retorno:
  - Un arreglo bidimensional de numpy. El número de filas es igual al tamaño de la población y el número de columnas es igual al número de variables que tiene el problema que se está resolviendo. Cada variable tiene su propio tamaño de paso.

- ***fixer.*** Si la solución no está dentro de los límites definidos para cada variable (restricciones de caja), actualiza el valor de la variable con el valor del límite que rebasó. De lo contrario, regresa la misma solución.
  Argumentos:
  - `ind`. Índice del individuo.

  Valor de retorno:

- Un arreglo de *numpy* que reemplazará la solución infactible de la población con la llave `offspring_population_x`.

A.3.2. Clase `EvolutionaryProgrammingConfig`

Variables

- ***mutation_op.*** Variable con el operador de mutación.
- ***survivor_selector.*** Variable con el esquema de selección de los individuos que pasan a la siguiente generación.
- ***fixer.*** Variable con una función auxiliar para los individuos que no cumplen las restricciones del problema.
- ***adaptive_mutation_op.*** Variable con el operador de mutación de los tamaños de paso σ.

Métodos

- ***mutate.*** Actualiza el operador de mutación de la variable `mutation_op`.
  Argumentos:
  - `mutate_`. Función o clase que realiza la mutación de la población almacenada con la llave `offspring_population_x`.

  Valor de retorno:
  - Retorna la configuración con la actualización del operador de cruza. El objetivo es poder aplicar varios operadores en cascada.

- ***survivor_selection.*** Actualiza el esquema de selección de la variable `survivor_selector`.
  Argumentos:
  - `survivor_function`.Función o clase que realiza la selección de individuos para la próxima generación.

  Valor de retorno:
  - Retorna la configuración con la actualización del operador de cruza. El objetivo es poder aplicar varios operadores en cascada.

- ***fixer_invalide_solutions.*** Actualiza la función auxiliar de la variable `fixer`.
  Argumentos:
  - `fixer_function`. Función o clase que ajustará los individuos de la población que no cumplen con al menos una de las restricciones del problema.

  Valor de retorno:

- Retorna la configuración con la actualización del operador de cruza. El objetivo es poder aplicar varios operadores en cascada.

- ***adaptive_mutation.*** Actualiza el operador de mutación de los σ de la variable `adaptive_mutation_op`.
  Argumentos:
  - `adaptive_mutation_function`. Función o clase que muta los tamaños de paso que se encuentran en `logger` con la llave `parent_population_sigma`.

  Valor de retorno:
  - Retorna la configuración con la actualización del operador de cruza. El objetivo es poder aplicar varios operadores en cascada.

A.3.3. Descripción de operadores

Los operadores de mutación, cruza y selección con los que cuenta la librería **Pyristic** son clases. La finalidad es unificar el formato de todos los operadores al ser llamados por los métodos de la clase `EvolutionaryProgramming`.

Operadores de mutación

sigma_mutator.

Operador de mutación en cada una de las soluciones de la población, donde, realiza la mutación de la siguiente manera:

$$x'_j = x_j + \sigma'_j \cdot N(0,1) \tag{A.5}$$

donde x'_j es la variable mutada, x_j la variable a mutar, σ'_j el tamaño de paso (previamente mutado) y N(0,1) devuelve un número aleatorio usando una distribución normal con media 0 y desviación estándar igual con 1.

Constructor:

- No recibe ningún argumento.

Métodos

- ***__call__.*** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `X`. Arreglo bidimensional de *numpy* representando a la población de soluciones de la iteración actual, donde, el número de filas es igual al tamaño de la población y el número de columnas es igual al número de variables que tiene el problema que se está resolviendo.
  - `Sigma`. Arreglo bidimensional de *numpy*, donde, cada fila representa los tamaños de paso y cada columna es una de las variables que tiene el problema que se está resolviendo.

  Valor de retorno:

  - Un arreglo bidimensional de *numpy* del mismo tamaño que el arreglo bidimensional de entrada X.

sigma_ep_adaptive_mutator.

Operador de mutación en los tamaños de desplazamiento de cada uno de los individuos de la población. La mutación se realiza de la siguiente manera:

$$\sigma_j' = \sigma_j \cdot (1 + \alpha \cdot N(0,1)) \tag{A.6}$$

donde σ_j' es la variable mutada, σ_j la variable a mutar, α parámetro de entrada por el usuario y N(0,1) devuelve un número aleatorio usando una distribución normal con media 0 y desviación estándar igual con 1.
Constructor:

- `decision_variables`. Número de variables de decisión del problema.
- `alpha`. Número que será empleado en la actualización de σ.

Métodos

- ***__call__*.** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `X`. Arreglo bidimensional de *numpy* que representa los tamaños de paso de cada uno de los individuos de la población.

  Valor de retorno:

  - Arreglo bidimensional de *numpy* con los nuevos valores de tamaño de paso.

Operadores de selección de sobrevivientes

merge_selector.

Esquema $(\mu + \lambda)$, selecciona μ individuos que son obtenidos al unir la población de hijos y la población actual. Los individuos que permanecerán en la próxima generación son aquellos que tengan un mejor valor de aptitud.
Constructor:

- No recibe ningún argumento.

Métodos

- ***__call__.*** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `parent_f`. Arreglo de numpy de la población almacenada en `parent_population_f`, donde, cada componente representa el valor de la función objetivo por el individuo i.
  - `offspring_f`. Arreglo de numpy de la población almacenada en `offspring_population_f`, donde, cada componente representa el valor de la función objetivo por el individuo i.
  - `features`. Diccionario que tiene las llaves de la información que se desea mantener. Cada llave contiene un arreglo de dos componentes, donde, la primera es la información de `parent_population` y la segunda componente es la información de `offspring_population`.

  Valor de retorno:

  - Diccionario con los individuos seleccionados por dicho esquema. Las llaves de este diccionario serán las mismas llaves recibidas en el parámetro features y adicional otra llave con el nombre `parent_population_f`, sin embargo, ahora sólo contendrá la información de los individuos que pasarán a la próxima generación.

replacement_selector.

El esquema (μ, λ), reemplaza la población actual con los μ mejores hijos de acuerdo a su valor de aptitud.
Constructor:

- No recibe ningún argumento.

Métodos

- ***__call__***. Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `parent_f`. Arreglo de numpy de la población almacenada en `parent_population_f`, donde, cada componente representa el valor de la función objetivo por el individuo i.
  - `offspring_f`. Arreglo de numpy de la población almacenada en `offspring_population_f`, donde, cada componente representa el valor de la función objetivo por el individuo i.
  - `features`. Diccionario que tiene las llaves de la información que se desea mantener. Cada llave contiene un arreglo de dos componentes, donde, la primera es la información de `parent_population` y la segunda componente es la información de `offspring_population`.

  Valor de retorno:

  - Diccionario con los individuos seleccionados por dicho esquema. Las llaves de este diccionario serán las mismas llaves recibidas en el parámetro features y adicional otra llave con el nombre `parent_population_f`, sin embargo, ahora sólo contendrá la información de los individuos que pasarán a la próxima generación.

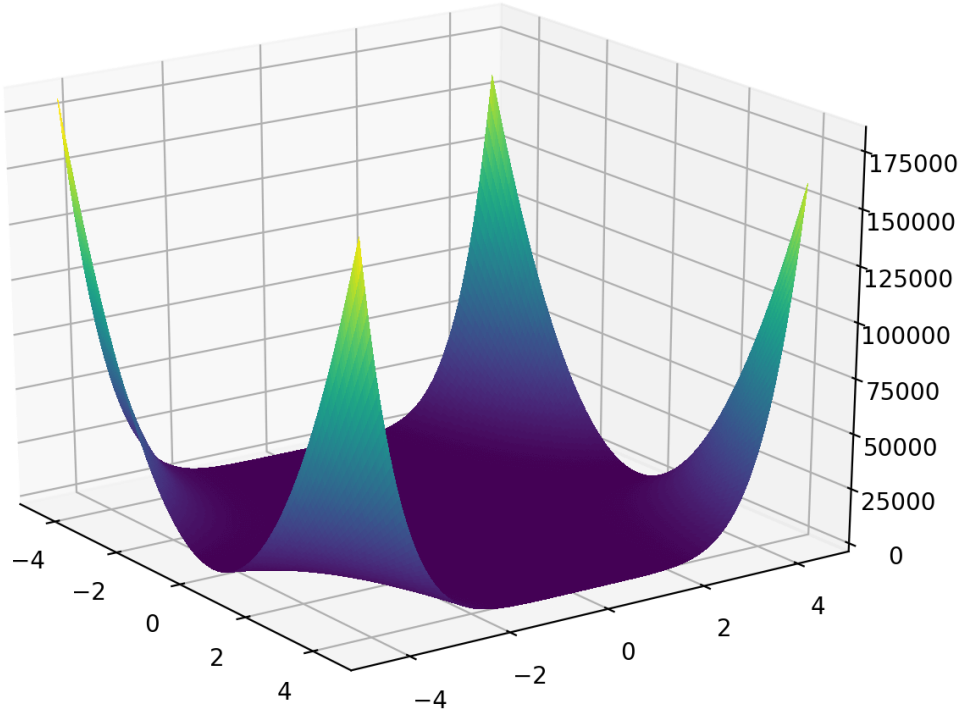
A.3.4. Función de Beale

$$\begin{aligned} \text{minimizar:} \quad & f(x_1, x_2) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + \\ & (2.625 - x_1 + x_1 x_2^3)^2 \\ \text{tal que:} \quad & -4.5 \leq x_1, x_2 \leq 4.5 \end{aligned} \tag{A.7}$$

El mínimo global se encuentra en $x^* = (3, 0.5)$ y $f(x^*) = 0$.

```
In [5]: Image(filename="include/beale.png", width=500, height=300)
```

```
Out[5]:
```



Para inicializar un objeto de la clase `EvolutionaryProgramming`, es necesario implementar los siguientes elementos.

Función objetivo

```
In [ ]: def f(x : np.ndarray) -> float:
            a = (1.5 - x[0] + x[0]*x[1])**2
            b = (2.25 - x[0] + x[0]*x[1]**2)**2
            c = (2.625 - x[0] + x[0]*x[1]**3)**2
            return a+b+c
```

Función de aptitud

Los algoritmos evolutivos, se define una función llamada **aptitud** que describe a cada individuo que tan bueno es en el problema a resolver. Los valores más grandes de aptitud corresponden a las mejores soluciones. Para el problema de Beale, la función de aptitud es: $F_a(i) = \frac{1}{f(x_1^{(i)}, x_2^{(i)})+1}$.

```
In [5]: def aptitudeBeale(X: np.array) -> np.array:
            return 1/ ( beale_['function'](X)+1)
```

Restricciones

Las restricciones se definen como una lista de funciones que retornan valores booleanos, estos valores permiten revisar si una solución es factible o no. En el caso de la función de Beale, solo se tienen restricciones de caja.

```
In [ ]: def is_feasible(x : np.ndarray) -> bool:
            for i in range(len(x)):
                if -4.5>x[i] or x[i] > 4.5:
                    return False
            is_feasible.__doc__="x1: -4.5 <= {:.2f} <= 4.5 \n
                         x2: -4.5 <= {:.2f} <= 4.5".format(x[0],x[1])

            return True

        constraints_ = [is_feasible]
```

Límites de las variables del problema

Los límites de las variables de decisión son establecidos en una matriz (lista de listas) de tamaño (2 x n), donde n es el número de variables. La primera fila contiene los límites inferiores de cada una de las variables de decisión y la segunda fila los límites superiores.

```
In [8]: lower_bound = [-4.5,-4.5]
        upper_bound = [4.5, 4.5]
        bounds = [lower_bound, upper_bound]
```

Nota:
En el caso de que todas las variables de decisión se encuentren en el mismo rango de búsqueda, se puede utilizar una única lista con dos valores numéricos, donde, el primer valor representa el límite inferior y el segundo el límite superior.

La función de Beale es una función de dos variables (x_1, x_2). La dos variables están acotadas en el mismo espacio de búsqueda $-4.5 < x_i < 4.5, i \in [1, 2]$, entonces, en lugar de emplear la representación descrita antes, podemos sustituirla por:

```
bounds = [-4.5, 4.5]
```

En esta representación, la primera componente se refiere al límite inferior, mientras, la segunda componente es el límite superior. Este arreglo será interpretado como el espacio de búsqueda para todas las variables de decisión.
La librería **Pyristic** tiene implementados algunos problemas de prueba en `utils.helpers.test_function`, entre ellos la función de Beale. Los problemas de prueba están definidos como diccionarios con las siguientes llaves:

- `function`. Función objetivo.
- `contraints`. Restricciones del problema, lista con al menos una función que regresa un valor booleano.
- `bounds`. Límites para cada una de las variables del problema. En el caso de que todas las variables del problema se encuentren en el mismo intervalo de búsqueda, se puede emplear una lista con dos valores numéricos.
- `decision_variables`. Número de variables de decisión.

Declaración de `EvolutionaryProgramming`

La metaheurística de PE implementada en la librería **Pyristic** se puede utilizar de las siguientes maneras:

- Crear una clase que herede de la clase `EvolutionaryProgramming` y sobreescribir las funciones antes mencionadas.
- Declarar un objeto del tipo `EvolutionaryProgrammingConfig` y ajustar los operadores.
- Realizar una combinación de las dos anteriores.

Es importante resaltar que se puede hacer uso de `EvolutionaryProgramming` sin modificar los operadores que tiene por defecto.

Ejecución de la metaheurística

Como mencionamos antes, una forma de utilizar la metaheurística es hacer una instancia de la clase `EvolutionProgramming` dejando su configuración por defecto.
Los argumentos que se deben indicar al momento de inicializar son:

- Función objetivo.
- Restricciones del problema.
- Límite inferior y superior (por cada variable de decisión).
- Número de variables que tiene el problema.

```
In [10]: Beale_optimizer = EvolutionaryProgramming(
                            function= aptitudeBeale,\
                            decision_variables=beale_['decision_variables'],\
                            constraints= beale_['constraints'],\
                            bounds= beale_['bounds'])
```

Recordemos que `beale_` es un diccionario con la información requerida por el constructor de la clase `EvolutionaryProgramming`.
Finalmente, se llama a la función `optimize`. Recibe los siguientes parámetros:

- **generations**. Número de generaciones (iteraciones de la metaheurística).
- **size_population**. Tamaño de la población (número de individuos).
- **verbose**. Muestra en qué iteración se encuentra nuestra búsqueda, por defecto está en True.
- ****kwargs****. Argumentos externos a la búsqueda.

Para resolver la función de Beale utilizaremos los siguientes parámetros:

- **generations** = 200
- **size_population** = 100
- **verbose** = True.

```
In [11]: Beale_optimizer.optimize(200,100)

100% 200/200 [00:00<00:00, 271.94it/s]


In [12]: print(Beale_optimizer)

Evolutive Programming search:
 f(X) = 2.0092647591277975e-21
 X = [3.  0.5]
 Constraints:
 x1: -4.5 <= 3.00 <= 4.5
 x2: -4.5 <= 0.50 <= 4.5
```

Para revisar el comportamiento de la metaheurística en determinado problema, la librería **Pyristic** cuenta con una función llamada `get_stats`. Esta función se encuentra en `utils.helpers` y recibe como parámetros:

- Objeto que realiza la búsqueda de soluciones.
- El número de veces que se quiere ejecutar la metaheurística.
- Los argumentos que recibe la función `optimize` (debe ser una tupla).
- Argumentos adicionales a la búsqueda, estos argumentos deben estar contenidos en un diccionario (opcional).

La función `get_stats` retorna un diccionario con algunas estadísticas de las ejecuciones.

```
In [13]: args = (700, 100, False)
         statistics = get_stats(Beale_optimizer, 21, args)
```

```
In [13]: args = (200, 100, False)
         statistics = get_stats(Beale_optimizer, 30, args,
                                transformer=beale_['function'])
```

```
{'Best solution': {'f': 0.0, 'x': array([3. , 0.5])},
 'Mean': 3.239004768779724e-28,
 'Median': 0.0,
 'Standard deviation': 1.2685335049178072e-27,
 'Worst solution': {'f': 5.945795635558406e-27, 'x': array([3. , 0.5])}}
```

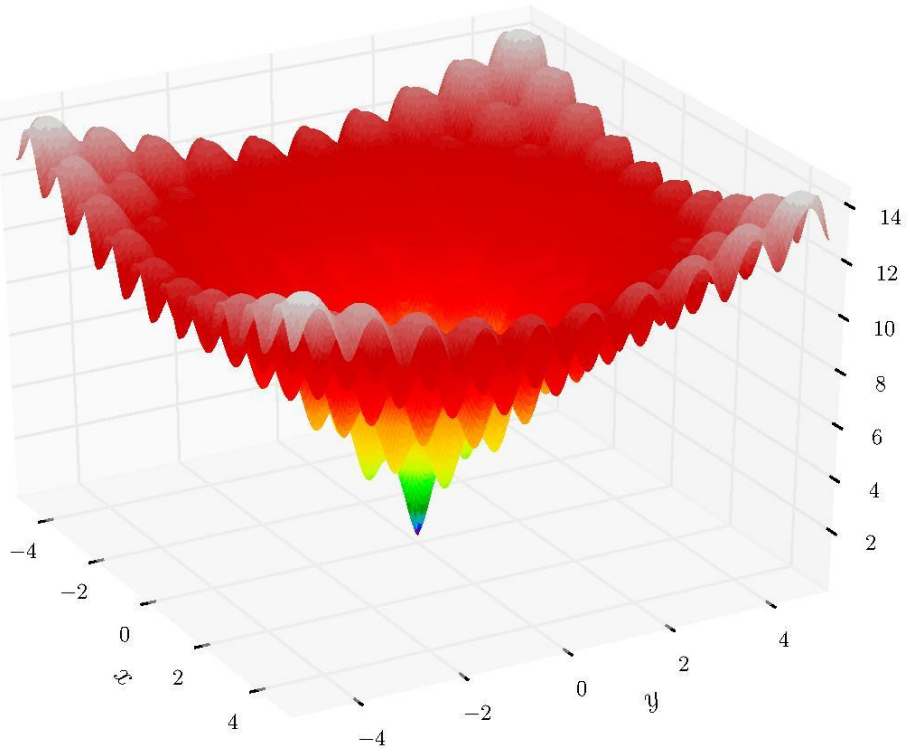
A.3.5. Función de Ackley

$$\text{mín } f(\boldsymbol{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n}x_i^2}\right) - exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e \quad \text{(A.8)}$$

El mínimo global está en $x^* = 0$ y $f(\boldsymbol{x}) = 0$ y su dominio es $|x_i| < 30$.

```
In [15]: Image(filename="include/ackley.jpg", width=500, height=300)
```

```
Out[15]:
```



```
In [16]: ackley_

Out[16]: {'function': CPUDispatcher(<function ackley_function at 0x7ffb974a8378>),
          'constraints': [CPUDispatcher(<function constraint1_ackley at 0x7ffb974a862
          'bounds': [-30.0, 30.0],
          'decision_variables': 10}
```

Función de aptitud

Los algoritmos evolutivos, se define una función llamada **aptitud** que describe a cada individuo que tan bueno es en el problema a resolver. Los valores más grandes de aptitud corresponden a las mejores soluciones. Para el problema de Beale, la función de aptitud es: $F_a(\boldsymbol{X}) = -f(\boldsymbol{X})$.

```
In [16]: def aptitudeAckley(x):
             return -1* ackley_['function'](x)
```

En el caso de la función de Ackley, al no estár con restricción de variables de decisión, podemos modificar el número de variables de decisión que tiene por defecto de la siguiente manera:

```
ackley_['decision_variable'] = 5
```

Para resolver la función de Ackley con 10 variables de decisión utilizaremos los siguientes parámetros:

- **generations** = 500
- **size_population** = 100

```
In [17]: Optimizer_by_default = EvolutionaryProgramming(
                               function= aptitudeAckley,\
                               decision_variables=ackley_['decision_variables'],\
                               constraints= ackley_['constraints'],\
                               bounds= ackley_['bounds'])

In [18]: Optimizer_by_default.optimize(500,100)

100% 500/500 [00:01<00:00, 287.94it/s]


In [19]: print(Optimizer_by_default)
```

```
Evolutive Programming search:
 f(X) = 14.236808930846905
 X = [ 2.05309403e+00 -3.99185951e+00  1.23496472e-02  2.24880725e-04
  6.01696518e-09 -5.80408096e+00 -1.40003025e+01  6.90122701e+00
 -9.91887617e-01  7.98370535e+00]
 Constraints:
 x1: -30 <= 2.05 <= 30
 x2: -30 <= -3.99 <= 30
 x3: -30 <= 0.01 <= 30
 x4: -30 <= 0.00 <= 30
 x5: -30 <= 0.00 <= 30
 x6: -30 <= -5.80 <= 30
 x7: -30 <= -14.00 <= 30
 x8: -30 <= 6.90 <= 30
 x9: -30 <= -0.99 <= 30
 x10: -30 <= 7.98 <= 30
```

La solución encontrada por PE no es la solución óptima. A continuación mostraremos la ejecución de la metaheurística modificando la configuración por defecto.

Declaración de `EvolutionaryProgramming` por configuración

A continuación vamos a mostrar la forma en que se declara un objeto del tipo `EvolutionaryProgrammingConfig` que es una clase auxiliar, donde, contendrá los operadores que se emplea en la ejecución de la metaheurística.

```
In [20]: configuration = (\
    EvolutionaryProgrammingConfig()
    .adaptive_mutation(\
        mutation.sigma_ep_adaptive_mutator(ackley_['decision_variables'], 2.0)
                           )
                       )
```

En este ejemplo mostraremos el impacto de inicializar el operador de mutación en los tamaños de paso con $\alpha = 2$, el constructor en caso de no incluir este parámetro lo inicializa con $\alpha = 0.5$.

```
In [21]: print(configuration)

--------------------------------
        Configuration
```

```
--------------------------------
Adaptive mutation: Sigma EP.
                -Alpha: 2.0

--------------------------------
```

```
In [28]: Optimizer_by_configuration = EvolutionaryProgramming(
                              function= aptitudeAckley,\
                              decision_variables=ackley_['decision_variables'],\
                              constraints= ackley_['constraints'],\
                              bounds= ackley_['bounds'], config=configuration)
```

A diferencia del ejemplo de la función de Beale, incluimos la configuración de los operadores que deseamos utilizar en la variable llamada `config` al crear un objeto de la clase `EvolutionaryProgramming`.

```
In [23]: Optimizer_by_configuration.optimize(500,100)

100% 500/500 [00:01<00:00, 286.96it/s]
```

```
In [24]: print(Optimizer_by_configuration)

Evolutive Programming search:
 f(X) = 1.904985106376913
 X = [-0.193743   -1.12343605  0.11622617  0.13151275  0.03091606  0.01765212
 -0.10585762 -0.05992731 -0.04231196  0.07508058]
 Constraints:
 x1: -30 <= -0.19 <= 30
 x2: -30 <= -1.12 <= 30
 x3: -30 <= 0.12 <= 30
 x4: -30 <= 0.13 <= 30
 x5: -30 <= 0.03 <= 30
 x6: -30 <= 0.02 <= 30
 x7: -30 <= -0.11 <= 30
 x8: -30 <= -0.06 <= 30
 x9: -30 <= -0.04 <= 30
 x10: -30 <= 0.08 <= 30
```

Herencia desde *EvolutionaryProgramming*

Otra forma de utilizar la metaheurística de nuestra librería es definiendo una clase que herede de `EvolutionaryProgramming`, donde, vamos a sobreescribir el método `adaptive_mutation` y así permitir incluir distintos valores para α.

```
In [25]: class EPAckley(EvolutionaryProgramming):
             def __init__(
                     self, function,\
                     decision_variables, constraints,\
                     bounds):
                 super().__init__(
                         function, decision_variables,\
                         constraints, bounds)

             def adaptive_mutation(self, **kwargs):
                 alpha_ = kwargs['alpha']
                 return mutation.sigma_ep_adaptive(\
                             self.logger['parent_population_sigma'], alpha_)

In [26]: additional_arguments = {'alpha':2.0}
```

El diccionario que hemos definido tiene el parámetro que se utiliza en la función `adaptive_mutation`.

```
In [27]: Optimizer_by_class = EPAckley(
                              function= aptitudeAckley,\
                              decision_variables=ackley_['decision_variables'],\
                              constraints= ackley_['constraints'],\
                              bounds= ackley_['bounds'])

In [28]: Optimizer_by_class.optimize(500,100,**additional_arguments)

100% 500/500 [00:01<00:00, 326.76it/s]


In [29]: print(Optimizer_by_class)

Evolutive Programming search:
 f(X) = 1.3313378339717095
 X = [-0.03217673 -0.20545881 -0.02026904  0.0380225  -0.04857584  0.13552289
  0.18398808 -0.18142151  0.04354127 -0.24854929]
 Constraints:
 x1: -30 <= -0.03 <= 30
 x2: -30 <= -0.21 <= 30
```

```
x3: -30 <= -0.02 <= 30
x4: -30 <= 0.04 <= 30
x5: -30 <= -0.05 <= 30
x6: -30 <= 0.14 <= 30
x7: -30 <= 0.18 <= 30
x8: -30 <= -0.18 <= 30
x9: -30 <= 0.04 <= 30
x10: -30 <= -0.25 <= 30
```

Resultados

Usando la configuración por defecto

```
In [30]: args = (500,100,False)
         statistics = get_stats(Optimizer_by_default, 21, args,
                transformer= ackley_['function'])

In [31]: pprint(statistics)

{'Best solution': {'f': 9.772165021436392,
                   'x': array([-1.98705450e+00, -2.00414714e-01,
                               -1.97284063e+00,  5.39641110e-04,
                                6.95424646e+00, -2.35497197e-02,
                                -5.95536851e+00,  1.98709111e+00,
                                -1.01744137e+00,  2.99379761e+00])},
 'Mean': 14.438796656966852,
 'Median': 14.57877829622885,
 'Standard deviation': 1.9055002610689908,
 'Worst solution': {'f': 17.81753107723958,
                    'x': array([ 13.02996391,  -2.99887137,
                                 13.98922842,  13.99468984,
                                 -0.96250749,   8.99658661,
                                 -10.05791112,   0.99054141,
                                 20.99209058,   4.00012043])}}
```

Indicando la configuración que va a utilizar PE

```
In [32]: args = (500,100,False)
         statistics = get_stats(Optimizer_by_configuration, 21, args,
            transformer=ackley_['function'])
```

```
In [33]: pprint(statistics)

{'Best solution': {'f': 0.12388792866390874,
                   'x': array([ 0.05835097,  0.01067072,
                                0.01612713,  0.00163387,
                                0.01306192,-0.00391465,
                                0.00555765,  0.00590317,
                                0.02159629,  0.03295279])},
 'Mean': 3.3380048902920185,
 'Median': 2.171638680046055,
 'Standard deviation': 3.0037931500732697,
 'Worst solution': {'f': 12.526042771165503,
                    'x': array([ 6.40082881e-02, -1.90140665e+00,
                                 1.23486372e-02,  4.37177466e-02,
                                 5.15705963e+00, -7.71560336e-02,
                                 9.83440971e-01, -1.96073006e+00,
                                -1.05264738e+00,  1.32558098e+01])}}
```

Creando una clase que hereda de *EvolutionaryProgramming*

```
In [34]: args = (500,100,False)
         statistics = get_stats(Optimizer_by_class, 21, args, additional_arguments,
                transformer=ackley_['function'])

In [35]: pprint(statistics)

{'Best solution': {'f': 0.025376897823943256,
                   'x': array([-0.00120468,  0.00352147,
                               -0.0045472 , -0.00741508,
                                0.00987255,0.00907506,
                                0.0033641 , -0.00051152,
                                0.00412521,  0.00694506])},
 'Mean': 3.1769907426285546,
 'Median': 2.922135369198639,
 'Standard deviation': 1.9952015696873395,
 'Worst solution': {'f': 8.32256538117469,
                    'x': array([-1.25405474, -0.44577715,
                                -0.25906636,  4.17355772,
                                -0.77957849,0.80523909,
                                2.54725241, -2.06024977,
                                -2.65550508, -0.35793377])}}
```

Las últimas dos estrategias utilizan la misma configuración, la única diferencia es la forma en que se han creado y los resultados obtenidos pueden variar por los números aleatorios generados.

A.4. Estrategias Evolutivas

La librería **Pyristic** incluye una clase llamada `EvolutionStrategy`, inspirada en la metaheurística de *Estrategias evolutivas* (EE), para resolver problemas de minimización. Para trabajar con esta clase se requiere hacer lo siguiente:

1. Definir:
    - La función objetivo f.
    - La lista de restricciones.
    - Lista de límites inferiores y superiores.
    - Configuración de operadores de la metaheurística (opcional).
2. Crear una clase que herede de `EvolutionStrategy`.
3. Sobreescribir las siguientes funciones de la clase `EvolutionStrategy`:
    - initialize_step_weights (opcional)
    - initialize_population (opcional)
    - fixer (opcional)
    - mutation_operator (opcional)
    - crossover_operator (opcional)
    - adaptive_crossover (opcional)
    - adaptive_mutation (opcional)
    - survivor_selection (opcional)

A continuación se mostrará cómo resolver dos problemas de optimización continua usando la clase `EvolutionStrategy`. El primer paso es importar la librería y los módulos que se van a utilizar.

Librerías externas

```
In [2]: from pprint import pprint
        import math
        import numpy as np
        import copy
        from IPython.display import Image
        from IPython.core.display import HTML
```

Componentes de `pyristic`

La estructura que está organizada la librería es:

- Las metaheurísticas están ubicadas en `heuristic`.
- Las funciones de prueba están ubicadas en `utils.test_function`.

- Las clases auxiliares para mantener la información de los operadores que serán empleados para alguna de las metaheurísticas basadas en los paradigmas del cómputo evolutivo están ubicadas en `utils.helpers`.
- Las metaheurísticas basadas en los paradigmas del cómputo evolutivo dependen de un conjunto de operadores (selección, mutación y cruza). Estos operadores están ubicados en `utils.operators`.

Para demostrar el uso de nuestra metaheurística basada en *estrategias evolutivas* tenemos que importar la clase llamada `EvolutionStrategy_search` que se encuentra en `heuristic.EvolutionStrategy`.

```
In [3]: from pyristic.heuristic.EvolutionStrategy_search import EvolutionStrategy
        from pyristic.utils.operators import selection, mutation, crossover
        from pyristic.utils.test_function import beale_, ackley_
        from pyristic.utils.helpers import EvolutionStrategyConfig,get_stats
```

A.4.1. Clase `EvolutionStrategy`

Variables

- ***logger.*** Diccionario con información relacionada a la búsqueda.
  - `best_individual`. Individuo con el mejor valor encontrado en la función objetivo.
  - `best_f`. Valor de la función objetivo.
  - `current_iter`. Iteración actual de la búsqueda.
  - `total_iter`. Número total de iteraciones.
  - `parent_population_size`. Tamaño de la población de padres.
  - `offspring_population_size`. Tamaño de la población de hijos.
  - `parent_population_x`. Arreglo bidimensional de *numpy*. Cada fila representa un individuo de la población actual y cada columna corresponde a una variable de decisión.
  - `offspring_population_x`. Arreglo bidimensional de *numpy*. Cada fila representa un individuo de la población de hijos y cada columna corresponde a una variable de decisión.
  - `parent_population_sigma`. Arreglo de *numpy*, donde, cada elemento representa el desplazamiento de todas las variables de decisión de cada individuo o un arreglo bidimensional de *numpy*, donde, cada fila representa el desplazamiento de cada una de las variables de decisión de un individuo.

- `offspring_population_sigma`.Arreglo de *numpy*, donde, cada elemento representa el desplazamiento de todas las variables de decisión de un individuo o un arreglo bidimensional de *numpy*, donde, cada fila representa el desplazamiento de cada una de las variables de decisión de un individuo.
- `parent_population_f`. Arreglo de *numpy* que contiene el valor de la función objetivo para cada uno de los individuos de la población con la llave `parent_population_x` en `logger`.
- `offspring_population_f`. Arreglo de *numpy* que contiene el valor de la función objetivo para cada uno de los individuos de la población con la llave `offspring_population_x`.

- ***f.*** Función objetivo.
- ***Constraints.*** Lista de restricciones del problema. Las restricciones deben ser funciones que retornan True o False, indicando si cumple dicha restricción.
- ***Bounds.*** Representa los límites definidos para cada una de las variables del problema. Se aceptan las siguientes representaciones:
  - Arreglo de *numpy* con solo dos componentes numéricas, donde, la primera componente es el límite inferior y la segunda componente es el límite superior. Esto significa que todas las variables de decisión estarán definidas para el mismo intervalo.
  - Arreglo bidimensional de *numpy* con únicamente dos filas, donde, la primer fila es el límite inferior para cada variable de decisión, mientras, la segunda fila representa el límite superior para cada variable de decisión.
- ***Decision_variables.*** El número de variables de decisión del problema.

Métodos

- ***__init__.*** Constructor de la clase.
  Argumentos:
  - `function`. Función objetivo.
  - `decision_variables`. Número de variables de decisión del problema.
  - `constraints`. Lista con las restricciones del problema (se describe los tipos de datos admisibles en el apartado de variables de la clase con el nombre `Constraints`).

- `bounds`. Límites de las variables de decisión (se describe los tipos de datos admisibles en el apartado de variables de la clase con el nombre `Bounds`).
- `config`. Estructura de datos (`EvolutionStrategyConfig`) con los operadores que se emplearán en la búsqueda.

Valor de retorno:

- Ninguno.

- ***optimize.*** Método principal, realiza la ejecución de la metaheurística.
  Argumentos:
  - `generations`. Número de generaciones (iteraciones de la metaheurística).
  - `population_size`. Tamaño de la población (número de individuos).
  - `offspring_size`. Tamaño de la población creada a partir de los operadores de cruza y mutación.
  - `eps_sigma`. Valor mínimo que pueden tener los tamaños de paso. Por defecto, está en 0.001.
  - `verbose`. Indica si se imprime en qué iteración se encuentra nuestra búsqueda. Por defecto, está en True.
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

  Valor de retorno:
  - Ninguno.

- ***fixer***. Si la solución no está dentro de los límites definidos para cada variable (restricciones de caja), actualiza el valor de la variable con el valor del límite que rebasó. De lo contrario, regresa la misma solución.
  Argumentos:
  - `ind`. Índice del individuo.

  Valor de retorno:
  - Un arreglo de *numpy* que reemplazará la solución infactible.

- ***initialize_population***. Crea una población de individuos aleatorios. Para ello se utiliza una distribución uniforme y se generan números aleatorios dentro de los límites indicados para cada variable. Los individuos generados son almacenados en `logger` con la llave `parent_population_x`. Esta función es llamada dentro de la función `optimize`.
  Argumentos:

- `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

Valor de retorno:

- Un arreglo bidimensional de *numpy*. Cada fila representa un individuo, cada columna indica los valores para cada variable de decisión del individuo.

- ***initialize_step_weights***. Inicializa el tamaño de desplazamiento de cada individuo de la población, por defecto se emplea un sigma por cada variable de decisión en cada uno de los individuos. Para ello se generan números aleatorios en el intervalo $[0, 1]$, utilizando una distribución uniforme. Los tamaños de desplazamiento están almacenados en logger con la llave `parent_population_sigma`.
  Argumentos:
  - `eps_sigma`. Valor mínimo que pueden tomar sigma (tamaños de paso).
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

  Valor de retorno:
  - Un arreglo bidimensional de *numpy*. Cada fila almacena la información de los tamaños de paso de cada individuo, cada columna pertenece al tamaño de paso de una de las variables de decisión.

- ***crossover_operator.*** Genera λ hijos (individuos nuevos), aplicando una recombinación sexual. Es decir, se seleccionan dos individuos aleatoriamente de `parent_population_x` que actuarán como padres y generarán un hijo. Este procedimiento se repite λ veces. Los nuevos individuos se almacenan en `logger` con la llave `offspring_population_x`.
  Argumentos:
  - `parent_ind1`. Índices de los individuos que son seleccionados para actuar como padre 1.
  - `parent_ind2`. Índices de los individuos que son seleccionados para actuar como padre 2.
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

  Valor de retorno:
  - Un arreglo bidimensional de *numpy* con los valores de las variables de decisión de los nuevos individuos.

Por defecto la metaheurística utiliza el operador de cruza discreta que se encuentra en `utils.operators.crossover` con el nombre de `discrete`.

- ***mutation_operator.*** Muta las variables de decisión de los individuos creados con el operador de cruza. Estos individuos están almacenados en el diccionario `logger` con la llave `offspring_population_x`. La mutación se realiza de la siguiente forma por defecto:

$$x_i' = x_i + \sigma_i' \cdot N_i(0,1) \tag{A.9}$$

donde x_i' es la variable mutada, x_i la variable a mutar, σ_i' el tamaño de paso (previamente mutado) y $N_i(0,1)$ devuelve un número aleatorio por cada variable de decisión utilizando una distribución normal con media 0 y desviación estándar igual con 1.

Nota:
Es importante tener en cuenta que la fila j de `offspring_population_x` debe corresponder con la fila j de `offspring_population_sigma`.

Argumentos:

- `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

Valor de retorno:

- Un arreglo bidimensional de numpy que almacena los nuevos valores de las variables de decisión.

- ***adaptive_crossover.*** Genera los tamaños de paso de los nuevos individuos, aplicando una recombinación sexual. Utiliza las mismas parejas de padres que se usaron con el operador `crossover_operator`. Los nuevos tamaños de paso son almacenados en `logger` con la llave `offspring_population_sigma`.
Argumentos:
  - `parent_ind1`. Índices de los individuos que son seleccionados para actuar como padre 1.
  - `parent_ind2`. Índices de los individuos que son seleccionados para actuar como padre 2.
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

  Valor de retorno:

- Un arreglo bidimensional de *numpy* con los valores de los nuevos tamaños de paso.

Por defecto la metaheurística utiliza el operador de cruza intermedia que se encuentra en `utils.operators.crossover` con el nombre de `intermediate`.

- ***adaptive_mutation.*** Muta los tamaños de paso que se encuentran almacenados en el diccionario `logger` con la llave `offspring_population_sigma`. Este método se ejecuta antes del método `mutation_operator`. La mutación se realiza de la siguiente forma por defecto:

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

Donde,

- σ_i es el tamaño de paso actual.
- σ_i' es tamaño de paso mutado.
- τ está definido como $\frac{1}{\sqrt{2n}}$, donde, n es el número de variables del problema.
- τ' está definido como $\frac{1}{\sqrt{2\sqrt{n}}}$, donde, n es el número de variables del problema.
- $N(0,1)$ devuelve un número aleatorio usando una distribución normal con media 0 y desviación estándar igual a 1. Es importante notar que se genera un único número aleatorio para todas las σ_i.
- $N_i(0,1)$ devuelve un número aleatorio por σ_i utilizando una distribución normal con media 0 y desviación estandas igual a 1.

Argumentos:

- `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

Valor de retorno:

- Un arreglo bidimensional de *numpy* con los tamaños de desplazamiento para cada una de las variables de decisión de los individuos.

- ***survivor_selection.*** Selecciona los individuos que formarán parte de la siguiente generación.
Argumentos:

- `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobreescribe alguno de los métodos que tiene la clase.

Valor de retorno:

- Un diccionario con las siguientes llaves:
  - `parent_population_fitness`. El valor de aptitud de cada individuo que pasará a la siguiente generación.
  - `parent_population_sigma`. El/los valor(es) de desplazamiento de los individuos seleccionados.
  - `parent_population_x`. El vector $\boldsymbol{x}$ de cada uno de los individuos.

Por defecto la metaheurística utiliza el esquema de selección $(\mu + \lambda)$ que se encuentra en `utils.operators.selection` con el nombre de `merge_selector`.

A.4.2. Clase `EvolutionStrategyConfig`

Variables

- ***cross_op.*** Variable con el operador de cruza.
- ***mutation_op.*** Variable con el operador de mutación.
- ***survivor_selector.*** Variable con el esquema de selección que decide cuáles individuos pasan a la siguiente generación.
- ***fixer.*** Variable con una función que determina qué hacer con los individuos que no cumplen las restricciones del problema.
- ***adaptive_crossover_op.*** Variable con el operador de cruza que se aplica a los tamaños de paso σ.
- ***adaptive_mutation_op.*** Variable con el operador de mutación que se aplica a los tamaños de paso σ.

Métodos

- ***cross.*** Actualiza el operador de cruza de la variable `cross_op`.
  Argumentos:
  - `crossover_`. Función o clase que realiza la cruza de la población almacenada con la llave `parent_population_x`.

  Valor de retorno:
  - Retorna la configuración con la actualización del operador de cruza. El objetivo es poder aplicar varios operadores en cascada.

- ***mutate.*** Actualiza el operador de mutación de la variable `mutation_op`.

Argumentos:

- `mutate_`. Función o clase que realiza la mutación de la población almacenada con la llave `offspring_population_x`.

Valor de retorno:

- Retorna la configuración con la actualización del operador de mutación. El objetivo es poder aplicar varios operadores en cascada.

- ***survivor_selection.*** Actualiza el esquema de selección de la variable `survivor_selector`. Argumentos:
  - `survivor_function`. Función o clase que realiza la selección de individuos que formarán parte de la siguiente generación.

  Valor de retorno:
  - Retorna la configuración con la actualización del esquema de selección de sobrevivientes. El objetivo es poder aplicar varios operadores en cascada.

- ***fixer_invalide_solutions.*** Actualiza la función de la variable `fixer`. Argumentos:
  - `fixer_function`. Función o clase que ajustará los individuos de la población que no cumplen con las restricciones del problema.

  Valor de retorno:
  - Retorna la configuración con la actualización de la función auxiliar. El objetivo es poder aplicar varios operadores en cascada.

- ***adaptive_crossover.*** Actualiza el operador de cruza de los σ de la variable `adaptive_crossover_op`. Argumentos:
  - `adaptive_crossover_function`. Función o clase que cruza los tamaños de paso de los individuos seleccionados para la cruza. Estos tamaños de paso están almacenados en el diccionario `logger` con la llave `offspring_population_sigma`.

  Valor de retorno:
  - Retorna la configuración con la actualización del operador de cruza en los tamaños de paso. El objetivo es poder aplicar varios operadores en cascada.

- ***adaptive_mutation.*** Actualiza el operador de mutación de los σ de la variable `adaptive_mutation_op`. Argumentos:
  - `adaptive_mutation_function`. Función o clase que muta los tamaños de paso que se encuentran en `logger` con la llave `offspring_population_sigma`.

Valor de retorno:

- Retorna la configuración con la actualización del operador de mutación en los tamaños de paso. El objetivo es poder aplicar varios operadores en cascada.

A.4.3. Descripción de operadores

Los operadores de mutación, cruza y selección con los que cuenta la librería **Pyristic** son clases. La finalidad es unificar el formato de todos los operadores al ser llamados por los métodos de la clase `EvolutionStrategy`.

Operadores de mutación

Operadores de mutación en los tamaños de paso Los operadores de mutación en los tamaños de paso es necesario definir la propiedad `length` (método con el decorador *@property*) porque este método será llamado al inicializar `parent_population_sigma` por la clase `EvolutionStrategy`. El objetivo es conocer si será un σ por individuo o por cada variable de decisión de cada uno de los individuos.

single_sigma_adaptive_mutator

Muta el valor del tamaño de paso σ, utilizado para mutar todas las variables de decisión de un individuo. La mutación se realiza como sigue:

$$\sigma' = \sigma \cdot e^{\tau \cdot N(0,1)} \tag{A.10}$$

Donde τ es un parámetro que proporciona el usuario. Sea n el número de variables de decisión del problema, su valor por defecto es:

$$\tau = \frac{1}{\sqrt{n}} \tag{A.11}$$

Constructor:

- `decision_variables`. Número de variables de decisión del problema.

Métodos

- ***length.*** Función auxiliar de la clase `EvolutionStrategy` que indica cuántos tamaños de paso se utilizan para cada individuo. En este caso cada individuo utiliza un único tamaño de paso.

  Argumentos:

  - No recibe ningún argumento.

  Valor de retorno:

  - Número de sigma's empleados para cada individuo de la población (este operador retorna 1).

- **__call__.** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `sigma`. Arreglo de *numpy* con m valores σ. m es el tamaño de la población.

  Valor de retorno:

  - Arreglo *numpy* con los nuevos valores de σ'.

mult_sigma_adaptive_mutator

Muta los valores de los tamaños de paso, considerando el uso de un tamaño de paso por variable de decisión. La mutación se realiza de la siguiente forma:

$$\sigma_i' = \sigma_i \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)} \tag{A.12}$$

Donde τ es un parámetro que proporciona el usuario. Sea n el número de variables de decisión, los valores por defecto son $\tau' = \frac{1}{\sqrt{2n}}$ y $\tau = \frac{1}{\sqrt{2\sqrt{n}}}$.

Constructor:

- *decision_variables.* Número de variables de decisión del problema.

Métodos

- ***length.*** Función auxiliar para la clase `EvolutionStrategy` que indica cuántos tamaños de paso debe tener cada individuo. En este caso es un tamaño de paso por cada variable de decisión de cada individuo.

  Argumentos:

  - No recibe ningún argumento.

  Valor de retorno:

  - Número de sigma's empleados para cada individuo de la población (este operador retorna el número de variables de decisión del problema).

- ***__call__.*** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `sigma`. Arreglo bidimensional de *numpy*. Cada fila contiene los valores σ_i de uno de los individuos de la población.

  Valor de retorno:

  - Arreglo *numpy* con los valores mutados σ_i'.

Operadores de cruza

discrete

Operador de cruza discreta. Sean $\boldsymbol{x}$ y $\boldsymbol{y}$ los padres 1 y 2, respectivamente. El nuevo individuo $\boldsymbol{z}$ está dado por:

$$z_i = \begin{cases} x_i & \text{Si } b_i = 1, \\ y_i & \text{Si } b_i = 0 \end{cases} \tag{A.13}$$

Donde $\boldsymbol{b}$ es un vector con valores aleatorios binarios del tamaño de las variables de decisión del problema y $\boldsymbol{z}$ es el individuo generado por la cruza.
Constructor:

- No recibe ningún argumento.

Métodos

- ***__call__*.** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `population`. Arreglo bidimensional de *numpy*. Cada fila es un individuo de la población actual.
  - `parent_ind1`. Arreglo de *numpy* que contiene los índices de los individuos seleccionados de `population` que actuarán como padre 1.
  - `parent_ind2`. Arreglo de *numpy* que contiene los índices de los individuos seleccionados de `population` que actuarán como padre 2.

  Valor de retorno:

  - Arreglo bidimensional de *numpy*. Cada fila es un individuo generado por la cruza.

intermediate

Operador de cruza intermedia. Sean $\boldsymbol{x}$ y $\boldsymbol{y}$ los padres 1 y 2, respectivamente, el nuevo individuo $\boldsymbol{z}$ está dado por:

$$z_i = \alpha \cdot x_i + (1 - \alpha) \cdot y_i \tag{A.14}$$

Donde α es un parámetro proporcionado por el usuario.
Constructor:

- `Alpha`. Proporción en la que contribuye cada padre para generar al nuevo individuo. Por defecto es 0.5.

Métodos

__call__. Este método nos permite hacer que nuestra clase se comporte como una función.
Argumentos:

- `population`. Arreglo bidimensional de numpy. Cada fila es un individuo de la población actual.
- `parent_ind1`. Arreglo de numpy que contiene los índices de los individuos seleccionados de population que actuarán como padre 1.
- `parent_ind2`. Arreglo de numpy que contiene los índices de los individuos seleccionados de population que actuarán como padre 2.

Valor de retorno:

- Arreglo bidimensional de *numpy*. Cada fila es un individuo generado por la cruza.

Operadores de selección de sobrevivientes

merge_selector.

Esquema $(\mu + \lambda)$, selecciona μ individuos que son obtenidos al unir la población de hijos y la población actual. Los individuos que permanecerán en la próxima generación son aquellos que tengan un mejor valor de aptitud.
Constructor:

- No recibe ningún parámetro.

Métodos

- ***__call__.*** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `parent_f`. Arreglo de *numpy* que contiene la aptitud de cada individuo de la población actual.
  - `offspring_f`. Arreglo de *numpy* que contiene la aptitud de cada individuo de la población de hijos.
  - `features`. Diccionario que almacena la información de las poblaciones. Cada llave del diccionario almacenará una lista con dos componentes, donde, la primera componente es la información de la población actual `parent_population` y la segunda componente es la información de la población de hijos `offspring_population`.

  Valor de retorno:

  - Diccionario con los individuos seleccionados por dicho esquema. Las llaves de este diccionario serán las mismas llaves recibidas en el parámetro `features` y adicional otra llave con el nombre `parent_population_f`, donde, ahora sólo contendrá la información de los individuos que pasarán a la próxima generación.

replacement_selector.

Esquema (μ, λ), reemplaza la población actual con los μ mejores hijos de acuerdo a su valor de aptitud.
Constructor:

- No recibe ningún parámetro.

Métodos

- ***__call__***. Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `parent_fitness`. Arreglo de *numpy* que contiene la aptitud de cada individuo de la población actual.
  - `offspring_fitness`. Arreglo de *numpy* que contiene la aptitud de cada individuo de la población de hijos.
  - `features`. Diccionario que almacena la información de las poblaciones. La primera componente es la información de la población actual `parent_population` y la segunda componente es la información de la población de hijos `offspring_population`.

  Valor de retorno:

  - Diccionario con los individuos seleccionados por dicho esquema. Las llaves de este diccionario serán las mismas llaves recibidas en el parámetro `features` y adicional otra llave con el nombre `parent_population_f`, donde, ahora sólo contendrá la información de los individuos que pasarán a la próxima generación.

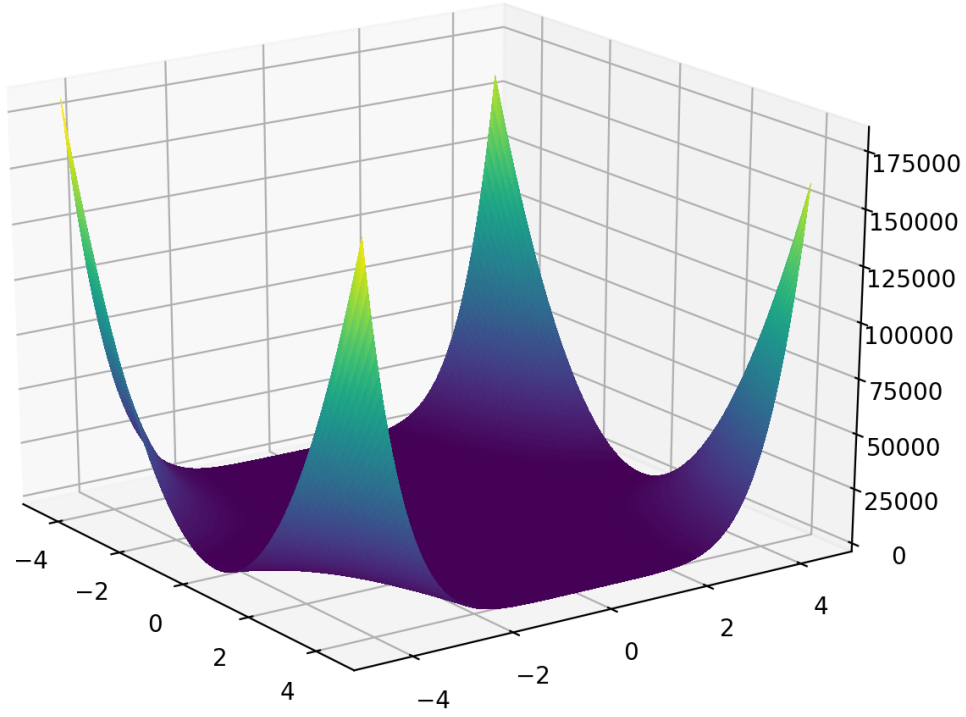
A.4.4. Función de Beale

$$\begin{aligned} \text{minimizar:} \quad & f(x_1, x_2) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + \\ & (2.625 - x_1 + x_1 x_2^3)^2 \\ \text{tal que:} \quad & -4.5 \leq x_1, x_2 \leq 4.5 \end{aligned} \tag{A.15}$$

El mínimo global se encuentra en $x^* = (3, 0.5)$ y $f(x^*) = 0$.

```
In [4]: Image(filename="include/beale.png", width=500, height=300)
```

```
Out[4]:
```



La librería **Pyristic** tiene implementados algunos problemas de prueba en **utils.helpers.test_function**, entre ellos la función de Beale. Los problemas de prueba están definidos como diccionarios con las siguientes llaves:

- `function`. Función objetivo.
- `constraints`. Lista con las funciones que validan las diferentes restricciones del problema. Todas las funciones deben regresar un valor booleano.
- `bounds`. Límites para cada una de las variables del problema. En el caso de que todas las variables del problema compartan el mismo intervalo de búsqueda, se utiliza una lista con dos valores numéricos. El primero indica el límite inferior y el segundo el límite superior. En caso contrario, se tendrá una lista con dos listas internas. La primera lista almacena los límites inferiores y la segunda los límites superiores.
- `decision_variables`. Número de variables de decisión.

```
In [5]: beale_
```

```
Out[5]: {'function': CPUDispatcher(<function beale_function at 0x7f4b5862a268>),
         'constraints': [CPUDispatcher(<function constraint1_beale at 0x7f4b50b47400>
         'bounds': [-4.5, 4.5],
         'decision_variables': 2}
```

Función de aptitud

Los algoritmos evolutivos, se define una función llamada **aptitud** que describe a cada individuo que tan bueno es en el problema a resolver. Los valores más grandes de aptitud corresponden a las mejores soluciones. Para el problema de Beale, la función de aptitud es: $F_a(i) = \frac{1}{f(x_1^{(i)}, x_2^{(i)})+1}$.

```
In [5]: def aptitudeBeale(X: np.array) -> np.array:
            return 1/ ( beale_['function'](X)+1)
```

Declaración de `EvolutionStrategy`

La metaheurística de EE implementada en la librería de **Pyristic** se puede utilizar de las siguientes maneras:

- Crear una clase que herede de la clase `EvolutionStrategy` y sobreescribir alguno de los métodos.
- Declarar un objeto de tipo *EvolutionStrategyConfig*, ubicado en *utils.helpers*, y utilizarlo para inicializar nuestro algoritmo de optimización de tipo `EvolutionStrategy`.
- Realizar una combinación de las dos anteriores.

Es importante resaltar que se puede hacer uso de la metaheurística sin modificar los operadores que tiene por defecto.

A.4.5. Ejecución de la metaheurística

Como mencionamos antes, una forma de utilizar la metaheurística es hacer una instancia de la clase `EvolutionStrategy` dejando su configuración por defecto.
Los argumentos que se deben indicar al momento de inicializar son:

- Función objetivo.
- Restricciones del problema.
- Límite inferior y superior (por cada variable de decisión).
- Número de variables que tiene el problema.

```
In [6]: Beale = EvolutionStrategy(
                                function= aptitudeBeale,\
                                decision_variables=beale_['decision_variables'],\
                                constraints= beale_['constraints'],\
                                bounds= beale_['bounds'])
```

Recordemos que `beale_` es un diccionario con la información requerida por el constructor de la clase `EvolutionStrategy`.

$(\mu + \lambda)$ - EE

Por defecto, la metaheurística trabaja de la siguiente forma:

- Selección de sobrevivientes utiliza un esquema $(\mu + \lambda)$.
- El operador de cruza para las variables de decisión es *recombinación discreta* (operador llamado `discrete`).
- El operador de cruza para los tamaños de paso es *recombinación intermedia* (operador llamado `intermediate`).
- Cada individuo tiene un tamaño de paso por variable (operador llamado `mult_sigma_adaptive_mutator`).

Una vez creado el objeto, se manda a llamar a su método `optimize` el cual recibe los siguientes parámetros:

- **generations** = 300
- **population_size** = 80
- **offspring_size** = 160

```
In [7]: Beale.optimize(300,80,160,verbose=True)

100% 300/300 [00:02<00:00, 137.55it/s]


In [8]: print(Beale)

Evolution Strategy search:
 f(X) = 3.611503831714945e-30
 X = [3.  0.5]
 Constraints:
 x1: -4.5 <= 3.00 <= 4.5
 x2: -4.5 <= 0.50 <= 4.5
```

Para realizar un estudio estadístico del comportamiento de la metaheurística, la librería **Pyristic** cuenta con una función llamada `get_stats`. Esta función se encuentra en `utils.helpers` y recibe como parámetros:

- Objeto que realiza la ejecución de la metaheurística.
- Número de veces que se quiere ejecutar la metaheurística.
- Tupla con los argumentos que recibe la función `optimize`.
- Argumentos adicionales a la búsqueda (opcional).

La función **get_stats** considera la solución devuelta por la metaheurística en cada ejecución y retorna un diccionario con la mejor y peor solución encontrada y la media y desviación estándar del valor de la función objetivo.

```
In [9]: args = (200, 80, 160, 0.0001, False)
        statistics = get_stats(Beale, 21, args, transformer=beale_['function'])

In [10]: pprint(statistics)

{'Best solution': {'f': 7.65172802912719e-24, 'x': array([3. , 0.5])},
 'Mean': 4.2654367173737946e-13,
 'Median': 1.1990144937278418e-19,
 'Standard deviation': 1.9074148651240894e-12,
 'Worst solution': {'f': 8.956762249866645e-12,
                    'x': array([3.00000465, 0.50000164])}}
```

(1+1) - EE

A continuación vamos a implementar la EE más simple conocida como $(1+1)-$EE para resolver la función de Beale. $(1+1)-$EE trabaja de la siguiente forma:

1. Utiliza un único valor de σ para mutar todas las variables de decisión.
2. El valor de σ se autoadapta utilizando la regla del éxito del $1/5$.

$$\sigma = \begin{cases} \sigma/c & \text{Si } p_s > 1/5, \\ \sigma \cdot c & \text{Si } p_s < 1/5, \\ \sigma & \text{Si } p_s = 1/5 \end{cases} \tag{A.16}$$

3. Se utiliza un solo individuo, éste es mutado para generar un hijo.

$$x_i' = x_i + \sigma_i' \cdot N_i(0,1) \tag{A.17}$$

4. El individuo más apto es el que pasará a la siguiente generación.

Esta versión no considera un operador de cruza. Por este motivo, vamos a sobreescribir `crossover_operator` y `adaptive_crossover` para que retornen el individuo original sin cambio alguno. Las funciones que debemos sobreescribir son:

- crossover_operator
- adaptive_crossover
- adaptive_mutation
- mutation_operator

- initializing_step_weights

```
In [11]: class ESBasic(EvolutionStrategy):
             def __init__(self, function  ,\
                          decision_variables:int,\
                          constraints: list,\
                          bounds: np.ndarray):
                 super().__init__(function, decision_variables,\
                                  constraints, bounds)

                 self.successful = 0


             def crossover_operator(self, parent_ind1,\
                                    parent_ind2,\
                                    **kargs):
                 return self.logger['parent_population_x']

             def adaptive_crossover(self, parent_ind1,\
                                    parent_ind2,\
                                    **kargs):
                 return self.logger['parent_population_sigma']

             def adaptive_mutation(self, **kargs):

                     if self.logger['current_iter'] % kargs['k'] != 0:
                         return self.logger['offspring_population_sigma']

                     ps = self.successful / kargs['k']
                     self.successful = 0
                     if( ps > 1/5):
                         return self.logger['offspring_population_sigma']/kargs['c']
                     elif( ps < 1/5):
                         return self.logger['offspring_population_sigma']*kargs['c']
                     else:
                         return self.logger['offspring_population_sigma']

             def mutation_operator(self, **kargs):
                 X_mutated = copy.deepcopy(self.logger['offspring_population_x'])
                 X_mutated += self.logger['offspring_population_sigma'] *
                              np.random.normal(0,1, size=X_mutated.shape)

                 if(
                    self.f(self.logger['offspring_population_x'][0]) >
```

```
                    self.f(X_mutated[0])):
                     self.successful+=1

                return X_mutated

            def initializing_step_weights(self, eps_sigma:int, **kwargs):
                return np.random.uniform(0,1, size=(
                    self.logger['parent_population_size'], 1))
```

```
In [12]: Beale_basic = ESBasic(
                              function= aptitudeBeale,\
                              decision_variables=beale_['decision_variables'],\
                              constraints= beale_['constraints'],\
                              bounds= beale_['bounds'])
```

Como se puede observar, estamos haciendo uso del argumento llamado `kargs`, el cual es un diccionario que contiene información adicional. En este caso `kargs` contiene dos constantes: la primera es el parámetro k que indica cada cuantas generaciones se va a actualizar el valor de σ y la segunda es el valor del parámetro c que se utiliza para actualizar el valor de σ. El diccionario tiene que ser incluido siempre con ** antes del nombre de la variable que lo almacena.

```
In [13]: additional_arguments = {'k':20,'c':0.83}
```

```
In [14]: Beale_basic.optimize(generations=200,\
                               population_size=1,\
                               offspring_size=1,\
                               **additional_arguments)
```

```
100% 200/200 [00:00<00:00, 4906.57it/s]
```

```
In [15]: print(Beale_basic)
```

```
Evolution Strategy search:
 f(X) = 2.4808725619230687
 X = [-0.36423684  2.1653851 ]
 Constraints:
 x1: -4.5 <= -0.36 <= 4.5
 x2: -4.5 <= 2.17 <= 4.5
```

```
In [16]: args = (500, 1, 1, 0.001,False)
         statistics = get_stats(
                Beale_basic,
                30, args,
                additional_arguments,
                aptitudeBeale)
```

```
In [17]: pprint(statistics)
```

```
{'Best solution': {'f': 5.095899477128232e-05,
                   'x': array([2.99200251, 0.49667427])},
 'Mean': 1.2601353824910624,
 'Median': 0.00736331186772052,
 'Standard deviation': 2.8115480703491436,
 'Worst solution': {'f': 9.718812230380905,
                    'x': array([ 0.04801314, -3.5862719 ])}}
```

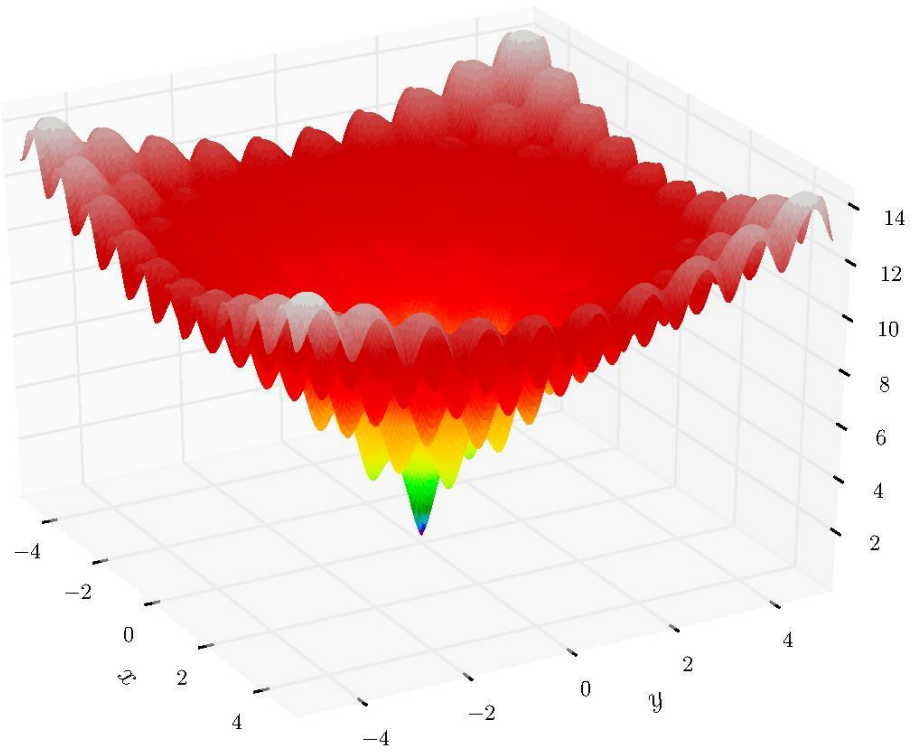
A.4.6. Función de Ackley

$$\text{mín } f(\boldsymbol{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n}x_i^2}\right) - exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e \tag{A.18}$$

El mínimo global está en $x^* = 0$, $f(\boldsymbol{x}) = 0$ y su dominio es $|x_i| < 30$.

```
In [18]: Image(filename="include/ackley.jpg", width=500, height=300)
```

```
Out[18]:
```



Al igual que la función de Beale, la librería **Pyristic** tiene implementada la función de Ackley en `utils.helpers.test_function`.

```
In [19]: ackley_
```

```
Out[19]: {'function': CPUDispatcher(<function ackley_function at 0x7f4b3c06e2f0>),
          'constraints': [CPUDispatcher(<function constraint1_ackley at 0x7f4b3c06e59
          'bounds': [-30.0, 30.0],
          'decision_variables': 10}
```

Función de aptitud

Los algoritmos evolutivos, se define una función llamada **aptitud** que describe a cada individuo que tan bueno es en el problema a resolver. Los valores más grandes de aptitud corresponden a las mejores soluciones. Para el problema de Ackley, la función de aptitud es: $F_a(\boldsymbol{X}) = -f(\boldsymbol{X})$.

```
In [20]: def aptitudeAckley(x):
             return -1* ackley_['function'](x)
```

Este problema no se encuentra restringido a un número de variables de decisión. Para modificar el número de variables de decisión hacemos:

```
ackley_['decision_variables'] = 5
```

($\mu + \lambda$)-EE

Emplearemos la metaheurística de EE, con la configuración por defecto, para resolver la función de Ackley con 10 variables de decisión.

```
In [20]: Ackley = EvolutionStrategy(**ackley_)
```

```
In [21]: '''
         Número de generaciones: 250
         Tamaño de población de padres: 100
         Tamaño de población de hijos: 200
         '''
         Ackley.optimize(250,100,200)
```

```
100% 250/250 [00:01<00:00, 149.82it/s]
```

```
In [22]: print(Ackley)
```

```
Evolution Strategy search:
 f(X) = 0.19851402189488043
 X = [ 0.06903314 -0.02719133 -0.01734121  0.0083857  -0.02041749  0.02305745
  0.04877207 -0.03463208 -0.03300106  0.01635406]
 Constraints:
 x1: -30 <= 0.07 <= 30
 x2: -30 <= -0.03 <= 30
 x3: -30 <= -0.02 <= 30
 x4: -30 <= 0.01 <= 30
 x5: -30 <= -0.02 <= 30
 x6: -30 <= 0.02 <= 30
 x7: -30 <= 0.05 <= 30
 x8: -30 <= -0.03 <= 30
 x9: -30 <= -0.03 <= 30
 x10: -30 <= 0.02 <= 30
```

```
In [23]: args = (250, 100, 200, 0.001 ,False)
         statistics = get_stats(Ackley, 30, args, transformer=ackley_['function'])
```

```
In [24]: pprint(statistics)

{'Best solution': {'f': 0.008958576078040625,
                   'x': array([ 0.00155744, -0.00045284,
                                0.00207098, -0.00060489,
                                0.00262941, 0.0008557 ,
                                -0.00080341,  0.00099172,
                                -0.00170624,  0.00528233])},
 'Mean': 0.08126613753517124,
 'Median': 0.04674853757252473,
 'Standard deviation': 0.07947966129764066,
 'Worst solution': {'f': 0.3279831631361394,
                    'x': array([-0.03177579,  0.01487862,
                                -0.03282081,  0.02822527,
                                -0.07654211,-0.03552047,
                                -0.01205284,  0.02290453,
                                0.11966758, -0.00441367])}}
```

(μ, λ)-EE

Ahora vamos a utilizar una configuración del tipo `EvolutionStrategyConfig` para definir los métodos que serán empleados en nuestro objeto del tipo `EvolutionStrategy`.

```
In [25]: configuration_ackley = (EvolutionStrategyConfig()
                                 .survivor_selection(
                                     selection.replacement_selector())
                                 .adaptive_mutation(
                                     mutation.single_sigma_adaptive_mutator(
                                             ackley_['decision_variables'])
                                                   )
                                 )
```

En este caso estamos modificando el esquema de selección de sobrevivientes y emplearemos un único tamaño de paso para cada individuo, donde, el argumento que tiene `single_sigma_adaptive_mutator` es el número de variables de decisión del problema.

```
In [26]: print(configuration_ackley)

----------------------------------------
        Configuration
----------------------------------------
Survivor selection: Replacement population
Adaptive mutation: Single Sigma
```

```
----------------------------------------


In [27]: solver_ackley_custom = EvolutionStrategy(
                        function= aptitudeAckley,\
                        decision_variables=ackley_['decision_variables'],\
                        constraints= ackley_['constraints'],\
                        bounds= ackley_['bounds'],config=configuration_ackley)

In [28]: solver_ackley_custom.optimize(250,100,200)

100% 250/250 [00:01<00:00, 164.92it/s]


In [29]: print(solver_ackley_custom)

Evolution Strategy search:
 f(X) = 6.722875353277301
 X = [-2.40256206  2.64503024  0.88760834 -2.27760043  1.01264436 -0.20435495
 -0.2826214  -0.1739199  -1.97614466  0.00283308]
 Constraints:
 x1: -30 <= -2.40 <= 30
 x2: -30 <= 2.65 <= 30
 x3: -30 <= 0.89 <= 30
 x4: -30 <= -2.28 <= 30
 x5: -30 <= 1.01 <= 30
 x6: -30 <= -0.20 <= 30
 x7: -30 <= -0.28 <= 30
 x8: -30 <= -0.17 <= 30
 x9: -30 <= -1.98 <= 30
 x10: -30 <= 0.00 <= 30




In [30]: args = (250, 100, 200,0.001 ,False)
         statistics = get_stats(solver_ackley_custom,
                        30, args,
                        transformer=ackley_['function'])

In [31]: pprint(statistics)

{'Best solution': {'f': 5.4117166569087765,
                   'x': array([ 0.267664  ,  1.44280275,
```

```
                                -0.64901279,  1.25632154,
                                0.84707397, -0.58992425,
                                0.28989363, -0.87462408,
                                -1.73511192, -0.99452029])},
 'Mean': 17.879391263858643,
 'Median': 19.950424956466673,
 'Standard deviation': 5.023293120504029,
 'Worst solution': {'f': 22.283488736223426,
                    'x': array([-28.50118574, -28.50118574,
                                -28.50118574, -28.50118574,
                                -28.50118574, -28.50118574,
                                -28.50118574, -28.50118574])}}
```

A.4.7. (1+1)-EE

A continación usaremos la EE más simple para intentar resolver la función de Ackley con 10 variables de decisión.

```
In [34]: Ackley_basic = ESBasic(
                              function= aptitudeAckley,\
                              decision_variables=ackley_['decision_variables'],\
                              constraints= ackley_['constraints'],\
                              bounds= ackley_['bounds'])

In [33]: Ackley_basic.optimize(10000,1,1,**additional_arguments )

100% 10000/10000 [00:00<00:00, 11337.41it/s]


In [34]: print(Ackley_basic)

Evolution Strategy search:
 f(X) = 19.113500402376996
 X = [-14.04563647  -9.10940732  -6.95817682   8.01075501 -23.01714587
  -7.99060593  -2.84359182 -12.17720417 -25.98627065   0.17446611]
 Constraints:
 x1: -30 <= -14.05 <= 30
 x2: -30 <= -9.11 <= 30
 x3: -30 <= -6.96 <= 30
 x4: -30 <= 8.01 <= 30
 x5: -30 <= -23.02 <= 30
 x6: -30 <= -7.99 <= 30
```

```
 x7: -30 <= -2.84 <= 30
 x8: -30 <= -12.18 <= 30
 x9: -30 <= -25.99 <= 30
 x10: -30 <= 0.17 <= 30
```

```
In [35]: args = (10000, 1, 1, 0.001,False)
         statistics = get_stats(Ackley_basic, 30, args,
                                **additional_arguments )

In [36]: pprint(statistics)

{'Best solution': {'f': 16.959169469710393,
                   'x': array([  8.9276951 ,  15.93280431,
                                -1.93419654,  -0.03219489,
                                3.96515827,   6.89752938,
                                -5.06776863, -11.02911313,
                                14.8878337 ,   7.03884713])},
 'Mean': 19.54747426262706,
 'Median': 19.691584669060312,
 'Standard deviation': 0.6647809401288951,
 'Worst solution': {'f': 20.235327209307368,
                    'x': array([ 2.18718978e+01, -6.09861924e+00,
                                -2.99300634e+01, -8.00091391e+00,
                                1.50159338e+01,  2.60925239e+01,
                                -2.93202750e-02, -2.30262787e+01,
                                1.83874769e+01,  2.50573755e+01])}}
```

A.5. Algoritmos genéticos

La librería **Pyristic** incluye una clase llamada `Genetic` inspirada en la metaheurística de *Algoritmos genéticos* (AG) para resolver problemas de minimización. Para trabajar con esta clase se requiere hacer lo siguiente:

1. Definir:
    - La función objetivo f.
    - La lista de restricciones.
    - Lista de límites inferiores y límites superiores.
    - Configuración de operadores de la metaheurística.
2. Crear una clase que hereda de `Genetic`.
3. sobrescritura de funciones auxiliares:
    - initialize_population (opcional)
    - fixer (opcional)
    - mutation_operator (opcional)
    - crossover_operator (opcional)
    - survivor_selection (opcional)
    - parent_selection (opcional)

A continuación se mostrarán los elementos que se deben importar.

Librerías externas

```
In [2]: from pprint import pprint
        import math
        import random
        import numpy as np
        import copy
```

```
In [3]: from IPython.display import Image
        from IPython.core.display import HTML
```

Componentes de `pyristic`

La estructura que está organizada la librería es:

- Las metaheurísticas están ubicadas en `heuristic`.
- Las funciones de prueba están ubicadas en `utils.test_function`.

- Las clases auxiliares para mantener la información de los operadores que serán empleados para alguna de las metaheurísticas basadas en los paradigmas del cómputo evolutivo están ubicadas en `utils.helpers`.
- Las metaheurísticas basadas en los paradigmas del cómputo evolutivo dependen de un conjunto de operadores (selección, mutación y cruza). Estos operadores están ubicados en `utils.operators`.

Para demostrar el uso de nuestra metaheurística basada en *algoritmos geneticos* tenemos que importar la clase llamada `Genetic` que se encuentra en `heuristic.GeneticAlgorithm_search`.

```
In [4]: from pyristic.heuristic.GeneticAlgorithm_search import Genetic
        from pyristic.utils.operators import selection,mutation,crossover

In [5]: import pyristic.utils.helpers as helpers
        from pyristic.utils.test_function import ackley_, beale_
```

A.5.1. Clase `Genetic`

Variables

- ***logger.*** Diccionario con información relacionada a la búsqueda.
  - `best_individual`. Mejor individuo encontrado.
  - `best_f`. Aptitud del mejor individuo.
  - `current_iter`. Iteración actual de la búsqueda.
  - `total_iter`. Número total de iteraciones.
  - `population_size`. Tamaño de la población.
  - `parent_population_x`. Arreglo bidimensional de numpy. Cada fila representa a un individuo de la población actual y cada columna corresponde a una variable de decisión.
  - `offspring_population_x`. Arreglo bidimensional de numpy. Cada fila representa a un individuo de la población de hijos y cada columna corresponde a una variable de decisión.
  - `parent_population_f`. Arreglo de numpy que contiene el valor de la función objetivo para cada uno de los individuos de la población de `parent_population_x`.
  - `offspring_population_f`. Arreglo de numpy que contiene el valor de la función objetivo para cada uno de los individuos de la población de `offspring_population_x`.
- ***f.*** Función objetivo.
- ***Constraints.*** Lista de restricciones del problema. Las restricciones deben

ser funciones que retornan True o False, indicando si cumple dicha restricción.

- ***Bounds.*** Representa los límites definidos para cada una de las variables del problema. Se aceptan las siguientes representaciones:
  - Arreglo de *numpy* con solo dos componentes numéricas, donde, la primera componente es el límite inferior y la segunda componente es el límite superior. Esto significa que todas las variables de decisión estarán definidas para el mismo intervalo.
  - Arreglo bidimensional de *numpy* con dos arreglos de numpy, donde, el primer arreglo de numpy representa el límite inferior para cada variable de decisión, mientras, la segunda componente representa el límite superior para cada variable de decisión.
- ***Decision_variables.*** El número de variables de decisión del problema.

Métodos

- **_init_** Constructor de la clase.
  Argumentos:
  - `function`. Función objetivo.
  - `decision_variables`. Número de variables de decisión.
  - `constraints`. Lista con las restricciones del problema. Las restricciones deben ser funciones que retornan True o False, dependiendo de si se cumple o no dicha restricción.
  - `bounds`. Límites de las variables de decisión (se describe los tipos de datos admisibles en el apartado de variables de la clase con el nombre Bounds).
  - `config`. Estructura de datos (`GeneticConfig`) con los operadores que se emplearán en la búsqueda.

  Valor de retorno:
  - Ninguno.
- ***optimize.*** Método principal, realiza la ejecución de la metaheurística.
  Argumentos:
  - `generations`. Número de generaciones (iteraciones de la metaheurística).
  - `size_population`. Tamaño de la población (número de individuos).
  - `verbose`. Indica si se imprime en qué iteración se encuentra nuestra búsqueda. Por defecto, está en True.

- `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobrescribe alguno de los métodos que tiene la clase.

Valor de retorno:

- Ninguno.

- ***fixer.*** Si nuestro individuo al ser evaluado no cumple las restricciones del problema, esta función auxiliar actualizará nuestro individuo de modo que sea válida. La función auxiliar es necesario definirla (no tiene ninguna por defecto), se especifica en la configuración (GeneticConfig) o sobrescribiendo dicha función.
  Argumentos:
  - `ind`. Índice del individuo.

  Valor de retorno:
  - Un arreglo de *numpy* con la solución factible.

- ***initialize_population.*** Crea una población de individuos aleatorios. Para ello se utiliza una distribución uniforme y se generan números aleatorios dentro de los límites indicados para cada variable. Los individuos generados son almacenados en `logger` con la llave `parent_population_x`. Esta función es llamada dentro de la funcion `optimize`.
  Argumentos:
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobrescribe alguno de los métodos que tiene la clase.

  Valor de retorno:
  - Un arreglo bidimensional de *numpy*. Cada fila representa un individuo, cada columna indica los valores para cada variable de decisión del individuo.

- ***mutation_operator.*** Muta las variables de decisión de la población de hijos, las cuales se encuentran almacenadas en el diccionario `logger` con la llave `offspring_population_x`. El operador de mutación es necesario definirlo (no tiene ningún operador por defecto), se especifica en la configuración (`GeneticConfig`) o sobrescribiendo el operador.
  Argumentos:
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobrescribe alguno de los métodos que tiene la clase.

  Valor de retorno:

  - Un arreglo bidimensional de *numpy* con la población mutada. Cada fila representa un individuo y cada columna corresponde a una variable de decisión.

- ***crossover_operator.*** Dada una población de padres, genera una población de hijos aplicando algún tipo de cruza. El operador de cruza es necesario definirlo (no tiene ningún operador por defecto), se especifica en la configuración (`GeneticConfig`) o sobrescribiendo el operador.
  Argumentos:
  - `parent_ind1`. Índices de los individuos que son seleccionados para actuar como padre 1.
  - `parent_ind2`. Índices de los individuos que son seleccionados para actuar como padre 2.
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobrescribe alguno de los métodos que tiene la clase.

  Valor de retorno:
  - Un arreglo bidimensional de *numpy* con la población generada por la cruza. Cada fila representa un individuo y cada columna corresponde a una variable de decisión.

- ***parent_selection.*** Selecciona a los individuos que actuarán como padres. El método de selección es necesario definirlo (no tiene ningún operador por defecto), se especifica en la configuración (`GeneticConfig`) o sobrescribiendo el método.
  Argumentos:
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobrescribe alguno de los métodos que tiene la clase.

  Valor de retorno:
  - Regresa un arreglo con los índices de los individuos seleccionados. Estos índices corresponden con la variable `parent_population_x` que está almacenada en el diccionario `logger`.

- ***survivor_selection.*** Selección de los individuos que pasarán a la siguiente generación. El método de selección de sobrevivientes es necesario definirlo (no tiene ningún operador por defecto), se especifica en la configuración (`GeneticConfig`) o sobrescribiendo el método.
  Argumentos:
  - `**kwargs`. Diccionario con argumentos externos a la búsqueda. Estos argumentos pueden ser empleados cuando se sobrescribe alguno de los métodos que tiene la clase.

Valor de retorno:

- Un diccionario con las siguientes llaves:
  - `parent_population_f`. Arreglo de *numpy* con la aptitud de cada uno de los individuos que pasará a la siguiente generación.
  - `parent_population_x`. Arreglo bidimensional de *numpy* con los individuos que pasarán a la siguiente generación.

A.5.2. Clase `GeneticConfig`

Variables

- ***cross_op.*** Variable con el operador de cruza.
- ***mutation_op.*** Variable con el operador de mutación.
- ***survivor_selection.*** Variable con el esquema de selección que decide cuáles individuos pasan a la siguiente generación.
- ***fixer.*** Variable con una función que determina qué hacer con los individuos que no cumplen las restricciones del problema.
- ***parent_selector.*** Variable que almacena el operador de selección.

Métodos

- ***cross.*** Actualiza el operador de cruza de la variable `cross_op`. Argumentos:
  - `crossover_`. Función o clase que realiza la cruza de la población almacenada con la llave `parent_population_x`.

  Valor de retorno:
  - Retorna la configuración con la actualización del operador de cruza. El objetivo es poder aplicar varios operadores en cascada.
- ***mutate.*** Actualiza el operador de mutación de la variable `mutation_op`. Argumentos:
  - `mutate_`. Función o clase que realiza la mutación de la población almacenada con la llave `offspring_population_x`.

  Valor de retorno:
  - Retorna la configuración con la actualización del operador de cruza. El objetivo es poder aplicar varios operadores en cascada.
- ***survivor_selection.*** Actualiza el operador de selección de la variable `survivor_selector`. Argumentos:
  - `survivor_function`. Función o clase que realiza la selección de individuos que pasarán a la siguiente generación.

Valor de retorno:

- Retorna la configuración con la actualización del operador de cruza. El objetivo es poder aplicar varios operadores en cascada.

- ***fixer_invalide_solutions.*** Actualiza la función auxiliar de la variable `fixer`. Argumentos:
  - `fixer_function`. Función o clase que ajustará los individuos de la población que no cumplen con las restricciones del problema.

  Valor de retorno:
  - Retorna la configuración con la actualización del operador de cruza. El objetivo es poder aplicar varios operadores en cascada.

- ***parent_selection.*** Actualiza el operador de selección de los individuos con mayores posibilidades de reproducción, se encuentran en la variable `parent_selector`. Argumentos:
  - `parent_function`. Función o clase que elige los individuos de acuerdo a su contribución de aptitud. Este método en la búsqueda es realizado antes de la cruza.

  Valor de retorno:
  - Retorna la configuración con la actualización del operador de cruza. El objetivo es poder aplicar varios operadores en cascada.

A.5.3. Descripción de operadores

Los operadores descritos en Estrategias Evolutivas también pueden ser empleados en AG. Los operadores de selección de sobrevivientes son los mismos para las clases `Genetic`, `EvolutionStrategy` y `EvolutionaryProgramming`.
Los operadores de mutación, cruza y selección con los que cuenta la librería **Pyristic** son clases. La finalidad es unificar el formato de todos los operadores al ser llamados por los métodos de la clase `Genetic`.

Operadores de mutación

insertion_mutator.

Operador empleado para generar permutaciones que selecciona aleatoriamente un elemento de la permutación y una nueva posición. Posteriormente, coloca el elemento en la nueva posición y desplaza el resto de los elementos hacia la derecha. Este proceso se repite n veces por cada individuo. Este operador es conocido como *mutación por desplazamiento* y es una generalización de *mutación*

por inserción.
Constructor:

- `n_elements`. Número de elementos a desplazar, por defecto el número es 1.

Métodos

- ***__call__***. Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `X`. Arreglo bidimensional de *numpy* que representa el conjunto de individuos de la población a mutar. Cada fila es un individuo de la población y cada columna corresponde con una variable de decisión.

  Valor de retorno:

  - Arreglo bidimensional de *numpy* con la población mutada. Cada fila es un individuo de la población y cada columna corresponde con una variable de decisión.

exchange_mutator.

Operador utilizado para permutaciones. Intercambia dos posiciones seleccionadas de manera aleatoria del individuo, las demás posiciones de la permutación permanecen igual.
Constructor:

- Ningún parámetro al inicializar.

Métodos

- ***__call__*.** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `X`. Arreglo bidimensional de *numpy* que representa el conjunto de individuos de la población a mutar. Cada fila es un individuo de la población y cada columna corresponde con una de las variables de decisión.

  Valor de retorno:

  - Arreglo bidimensional de *numpy* con la población mutada. Cada fila es un individuo de la población y cada columna corresponde con una de las variables de decisión.

boundary_mutator.

Operador para representación real conocido como *de límite*. Sean LB y UB los límites inferiores y superiores respectivamente, este operador selecciona una posición aleatoria, i, del vector $\boldsymbol{x}$ y realiza lo siguiente:

$$x'_i = \begin{cases} LB & \text{si } R \leq 0.5 \\ UB & \text{si } R > 0.5 \end{cases} \tag{A.19}$$

Constructor:

- `bounds`. Límites de las variables de decisión del problema. Acepta los siguientes formatos:

  - Arreglo bidimensional de *numpy*. La primera fila contiene los límites inferiores de cada una de las variables de decisión y la segunda fila los límites superiores.
  - Arreglo de *numpy* con dos valores numéricos. El primero es el límite inferior y el segundo es el límite superior. Estos valores serán los límites para todas las variables de decisión del problema.

Métodos

- ***__call__.*** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `X`. Arreglo bidimensional de *numpy* que representa el conjunto de individuos de la población a mutar. Cada fila es un individuo de la población y cada columna corresponde con una de las variables de decisión.

  Valor de retorno:

  - Arreglo bidimensional de *numpy* con la población mutada. Cada fila es un individuo de la población y cada columna corresponde a una de las variables de decisión.

uniform_mutator.

Operador para representación real. Sean LB y UB los límites inferiores y superiores respectivamente, este operador selecciona aleatoriamente una posición i del vector $\boldsymbol{x}$ y realiza lo siguiente:

$$x_i' = rnd(LB, UB) \tag{A.20}$$

Donde, $rnd()$ genera un valor aleatorio utilizando una distribución uniforme.

Constructor:

- `bounds`. Límites de las variables de decisión del problema.
  Acepta los siguientes formatos:
  - Arreglo bidimensional de *numpy* . La primera fila contiene los límites inferiores de cada una de las variables de decisión y la segunda fila los límites superiores.
  - Arreglo de *numpy* con dos valores numéricos. El primero es el límite inferior y el segundo es el límite superior. Estos valores serán los límites para todas las variables de decisión del problema.

Métodos

- ***__call__*.** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `X`. Arreglo bidimensional de *numpy* que representa el conjunto de individuos de la población a mutar. Cada fila es un individuo de la población y cada columna corresponde con una de las variables de decisión.

  Valor de retorno:

  - Arreglo bidimensional de *numpy* con la población mutada. Cada fila es un individuo de la población y cada columna corresponde a una de las variables de decisión.

non_uniform_mutator.

Operador para representación real que selecciona aleatoriamente una posición i del vector $\boldsymbol{x}$ y realiza lo siguiente.

$$\boldsymbol{x}_i' = \boldsymbol{x}_i + N(0, \sigma) \tag{A.21}$$

Donde N genera un valor aleatorio utilizando una distribución normal con media 0 y desviación estándar σ.

Constructor:

- `sigma`. Valor numérico con la desviación estándar que se va a utilizar, por defecto es 1.

Métodos

- ***__call__.*** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `X`. Arreglo bidimensional de *numpy* que representa el conjunto de individuos de la población a mutar. Cada fila es un individuo de la población y cada columna corresponde con una de las variables de decisión.

  Valor de retorno:

  - Arreglo bidimensional de *numpy* con la población mutada. Cada fila es un individuo de la población y cada columna corresponde a una de las variables de decisión.

Operadores de cruza

Estos operadores generan una nueva población de individuos que será almacenada en `logger` con la llave `offspring_population_x`.

n_point_crossover.

Este operador es una generalización de la cruza de un punto. Dado dos padres, se crean dos nuevos individuos. Para ello se seleccionan de manera aleatoria n puntos de cruza. Los nuevos hijos van copiando posición a posición la información de uno de los padres. Cada vez que se encuentra un punto de cruza, intercambian el padre del cual están realizando la copia.

Constructor:

- `n_cross`. Número de puntos de cruza, por defecto es 1.

Métodos

- ***__call__.*** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `X`. Arreglo bidimensional de *numpy* que contiene el conjunto de individuos de la población actual (`parent_population_x`). Cada fila es un individuo de la población y cada columna corresponde a una variable de decisión.
  - `parent_ind1`. Índices de los individuos que son seleccionados para actuar como padre 1.
  - `parent_ind2`. Índices de los individuos que son seleccionados para actuar como padre 2.

  Valor de retorno:

  - Arreglo bidimensional con la población de nuevos individuos. Cada fila es un individuo de la población y cada columna corresponde a una variable de decisión.

uniform_crossover.

Dado dos padres P_1 y P_2, se crean dos nuevos individuos H_1 y H_2 empleando un cambio entre la información proporcionada por el padre que le corresponderá a cada hijo. La información será seleccionada del padre P_i con una probabilidad p_c para el hijo H_i. La cruza se realiza de la siguiente manera:

$$H_{1,i}, H_{2,i} = \begin{cases} P_{1,i}, P_{2,i} & \text{Si } R_i \leq p_c \\ P_{2,i}, P_{1,i} & \text{Si } R_i > p_c \end{cases} \tag{A.22}$$

Donde R es un vector que indica un número aleatorio entre $[0, 1]$.

Constructor:

- `flip_prob`. Probabilidad de que una posición sea considerada como punto de cruza.

Métodos

- ***__call__.*** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `X`. Arreglo bidimensional de *numpy* que contiene el conjunto de individuos de la población actual (`parent_population_x`). Cada fila es un individuo de la población y cada columna corresponde a una variable de decisión.
  - `parent_ind1`. Indices de los individuos que son seleccionados para actuar como padre 1.
  - `parent_ind2`. Indices de los individuos que son seleccionados para actuar como padre 2.

  Valor de retorno:

  - Arreglo bidimensional con la población de nuevos individuos. Cada fila es un individuo de la población y cada columna corresponde a una variable de decisión.

permutation_order_crossover.

Operador empleado para permutaciones. Dado dos padres P_1 y P_2, genera dos nuevos individuos H_1 y H_2. Para el primer hijo H_1, selecciona un segmento aleatorio (longitud variable) del padre P_1, este segmento es copiado a H_1 en las mismas posiciones. Las posiciones restantes son completadas con la información del padre P_2, de izquierda a derecha, sin considerar los elementos que aparecen en el segmento copiado del padre P_1. Para el segundo hijo H_2, se realiza el mismo procedimiento pero intercambiando a los padres.

Constructor:

- Ningún parámetro al inicializar.

Métodos

- ***__call__***. Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `X`. Arreglo bidimensional de *numpy* que contiene el conjunto de individuos de la población actual (`parent_population_x`). Cada fila es un individuo de la población y cada columna corresponde a una variable de decisión.
  - `parent_ind1`. Indices de los individuos que son seleccionados para actuar como padre 1.
  - `parent_ind2`. Indices de los individuos que son seleccionados para actuar como padre 2.

  Valor de retorno:

  - Arreglo bidimensional con la población de nuevos individuos. Cada fila es un individuo de la población y cada columna corresponde a una variable de decisión.

simulated_binary_crossover.

Operador para representación real. Dado dos padres P_1 y P_2, genera dos nuevos individuos H_1 y H_2 de la siguiente forma:

$$H_1 = 0.5[(P_1 + P_2) - \beta|P_2 - P_1|]H_2 = 0.5[(P_1 + P_2) + \beta|P_2 - P_1|] \tag{A.23}$$

Donde β se define como sigue:

$$\beta = \begin{cases} (2u)^{\frac{1}{n_c+1}} & \text{Si } u \leq 0.5, \\ \left(\frac{1}{2(1-u)}\right)^{\frac{1}{n_c+1}} & \text{Si } u > 0.5 \end{cases} \tag{A.24}$$

Regularmente, n_c es igual con 1 ó 2 y $u \in [0, 1]$.

Constructor:

- `n_c`. Parámetro proporcionado por el usuario, por defecto es 1.

Métodos

- ***__call__.*** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `X`. Arreglo bidimensional de *numpy* que contiene el conjunto de individuos de la población actual (`parent_population_x`). Cada fila es un individuo de la población y cada columna corresponde a una variable de decisión.
  - `parent_ind1`. Índices de los individuos que son seleccionados para actuar como padre 1.
  - `parent_ind2`. Índices de los individuos que son seleccionados para actuar como padre 2.

  Valor de retorno:

  - Arreglo bidimensional con la población de nuevos individuos. Cada fila es un individuo de la población y cada columna corresponde a una variable de decisión.

Operadores de selección de padres

Estos operadores están encargados de seleccionar a los individuos que actuarán como padres en el proceso de cruza.

roulette_sampler.

Operador de selección proporcional que simula el comportamiento de una ruleta. La porción de ruleta asignada a cada individuo depende de su valor de aptitud y la aptitud promedio del resto de los individuos.

Constructor:

- Ningún parámetro al inicializar.

Métodos

- ***__call__.*** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `population_f`. Arreglo de *numpy* con valores numéricos que representan los valores obtenidos al evaluar el individuo en la posición i en la función objetivo.

  Valor de retorno:

  - Arreglo de *numpy* con valores enteros en el intervalo $[0, n)$, donde n es el número total de individuos en la población actual. Cada posición del arreglo indica el índice del individuo de la población seleccionado para actuar como padre.

stochastic_universal_sampler.

Método de selección proporcional que garantiza que cada individuo actúe como padre al menos m veces, donde m es la parte entera del valor esperado del individuo. La decisión de que un individuo sea seleccionado $m + 1$ veces, depende de un valor aleatorio.

Constructor:

- Ningún parámetro al inicializar.

Métodos

- ***__call__.*** Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `population_f`. Arreglo de *numpy* con valores numéricos que representan los valores obtenidos al evaluar el individuo en la posición i en la función objetivo.

  Valor de retorno:

  - Arreglo de *numpy* con valores enteros en el intervalo $[0, n)$, donde n es el número total de individuos en la población actual. Cada posición del arreglo indica el índice del individuo de la población seleccionado para actuar como padre.

deterministic_sampler.

Método de selección proporcional que garantiza que cada individuo actúe como padre al menos m veces, donde m es la parte entera del valor esperado del individuo. Para decidir si un individuo actúa como padre $m + 1$ veces, se ordenan a los individuos de acuerdo a la parte decimal de su valor esperado y se van seleccionando a los de mayor valor.

Constructor:

- Ningún parámetro al inicializar.

Métodos

- ***__call__*.** Este método nos permite hacer que nuestra clase se comporte como una función.
  Argumentos:

  - `population_f`. Arreglo de *numpy* con valores numéricos que representan los valores obtenidos al evaluar el individuo en la posición i en la función objetivo.

  Valor de retorno:

  - Arreglo de *numpy* con valores enteros en el intervalo $[0, n)$, donde n es el número total de individuos en la población actual. Cada posición del arreglo indica el índice del individuo de la población seleccionado para actuar como padre.

tournament_sampler.

Este operador crea grupos aleatorios de individuos de tamaño m. En cada grupo, se selecciona al mejor individuo o al peor individuo de acuerdo a su aptitud. La probabilidad de elegir al mejor individuo es p y la probabilidad de elegir al peor individuo es $1 - p$.

Constructor:

- `chunks_`. Tamaño de los grupos, por defecto es 2.
- `prob_`. Probabilidad p con la que se selecciona al mejor individuo.

Métodos

- ***__call__***. Este método nos permite hacer que nuestra clase se comporte como una función.

  Argumentos:

  - `population_f`. Arreglo de *numpy* con valores numéricos que representan los valores obtenidos al evaluar el individuo en la posición i en la función objetivo.

  Valor de retorno:

  - Arreglo de *numpy* con valores enteros en el intervalo $[0, n)$, donde n es el número total de individuos en la población actual. Cada posición del arreglo indica el índice del individuo de la población seleccionado para actuar como padre.

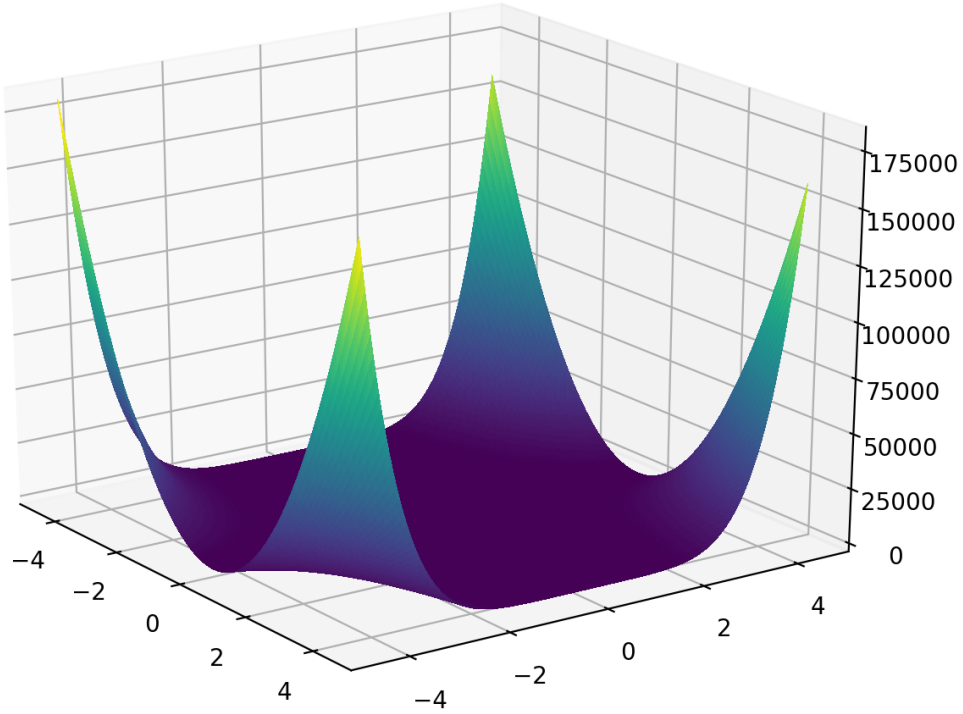
A.5.4. Función de Beale

$$\begin{aligned} \text{minimizar:} \quad & f(x_1, x_2) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + \\ & (2.625 - x_1 + x_1 x_2^3)^2 \\ \text{tal que:} \quad & -4.5 \leq x_1, x_2 \leq 4.5 \end{aligned} \tag{A.25}$$

El mínimo global se encuentra en $x^* = (3, 0.5)$ y $f(x^*) = 0$.

```
In [6]: Image(filename="include/beale.png", width=500, height=300)
```

```
Out[6]:
```



La librería **Pyristic** tiene implementados algunos problemas de prueba en `utils.test_function`, entre ellos la función de Beale. Los problemas de prueba están definidos como diccionarios con las siguientes llaves:

- `function`. Función objetivo.
- `constraints`. Restricciones del problema.
- `bounds`. Límites para cada una de las variables del problema. En el caso de que todas las variables del problema se encuentren en el mismo intervalo de búsqueda, se puede emplear una lista con dos valores numéricos.
- `decision_variables`. Número de variables que tiene dicho problema.

```
In [7]: beale_
```

```
Out[7]: {
'function': CPUDispatcher(<function beale_function at 0x7fdf642ad1e0>),
'constraints': [CPUDispatcher(<function constraint1_beale at 0x7fdf642ada60>)],
'bounds': [-4.5, 4.5],
'decision_variables': 2}
```

Función de aptitud

Los algoritmos evolutivos, se define una función llamada **aptitud** que describe a cada individuo que tan bueno es en el problema a resolver. Los valores más grandes de aptitud corresponden a las mejores soluciones. Para el problema de Beale, la función de aptitud es: $F_a(i) = \frac{1}{f(x_1^{(i)}, x_2^{(i)})+1}$.

```
In [6]: def aptitudeBeale(X: np.array) -> np.array:
            return 1/ ( beale_['function'](X)+1)
```

Declaración de `Genetic`

La metaheurística AG implementada en la librería **Pyristic** se puede utilizar de las siguientes maneras:

- Crear una configuración de los operadores que son almacenados en un objeto del tipo `GeneticConfig` y posteriormente, se declara un objeto de la clase `Genetic`, donde, el constructor recibe la configuración en el parámetro con el nombre `config`.
- Crear una clase que herede de la clase `Genetic` y sobrescribir los métodos.
- Realizar una combinación de las dos anteriores.

La clase `Genetic` no tiene ningún operador definido por defecto, porque, no se tiene conocimiento si el problema a resolver es continuo o discreto.

Ejecución de la metaheurística

Para crear una instancia de la clase `Genetic`, vamos a definir una configuración de los operadores que serán empleados. La configuración se hace a través de la clase `GeneticConfig`.

```
In [8]: configuration_beale = (helpers.GeneticConfig()
            .cross(crossover.intermediate(0.5))
            .mutate(mutation.uniform_mutator(beale_['bounds']))
            .survivor_selection(selection.merge_selector())
            .parent_selection(selection.tournament_sampler(3,0.5))
            .fixer_invalide_solutions(helpers.ContinuosFixer(beale_['bounds'])))

In [9]: print(configuration_beale)

--------------------------------
        Configuration
--------------------------------
Crossover operator: Intermediate
        Arguments:
```

```
                -Alpha:0.5
Mutation operator: Uniform
         Arguments:
                 -Lower bound: -4.5
                 -Upper bound: 4.5
Survivor selection: Merge population
Fixer: continuos
Parent selection: Tournament sampling
         Arguments:
                -Chunks: 3
                -prob: 0.5

--------------------------------
```

En este ejemplo se ha empleado la siguiente configuración:

- `Crossover operator`. Cruza intermedia.
- `Mutation operator`. Mutación uniforme.
- `Survivor selection`. Esquema $(\mu + \lambda)$.
- `Parent selection`. Selección mediante torneos de tamaño 3, utilizando $p = 0.5$.
- `Fixer`. En caso de que haya soluciones infactibles, se utiliza una función auxiliar que actualiza cada variable de decisión que se encuentre fuera del espacio de búsqueda con el valor del límite que rebasó.

A continuación, se inicializa un objeto del tipo `Genetic` con los siguientes parámetros:

- `function`: Función para optimizar.
- `decision_variables`: Número de variables que tiene el problema.
- `constraints`: Restricciones del problema (por defecto, es una lista vacía).
- `bounds`: Límites del problema (por defecto es una lista vacía).
- `config`: Configuración de los operadores (por defecto es None).

```
In [9]: bealeGenetic  = Genetic(
                            function= aptitudeBeale,\
                            decision_variables=beale_['decision_variables'],\
                            constraints= beale_['constraints'],\
                            bounds= beale_['bounds'],\
                            config=configuration_beale
                        )
```

Finalmente, ejecutamos el método `optimize` del objeto `bealeGenetic` usando:

- **generations** = 200.
- **size_population** = 100.
- **verbose** = True.

```
In [11]: bealeGenetic.optimize(200,100)

100% 200/200 [00:01<00:00, 175.62it/s]


In [12]: print(bealeGenetic)

Genetic search:
 f(X) = 0.0001222729774154694
 X = [2.99420989 0.49629651]
 Constraints:
 x1: -4.5 <= 2.99 <= 4.5
 x2: -4.5 <= 0.50 <= 4.5
```

Análisis estadístico

Para revisar el comportamiento de la metaheurística en determinado problema, la librería Pyristic cuenta con una función llamada `get_stats`. Esta función se encuentra en `utils.helpers` y recibe como parámetros:

- Objeto que realiza la búsqueda de soluciones.
- Número de veces que se quiere ejecutar la metaheurística.
- Argumentos que recibe la función optimize (debe ser una tupla).
- Argumentos adicionales a la búsqueda (opcional).

La función `get_stats` retorna un diccionario con algunas estadísticas de las ejecuciones.

```
In [13]: args = (200, 100,False)
         statistics = helpers.get_stats(bealeGenetic, 30, args,{},
             beale_['function'])

In [14]: pprint(statistics)
```

```
{'Best solution': {'f': 3.8528422849262045e-06,
                   'x': array([2.99547576, 0.49871969])},
 'Mean': 0.0002520283655572658,
 'Standard deviation': 0.0005704335576591761,
 'Worst solution': {'f': 0.003221790920680413,
                    'x': array([2.88263129, 0.47541514])}}
```

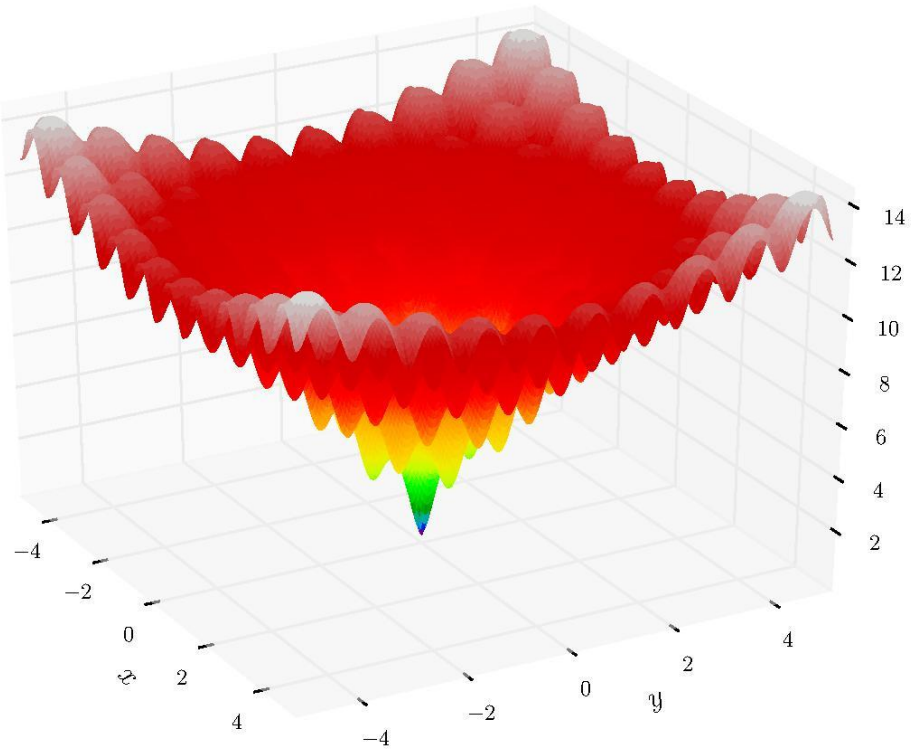
A.5.5. Función de Ackley

$$\text{mín } f(\boldsymbol{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n}x_i^2}\right) - exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e \tag{A.26}$$

El mínimo global está en $x_i^* = 0$, $f(\boldsymbol{x}) = 0$ y su dominio es $|x_i| < 30$.

```
In [15]: Image(filename="include/ackley.jpg", width=500, height=300)
```

Out[15]:



Función de aptitud

Los algoritmos evolutivos, se define una función llamada **aptitud** que describe a cada individuo que tan bueno es en el problema a resolver. Los valores más grandes de aptitud corresponden a las mejores soluciones. Para el problema de Ackley, la función de aptitud es: $F_a(\boldsymbol{X}) = -f(\boldsymbol{X})$.

```
In [15]: def aptitudeAckley(X: np.array) -> np.array:
             return -1*ackley_['function'](X)
```

Para este problema se va a crear una clase llamada *custom_AG* que herede de la clase `Genetic`. Posteriormente, se van a sobrescribir los métodos para que utilicen los siguientes operadores:

- ***parent_selection:*** `tournament_sampler`
- ***survivor_selection:*** `merge_selector`
- ***crossover_operator:*** `intermediate`
- ***mutation_operator:*** `non_uniform_mutator`
- ***fixer:*** `ContinuosFixer`

La siguiente tabla muestra la forma en que se encuentran definidos los operadores en la librería.

Operador	Función	Clase
Mutación	{nombre}_mutation	{nombre}_mutator
Cruza	{nombre}_cross	{nombre}_crossover
Selección de padres	{nombre}_sampling	{nombre}_sampler
Selección de sobrevivientes		{nombre}_selector

Cuando los operadores no requieren de parámetros adicionales o actualizar sus parámetros de los operadores cada que se llama el método `optimize`, se recomienda utilizar los operadores definidos como clases e inicializar la clase `Genetic` con la configuración, como en el ejemplo de la función de Beale. Cuando los operadores requieren de parámetros adicionales, se recomienda usar los operadores definidos como funciones, crear una clase que herede de `Genetic` y sobrescribir los operadores como se muestra a continuación.

```
In [16]: class custom_AG(Genetic):

             def __init__(self, function: helpers.function_type,\
                                decision_variables:int,\
                                constraints:list=[],\
                                bounds: list=[],\
                                config = None):
                 super().__init__(function,\
                           decision_variables,\
                           constraints,\
```

```
                        bounds,\
                        config)
            self._survivor_selection = selection.merge_selector()

        def crossover_operator(self, parent_ind1: np.ndarray,\
                                     parent_ind2: np.ndarray) -> np.ndarray:
            return crossover.intermediate_cross(
                    self.logger['parent_population_x'],\
                    parent_ind1,parent_ind2)

        def mutation_operator(self, **kwargs):
            return mutation.non_uniform_mutation(\
                   self.logger['offspring_population_x'])

        def survivor_selection(self,**kwargs) -> np.ndarray:
            individuals = {}
            individuals['population'] = [
                   self.logger['parent_population_x'],\
                   self.logger['offspring_population_x']]
            return self._survivor_selection(
               self.logger['parent_population_fitness'],\
               self.logger['offspring_population_fitness'],\
                                           individuals)

        def parent_selection(self, **kwargs) -> np.ndarray:
            return selection.tournament_sampling(
                   self.logger['parent_population_fitness'],\
               3,0.5)

        def fixer(self, ind: int)-> np.ndarray:
            return np.clip(
                   self.logger['offspring_population_x'][ind],
                   self.Bounds[0], self.Bounds[1])
```

```
In [16]: AckleyGenetic = custom_AG(
                            function= aptitudeAckley,\
                            decision_variables=ackley_['decision_variables'],\
                            constraints= ackley_['constraints'],\
                            bounds= ackley_['bounds'])
```

Es importante tener en cuenta que se debe incluir `super().__init__()` para llamar al constructor de `Genetic` con los argumentos antes mencionados.

```
In [18]: AckleyGenetic.optimize(200,100)
```

```
100% 200/200 [00:01<00:00, 174.51it/s]
```

```
In [19]: print(AckleyGenetic)
```

```
Genetic search:
 f(X) = 4.440892098500626e-16
 X = [-1.64066073e-24 -3.77045824e-27  9.64238971e-25  2.46046705e-26
  6.09275423e-25  7.77573700e-29  1.16274429e-27 -3.99105718e-28
 -6.46120464e-26  8.52540745e-28]
 Constraints:
 x1: -30 <= -0.00 <= 30
 x2: -30 <= -0.00 <= 30
 x3: -30 <= 0.00 <= 30
 x4: -30 <= 0.00 <= 30
 x5: -30 <= 0.00 <= 30
 x6: -30 <= 0.00 <= 30
 x7: -30 <= 0.00 <= 30
 x8: -30 <= -0.00 <= 30
 x9: -30 <= -0.00 <= 30
 x10: -30 <= 0.00 <= 30
```

```
In [20]: args = (200, 100,False)
         statistics = helpers.get_stats(AckleyGenetic, 30, args,
                     transformer=ackley_['function'])
```

```
In [21]: pprint(statistics)
```

```
{'Best solution': {'f': 4.440892098500626e-16,
                   'x': array([ 9.41657632e-28, -2.68102266e-26,
                                -1.18654066e-25, -6.30912257e-28,
                                7.93873047e-27, -1.27571176e-26,
                                1.87386483e-26,  6.02543005e-28,
                                -1.13657055e-25, -8.15707729e-26])},
 'Mean': 4.440892098500626e-16,
 'Standard deviation': 0.0,
 'Worst solution': {'f': 4.440892098500626e-16,
                    'x': array([ 9.41657632e-28, -2.68102266e-26,
                                -1.18654066e-25, -6.30912257e-28,
                                7.93873047e-27, -1.27571176e-26,
                                1.87386483e-26,  6.02543005e-28,
                                -1.13657055e-25, -8.15707729e-26])}}
```

A.5.6. Problema del agente viajero

$$\begin{aligned} \text{minimizar:} \quad & f(x) = d(x_n, x_1) + \sum_{i=1}^{n-1} d(x_i, x_{i+1}) \\ \text{tal que:} \quad & x_i \in \{1, 2, \cdots, n\} \end{aligned} \tag{A.27}$$

Donde $d(x_i, x_j)$ es la distancia 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. A continuación se define una instancia de este problema utilizando 10 ciudades.

```
In [22]: num_cities = 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],\
         ]
```

`dist_matrix` es una matriz de adyacencia. Cada componente $x_{i,j}$ es la distancia de la ciudad x_i a la ciudad x_j.

Función objetivo

```
In [23]: def f_salesman(x : (list,np.ndarray)) -> float:
             global dist_matrix
             total_dist = dist_matrix[int(x[-1])][0]
             for i in range(1,len(x)):
                 u,v = int(x[i]), int(x[i-1])
                 total_dist+= dist_matrix[u][v]

             return float(total_dist)
```

Función de aptitud

```
In [23]: def aptitudeSalesman(x):
             return -1 * f_salesman(x)
```

```
In [24]: contraints_viajero = []
         bounds_viajero = []
         decision_variables_viajero = 10
```

Dado que para este problema vamos a usar una representación de permutaciones, no es necesario definir los límites de las variables de decisión.

Definición de la configuración

Para este problema se va a utilizar una combinación de las dos formas empleadas anteriormente para hacer uso de la metaheurística de AG. A continuación, definimos la clase `TravellingSalesman_solver` que hereda de `Genetic` y sobrescribimos el método `initialize_population` de tal forma que cada individuo de la población sea una permutación de las ciudades.

```
In [26]: class TravellingSalesman_solver(Genetic):
             def __init__(self,  function: helpers.function_type,\
                                 decision_variables:int,\
                                 constraints:list=[],\
                                 bounds: list=[],\
                                 config = None):
                 super().__init__(
                         function,\
                         decision_variables,\
                         constraints,\
                         bounds,\
                         config)

             def initialize_population(self, **kwargs) -> np.ndarray:
                 individuals = []

                 for i in range(self.logger['population_size']):
                     individuals += [np.random.permutation(range(1,
                            self.Decision_variables))]

                 return np.array(individuals)
```

En este ejemplo se ha empleado la siguiente configuración:

- `Crossover operator`. Permutaciones - Order Crossover (`permutation_order`).
- `Mutation operator`. Mutación para permutaciones: por intercambio recíproco (`exchange_mutator`).
- `Survivor selection`. Esquema $(\mu + \lambda)$.
- `Parent selection`. Selección mediante torneos de tamaño 5, utilizando $p = 0.8$.
- `Fixer`. No se emplea ninguna función auxiliar para corregir las soluciones.

```
In [25]: configuration_Travelling = (helpers.GeneticConfig()
                    .cross(crossover.permutation_order())
                    .mutate(mutation.insertion_mutator(1))
                    .survivor_selection(selection.merge_selector())
                    .parent_selection(selection.tournament_sampler(5,0.8))
                    .fixer_invalide_solutions(helpers.NoneFixer()))
```

Finalmente, se hace una instancia de la clase `TravellingSalesman_solver` y se llama a su método `optimize`.

```
In [27]: travellerGenetic = TravellingSalesman_solver(
                                aptitudeSalesman,\
                                decision_variables_viajero,\
                                contraints_viajero,bounds_viajero,\
                                configuration_Travelling)
```

```
In [28]: travellerGenetic.optimize(200,100)

100% 200/200 [00:04<00:00, 45.77it/s]
```

```
In [29]: print(travellerGenetic)

Genetic search:
 f(X) = 248.0
 X = []
```

Análisis estadístico

```
In [49]: args = (200, 100,False)
         statistics = helpers.get_stats(travellerGenetic, 30, args,
                        transformer=f_salesman)
```

```
In [31]: pprint(statistics)

{'Best solution': {'f': 248.0,
                   'x': array([])},
 'Mean': 248.0,
 'Standard deviation': 0.0,
 'Worst solution': {'f': 248.0,
                    'x': array([])}}
```

Apéndice B

Interfaz gráfica

B.1. PyristicLab

> "By giving people the power to share, we're making the world more transparent."
>
> *Mark Zuckerberg*

Al inicio de este trabajo se describió el modo de empleo de la librería *pyristic*, donde, hemos mostrado la forma de abordar algunos problemas combinatorios y continuos. En este capítulo nuestro propósito es mostrar el proyecto complementario *pyristicLab*. PyristicLab es una aplicación web de código abierto que facilita el uso de la librería *pyristic*, donde, nos permite obtener estadísticas relevantes de manera ágil sobre el problema a resolver con solo implementar la función objetivo.

B.1.1. Instalación

Antes de proceder a examinar *pyristicLab* es necesario resolver la compatibilidad de dependencias.
La versión aceptada de las dependencias son:

1. pyristic==0.1.2
2. dash>=1.20.0
3. dash-bootstrap-components>=0.11.1

El proceso de instalación es el siguiente:

1. Descargamos el proyecto:

```
$ git clone https://github.com/JAOP1/pyristicLab.git
```

2. Creamos un entorno de conda para asegurarnos que no haya problemas de compatibilidad entre librerias de python (opcional).

```
$ conda create --name pyristic-env
```

3. Instalamos pyristic.

```
$ pip install pyristic
```

4. Instalamos Dash.

```
$ pip install dash
```

5. Instalamos dash-bootstrap

```
$ pip install dash-bootstrap-components
```

Si todo salió correctamente, podemos ejecutar el siguiente comando en el directorio de pyristicLab.

```
$ python main.py
```

Finalmente, debemos ver el siguiente mensaje en consola.

```
(MineriaDatos) [dell@dell-pc pyristicLab]$ ls
app.py  assets  layouts  LICENSE  main.py  __pycache__  README.md  requirements.txt  testFile.py
(MineriaDatos) [dell@dell-pc pyristicLab]$ python main.py
Dash is running on http://127.0.0.1:8050/home/

 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
```

Figura B.1: Mensaje en consola del proyecto pyristicLab.

Al abrir nuestro navegador, colocamos la url indicada en la consola (**http://127.0.0.1:8050/home/**) y nos mostrará la interfaz gráfica.

Figura B.2: Interfaz de pyristicLab.

B.1.2. Modo de uso

Antes de comenzar, es necesario definir el problema a resolver en el archivo *testFile.py* (por defecto tiene la función de Ackley). En este archivo, se tiene las siguientes variables reservadas:

- *optimizationProblem*: Almacena un diccionario que tiene las siguientes llaves:
  - `function`: La función objetivo.
  - `constraints`: Arreglo que contiene funciones booleanas que evalúan las soluciones.
  - `bounds`: Arreglo con los límites para cada una de las variables de decisión.
  - `decision_variables`: El número de variables de decisión del problema a optimizar.
- *aptitudeFunction*: En el caso de emplear algoritmos genéticos, es necesario implementar una función de aptitud (se emplea en la selección de padres).

En la imagen B.2 en la esquina inferior izquierda se encuentra una flecha, al dar clic se desplegará el menú de configuración que se muestra en la imagen B.3. Los algoritmos que cuenta pyristic actualmente se clasifican para resolver problemas de optimización continua y combinatorios. Al seleccionar el tipo de problema a resolver, se desplegará un listado de algoritmos como se muestra en la imagen B.3.

> Para mayor información consulta: https://jaop1.github.io/pyristic/

Al seleccionar alguno de los algoritmos, se desplegará un listado de parámetros que puedes emplear, finalmente se encuentra un botón con la leyenda *Ejecutar*. Después

Figura B.3: Barra de configuración de pyristicLab.

de dar clic, comenzará a generar las estadísticas que se mencionaron a lo largo de este trabajo.
En la imagen B.5 muestra los resultados obtenidos tras un conjunto de ejecuciones del algoritmo seleccionado. El promedio, la desviación estándar y la mediana se encuentran ubicados en la parte superior. La gráfica muestra el valor de la función objetivo obtenida en cada aplicación del algoritmo y el costado derecho está la configuración seleccionada.

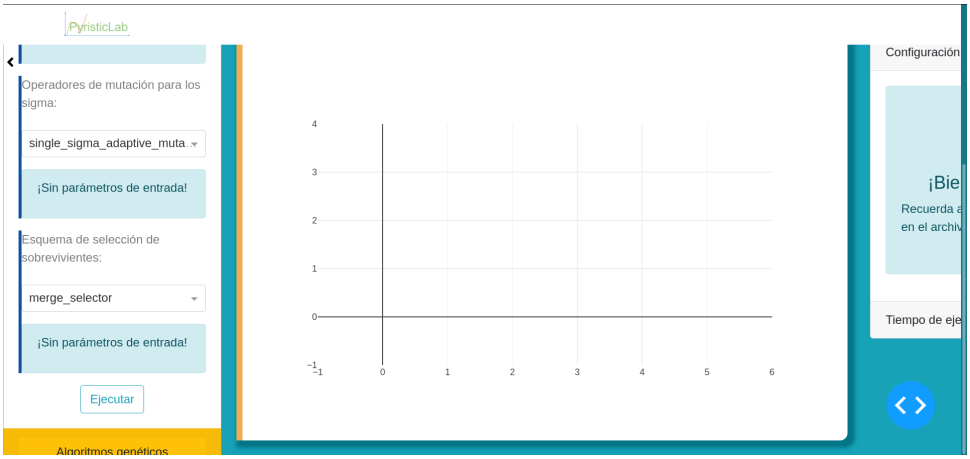


Figura B.4: Configuración de estrategias evolutivas.

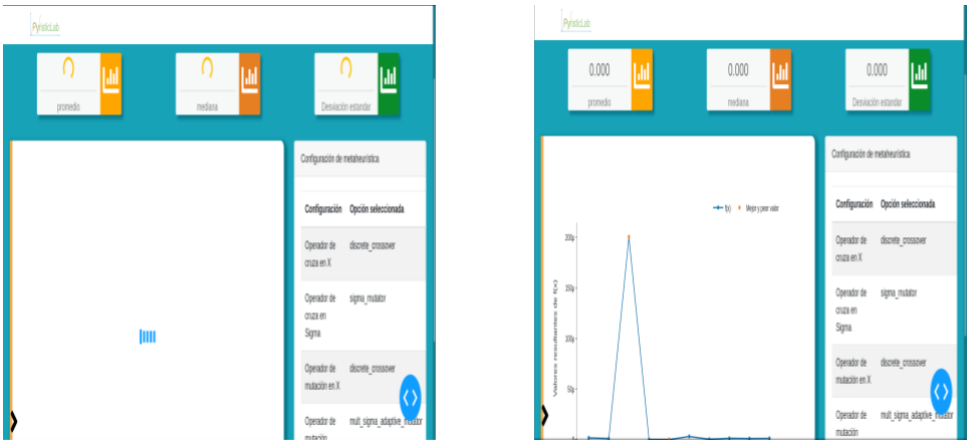


Figura B.5: Despliegue de resultados.

Bibliografía

[1] T. Dantzig and J. Mazur. *Number: The Language of Science*. A Plume book. Penguin Publishing Group, 2007. pages 9

[2] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution*. Wiley, Chichester, WS, UK, 1966. pages 2, 33

[3] Michel Gendreau and Jean-Yves Potvin. *Tabu Search*, pages 165–186. Springer US, Boston, MA, 2005. pages 75

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

[5] Fred Glover. Tabu search - part i. *INFORMS Journal on Computing*, 2:4–32, 01 1990. pages 2, 14

[6] Fred Glover. Tabu searchpart ii. *ORSA Journal on Computing*, 2:4–32, 02 1990. pages 2, 14

[7] Fred Glover and Manuel Laguna. *Tabu Search*, volume 7. Springer US, USA, 1997. pages 14

[8] J.H. Holland. *Adaption in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor. pages 57

[9] John H. Holland. Genetic algorithms. *Scientific American*, (1):66–73, 1992. pages 2, 51

[10] Eberhart R. Kennedy J. Particle swarm optimization. *Proceedings of ICNN'95 - International Conference on Neural Networks*, pages 1942–1948, 1995. pages 2

[11] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. pages 2, 23

[12] C Koulamas, SR Antony, and R Jaen. A survey of simulated annealing applications to operations research problems. *Omega*, 22(1):41 – 56, 1994. pages 23

[13] J. K. Lenstra and A. H. G. Rinnooy Kan. Some simple applications of the travelling salesman problem. *Journal of the Operational Research Society*, 26(4):717–733, 1975. pages 11

[14] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953. pages 2, 23

[15] Rody Oldenhuis. Test functions for global optimization algorithms. pages 12, 13

[16] Singiresu S. Rao. *Engineering Optimization Theory and Practice*. John Wiley & Sons, Ltd, 2019. pages 7

[17] Hans-Paul Schwefel. Kybernetische evolution als strategie der exprimentellen forschung in der stromungstechnik. Master's thesis, Technical University of Berlin, 1965. pages 2, 40

[18] Hans-Paul Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*. Birkhäuser Basel, 1977. pages 40

[19] Patrick Siarry and Gerard Berthiau. Fitting of tabu search to optimize functions of continuous variables. *International Journal for Numerical Methods in Engineering*, 40:2449 – 2457, 07 1997. pages 14

[20] Price K. Storn R. Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces. *Technical Report TR-95-012, Berkeley*, 1995. pages 2