Índice

Índice de figuras

DESARROLLO DE UNA HERRAMIENTA PARA LA APLICACIÓN DE ALGORITMOS DE ENJAMBRES COMO TÉCNICA DE CUANTIFICACIÓN DE COLOR

1. Introducción

La reducción de color en imágenes es un problema complejo en estos tiempos en los que nos comunicamos frecuentemente con el uso de imágenes ya que involucra la transformación de una imagen con una gran cantidad de colores en una versión simplificada con un número limitado de colores.

Puede parecer un proceso simple a primera vista, pero representa varios desafíos y requiere del uso de técnicas avanzadas, como algoritmos de enjambres, para obtener resultados óptimos. Algunas de las razones que explican por qué la reducción de color en imágenes es un problema complejo son:

- **Gran volumen de datos:** Ya que estas imágenes suelen contener millones de píxeles y cada píxel tiene diversa información sobre sí mismo, procesar todos estos datos de manera eficiente y efectiva es un gran desafío.
- **Preservación de detalles importantes:** para la cuantificación de color es esencial preservar los detalles importantes de la imagen, como bordes y características distintivas. La simplificación excesiva puede llevar a la pérdida de información crítica y degradar la calidad visual de la imagen.

- **Selección de colores representativos:** Elegir los colores que representarán la imagen reducida de manera precisa es una tarea compleja. La selección de estos colores debe hacerse de manera inteligente para garantizar que la imagen simplificada sea lo más fiel posible a la original.

- **Mantenimiento de la apariencia visual:** La reducción de color debe lograr un equilibrio entre la reducción del número de colores y la retención de la apariencia visual general de la imagen. Los cambios agresivos en los colores pueden hacer que la imagen resultante sea irreconocible o menos atractiva.

- **Optimización de la eficiencia:** Procesar imágenes en tiempo real o en aplicaciones donde la velocidad es crucial (como la transmisión de video) requiere algoritmos de reducción de color eficientes que puedan realizar cálculos rápidos y precisos.

- **Evaluación de la calidad:** Medir la calidad de una imagen reducida es subjetivo y puede depender de la percepción humana. Evaluar y comparar algoritmos en términos de calidad de reducción de color es un aspecto importante de la investigación en este campo.

1.1 Representación de imágenes en color

La representación de imágenes en color en un ordenador es un proceso fundamental que implica codificar y almacenar la información de color de una imagen para luego procesarla, visualizarla y manipularla. Para lograr esto, los ordenadores utilizan modelos de color, siendo el modelo RGB (Red, Green, Blue) uno de los más comunes.

En el modelo RGB, cada píxel de una imagen se representa como una combinación de tres canales de color: rojo, verde y azul. Cada canal puede tomar valores que representan la intensidad de ese color en un píxel específico.

La combinación de estos tres canales de color en cada píxel permite crear una gran cantidad de colores y tonalidades, lo que nos permite representar imágenes en color de alta calidad. La información de color de cada píxel se almacena en una matriz tridimensional, donde las dos primeras dimensiones representan la posición espacial del píxel en la imagen y la tercera dimensión almacena los valores de intensidad de rojo, verde y azul.

Esta representación de color en RGB es la base para todas las operaciones de procesamiento de imágenes en color, incluida la reducción de color. Durante este proceso, el objetivo es reducir el número de colores de la imagen, manteniendo al mismo tiempo la apariencia visual deseada. Los algoritmos de enjambres nos ayudan a "escoger" los colores adecuados para preservar la calidad en la imagen reducida.

1.2 Los modos de color

Ya que se ha hablado de como un ordenador representa las imágenes en color utilizando el modo de color o modelo RGB me parece importante hablar brevemente de otros modelos que existen.

Uno de los más simples es la **escala de grises** que tiene un solo canal, en el que representa valores de grises del blanco al negro. Pueden tener una profundidad de 8 o de 16 bits por píxel.

Otro de ellos es el **indexado** en el que las imágenes tienen una gama de colores reducida para que sus archivos sean más pequeños. Las imágenes indexadas por lo general se suelen usar en páginas web o en gráficos con pocos colores. Solo cuentan con un canal de 8 bits.

Otros dos modelos bastantes extendidos hoy en día son:

1. **CMYK:** Es un modo de color de cuatro canales. Los valores representan el color cian, el magenta, el amarillo y el negro. Al igual que el RGB puede tener 8 o 16 bits por canal. Este modo compone los colores de manera sustractiva, es decir, parte del blanco y va restando colores usando sus complementarios.

   Este modo es el que usan habitualmente las impresoras. A la hora de retocar las fotos en el ordenador nos servirá de gran ayuda para ver si los colores de la imagen que hemos trabajado en RGB serán los mismos que los impresos en CMYK ya que a veces no coinciden. Esto ocurre porque hay algunos colores RGB que no pueden conseguirse en CMYK.
2. **LAB:** En este modo existen 3 canales, L, A y B. L representa la información tonal o luminancia (intensidad de la luz en la imagen) y A y B la información de color o crominancia (como los colores se combinan y varían en una imagen), siendo A la que añade los colores en la línea del rojo y el verde y B la que añade el contenido en amarillo o azul.[1]

1.3 El espacio de color RGB

Un espacio de color es un sistema de interpretación del color, es decir, una organización específica de los colores en una imagen o video. Depende del modelo de color en combinación con los dispositivos físicos que permiten las representaciones reproducibles de color, por

ejemplo, las que se aplican en señales analógicas (televisión a color) o representaciones digitales. Un espacio de color puede ser arbitrario, con colores particulares asignados según el sistema y estructurados matemáticamente.[2]

Los algoritmos de reducción de color utilizan varios métodos para seleccionar un conjunto limitado de colores representativos de la imagen original. Estos colores se eligen mediante métodos de agrupación o clustering como el algoritmo K-means, donde se agrupan los colores similares en un número menor de grupos, y cada grupo se representa con un color promedio.

Una vez que se han seleccionado los colores representativos, la imagen se reemplaza por una versión simplificada en la que los píxeles originales se ajustan al color más cercano en la paleta de colores reducida. Esto permite reducir la cantidad de información de color en la imagen, lo que ahorra espacio de almacenamiento y puede mejorar la eficiencia en el procesamiento y la transmisión de imágenes, entre otros beneficios.

En resumen, el espacio de color RGB es la base para representar colores en imágenes digitales, y la técnica de reducción de color utiliza estrategias basadas en este espacio para simplificar y optimizar la representación de colores en una imagen, manteniendo al mismo tiempo la calidad visual deseada. Los algoritmos de enjambres y otras técnicas de optimización pueden desempeñar un papel importante en la selección de los colores representativos en este contexto.

2. El problema de la cuantificación de color

2.1 Definición del problema

Una imagen digital se forma con una cantidad indeterminada de píxeles la cual usa un cierto espacio de color. Cuando este espacio de color es el RGB, cada píxel se representa con 3 números enteros entre el 0 y el 255 por lo cual este espacio de color nos permite usar 256^3 colores diferentes.

Como se ha dicho anteriormente la cuantificación de color trata de reducir este número de colores evitando la pérdida de información. Para conseguirlo realiza dos operaciones:

1. **Selección de colores representativos:** Esta operación implica la elección de un conjunto limitado de colores representativos que serán utilizados para reemplazar los colores originales de la imagen. Para seleccionar estos colores, se aplican algoritmos de agrupación de colores, como el algoritmo K-means, que agrupa los colores similares en clusters y utiliza los centroides de estos clusters como colores representativos. Este conjunto de colores representativos lo llamamos paleta cuantizada. Esta operación es esencial para simplificar la representación de color de la imagen y reducir el número de colores utilizados.

2. **Asignación de colores representativos a los píxeles:** Una vez que se han seleccionado los colores representativos, se lleva a cabo la operación de asignación de colores. En esta etapa, cada píxel en la imagen original se asigna al color representativo más cercano en términos de distancia de color. Esto implica calcular la diferencia de color entre el color original del píxel y los colores representativos de la paleta cuantizada y seleccionar el color representativo que minimice esta diferencia. Esta operación es fundamental para reemplazar los colores originales por sus equivalentes representativos y, de esta manera, lograr la reducción de color en la imagen.

Ilustración 1 Imagen Inicial

Ilustración 2 Imagen resultante con 32 colores

2.2 Aplicaciones prácticas

La cuantificación de color tiene diversas aplicaciones prácticas en varios campos, tales como:

- **Procesamiento de imágenes y fotografía:** Se utiliza para reducir el número de colores en una imagen digital, lo que es útil para la compresión de imágenes y la optimización para dispositivos y medios con capacidad de color limitada.

- **Diseño gráfico y web:** Ayuda a los diseñadores a crear paletas de colores coherentes y limitadas para mantener la consistencia y mejorar la estética en diseños y sitios web.

- **Reconocimiento de patrones e identificación de objetos:** la cuantificación de color puede ayudar a simplificar la identificación y clasificación de objetos al reducir la complejidad de las imágenes, facilitando el reconocimiento de patrones.

- **Impresión y fabricación de textiles:** En la industria de la impresión y en la fabricación de textiles, la cuantificación de color es crucial para asegurar que los colores utilizados en los diseños sean reproducibles y consistentes en diferentes lotes de producción.

- **Cartografía y análisis geoespacial:** Se utiliza para simplificar la representación visual de mapas y datos geoespaciales, ayudando a resaltar características importantes y mejorar la legibilidad.

- **Análisis médico y de imágenes biomédicas:** En el campo de la medicina, la cuantificación de color se aplica en el análisis de imágenes biomédicas para mejorar la visualización y el diagnóstico de ciertas condiciones al resaltar variaciones sutiles en los tejidos.

- **Análisis de calidad y control en la industria alimentaria:** Se usa para evaluar la calidad y la madurez de alimentos basándose en su color, lo cual es especialmente útil en líneas de producción automatizadas.

- **Arte y restauración:** En el arte digital y la restauración de obras, la cuantificación de color puede ayudar a analizar y replicar los colores utilizados por los artistas originales, así como a detectar alteraciones o daños.

- **Seguridad y vigilancia:** La cuantificación de color puede mejorar el rendimiento de los sistemas de vigilancia al reducir la complejidad de las imágenes, facilitando la detección de movimientos o actividades sospechosas.

- **Investigación científica y análisis de datos:** En diversas áreas científicas, la cuantificación de color puede ser útil para visualizar y analizar datos, especialmente en representaciones gráficas de información compleja.

3. Los algoritmos de enjambres.

3.1 Características generales.

Los algoritmos de enjambres o inteligencia de enjambres es la disciplina que se ocupa de los sistemas naturales y artificiales compuestos por muchos individuos que se coordinan mediante el control descentralizado y la autoorganización. Se trata de un subcampo de la inteligencia artificial que se centra en los comportamientos colectivos que resultan de las interacciones de los individuos entre sí y con su entorno o medio ambiente al igual que lo hacen los enjambres naturales.

Algunos de los sistemas estudiados por la inteligencia del enjambre son las colonias de hormigas y termitas, los bancos de peces, las bandadas de pájaros o manadas de animales terrestres.

Ciertos artefactos humanos también caen en el dominio de la inteligencia de enjambre. En particular, algunos sistemas de múltiples robots, y también algunos programas de computadora que están escritos para abordar problemas de optimización y análisis de datos.[3]

Algunas propiedades de estos algoritmos son:

1. **La autonomía**: Cada agente (o partícula) en el enjambre opera de manera individual, tomando decisiones basadas en su información local y, posiblemente, en alguna información global disponible para el enjambre.
2. **Descentralización**: No existe un control centralizado que dirija las acciones de los agentes; en su lugar, el comportamiento global emerge de las interacciones locales entre los agentes y entre los agentes y su entorno.

3. **Distribución**: Los algoritmos de enjambres suelen ser inherentemente distribuidos, lo que los hace robustos y escalables, ya que el fallo de una partícula tiene un impacto limitado en el enjambre en su totalidad.
4. **Autoorganización**: Las interacciones entre las partículas y la aplicación de reglas sencillas dan lugar a un comportamiento colectivo "inteligente" y autoorganizado. Este comportamiento es emergente, es decir, surge de las acciones de las partículas individuales.
5. **Interacciones simples**: Las reglas que rigen las interacciones entre las partículas son simples, pero pueden dar lugar a comportamientos colectivos complejos y a la solución de problemas complejos.
6. **Retroalimentación positiva y negativa**: Los mecanismos de retroalimentación, tanto positiva como negativa, son muy importantes para el desarrollo y la estabilización de los patrones de comportamiento del enjambre. La retroalimentación positiva promueve la formación de estructuras o caminos para hallar una mejor solución mientras que la retroalimentación negativa ayuda a prevenir la saturación de datos o una mala solución rápida.
7. **Exploración y explotación**: Los algoritmos de enjambres equilibran entre la exploración del espacio de búsqueda para descubrir nuevas soluciones y la explotación de las mejores soluciones encontradas para afinarlas.
8. **Adaptabilidad**: La inteligencia de enjambre es altamente adaptable a cambios en el entorno o en los parámetros del problema, lo que permite al enjambre encontrar nuevas soluciones cuando las condiciones cambian.
9. **Robustez**: Los enjambres son robustos frente a fallos y variaciones, ya que la pérdida de algunas partículas generalmente no impide que el enjambre en su conjunto continúe funcionando eficazmente.

3.2 Algoritmo de optimización por enjambre de partículas (PSO)

3.2.1 Introducción

La **Optimización por Enjambres de Partículas** (conocida como **PSO**, por sus siglas en inglés, **Particle Swarm Optimization**) es una técnica de optimización/búsqueda.

Este método fue descrito alrededor de 1995 por Kennedy y Eberhart, y se inspira en el comportamiento de los enjambres de insectos en la naturaleza. En concreto, podemos pensar en un enjambre de abejas, ya que éstas a la hora de recolectar polen buscan la región del espacio en la que existe más densidad de flores, porque la probabilidad de que haya polen es mayor. La misma idea fue trasladada al campo de la computación en forma de algoritmo y se emplea en la actualidad en la optimización de distintos tipos de sistemas.

Formalmente hablando, se supone que tenemos una función desconocida, f(x1,…,xn), que podemos evaluar en los puntos que queramos pero a modo de caja negra, por lo que no podemos conocer su expresión. El objetivo es el habitual en optimización, encontrar valores de x1,…,xn para los que la función f sea máxima (o mínima, o bien verifica alguna relación extrema respecto a alguna otra función). A f se le suele llamar función fitness, ya que va a determinar cómo de buena es la posición actual para cada partícula.[4]

3.2.2 Autores

El Algoritmo de Optimización por Enjambre de Partículas (Particle Swarm Optimization, PSO) fue desarrollado por James Kennedy, un científico social, y Russell Eberhart, un ingeniero eléctrico, en 1995. El algoritmo se inspira en el comportamiento social y de forrajeo de las aves y los peces. Kennedy y Eberhart presentaron el PSO como una técnica computacional que simula el comportamiento social de los enjambres para resolver problemas de optimización. Desde su creación, el PSO ha sido ampliamente utilizado y adaptado para una variedad de aplicaciones complejas en varios campos gracias a su simpleza y eficacia.

James Kennedy (nacido el 5 de noviembre de 1950) es un psicólogo social estadounidense, más conocido como creador e investigador de la optimización del enjambre de partículas. Los primeros artículos sobre el tema, de Kennedy y Russell C. Eberhart, se presentaron en 1995; Desde entonces se han publicado decenas de miles de artículos sobre enjambres de partículas. El libro Academic Press/Morgan Kaufmann, Swarm Intelligence, de Kennedy y Eberhart con Yuhui Shi, se publicó en 2001. [5]

Russell C. Eberhart, ingeniero eléctrico estadounidense, mejor conocido como codesarrollador del concepto de optimización del enjambre de partículas. Es profesor de Ingeniería Eléctrica e Informática y profesor adjunto de Ingeniería Biomédica en la Escuela Purdue de Ingeniería y Tecnología de la Universidad Purdue de Indiana (IUPUI). Es miembro del IEEE y miembro del Instituto Americano de Ingeniería Médica y Biológica.

Obtuvo un doctorado. en ingeniería eléctrica de la Universidad Estatal de Kansas en 1972. Fue editor asociado de IEEE Transactions on Evolutionary Computation y ex presidente del IEEE Neural Networks Council. [6]

3.2.3 Variables

- Se considera un conjunto de **I** individuos llamados en este algoritmo **partículas**. Cada una de las **I** partículas tendrá asociado un estado que se irá modificando con el tiempo. Estas partículas tendrán:
    - Posición (X_i): representa el estado de la partícula, es una posible solución al problema.
    - Velocidad (V_i): determina la actualización de la posición.
    - Mejor posición personal (b_i): esta variable representa la mejor solución encontrada por la partícula durante la ejecución.
- La solución al problema viene dada por la mejor posición encontrada por el enjambre la cual llamamos la mejor posición global representada como: g(t).

En este algoritmo las partículas (individuos) se mueven por el espacio de solución del problema guiadas por ellas mismas y por el conjunto de todo el enjambre.

Este movimiento hace variar su posición, velocidad y mejor posición personal.

3.2.4 Definición del algoritmo

PSEUDOCODIGO

Inicializar la población de individuos

REPETIR

Evaluar el fitness de cada individuo.

Actualizar la mejor solución personal de cada individuo.

Actualizar la mejor solución global.

Actualizar la velocidad y posición de cada individuo.

HASTA(condición de parada)

Inicializar la población de individuos

Al principio del algoritmo se inicializan las variables de cada individuo para comenzar. Para cada individuo se inicia su **posición** y su **velocidad.**

Su posición será un valor aleatorio dentro del espacio de búsqueda y su velocidad será un valor aleatorio entre [v_{min} , v_{max}].

Evaluar el fitness de cada individuo

Para calcular este valor se aplica la función objetivo del problema a la posición de la partícula, en el caso de la cuantificación de color serán tales como MSE, MAE, SSIM o MS-SSIM (a partir de ahora función objetivo).

$$\mathbf{fitness_i \approx f(x_i)}$$

Este valor llamado fitness determina la calidad de la solución cuya posición representa.

Actualizar la mejor solución personal de cada individuo.

Si al calcular el fitness un individuo encuentra una mejor solución personal (es decir, encuentra una mejor posición lo que resulta en un mejor fitness) que la que tuviese almacenada para ella hasta el momento, actualiza la mejor solución con el valor de esta.

Se da un nuevo valor a b_i

Actualizar la mejor solución global.

Con la solución global ocurre lo mismo que con la mejor solución personal de cada individuo, si se ha encontrado una solución que mejora a la que hubiese almacenada como mejor solución global, se guarda como nueva mejor solución global.

Se da un nuevo valor a g(t).

Actualizar la velocidad y posición de cada individuo.

En este paso se realizan dos procedimientos:

1. *Actualizar velocidad de cada individuo.*

   Se calcula una nueva velocidad de cada individuo i:

   $v_i\ (t+1) = \mathbf{w}\ v_i\ (t) + \mathbf{f_1}\ \mathbf{e_1}\ [b_i\ (t)-x_i\ (t)] + \mathbf{f_2}\ \mathbf{e_2}\ [g(t)-x_i\ (t)]$

   Cuyos parámetros son:

   e_1, e_2: valores aleatorios en [0, 1].

   W: inercia.

   f_1: parámetro cognitivo.

   f_2: parámetro social.

Esta actualización incluye tres componentes que son:

-Velocidad anterior:

$v_i\ (t+1) = \mathbf{w}\ \mathbf{v_i\ (t)} + f_1\ e_1\ [b_i\ (t)-x_i\ (t)] + f_2\ e_2\ [g(t)-x_i\ (t)]$

Donde w (inercia) debe seleccionarse atendiendo a que si es grande se realizará una exploración global, si es pequeña será exploración local. Se aconseja que este parámetro decrezca con las iteracciones.

-La distancia a la mejor posición personal:

$v_i\ (t+1) = w\ v_i\ (t) + \mathbf{f_1}\ \mathbf{e_1}\ \mathbf{[b_i\ (t)-x_i\ (t)]} + f_2\ e_2\ [g(t)-x_i\ (t)]$

En esta componente encontramos dos parámetros:

e_1: será un valor aleatorio entre [0, 1].

f_1: el parámetro cognitivo que determina la importancia que se da a la experiencia propia de cada individuo.

-La distancia a la mejor solución global:

$v_i\ (t+1) = w\ v_i\ (t) + f_1\ e_1\ [b_i\ (t)-x_i\ (t)] + \mathbf{f_2\ e_2\ [g(t)-x_i\ (t)]}$

En esta componente tenemos los parámetros:

e_2: será un valor aleatorio entre [0, 1].

f_2: el parámetro social que determina la importancia que se da a la experiencia del enjambre.

Con el valor calculado de la velocidad, se debe ajustar al intervalo válido de la velocidad, es decir, nuestro nuevo valor: $v_i\ (t+1)$ deberá estar entre $[v_{min}\ ,\ v_{max}\ ]$.

Para ello:

- Si $v_i\ (t+1) > v_{max}$ => $v_i\ (t+1) = v_{max}$

-Si $v_i\ (t+1) < v_{min}$ => $v_i\ (t+1) = v_{min}$

2. *Actualizar posición de cada individuo.*

Se calcula la nueva posición de cada individuo i.

$x_i\ (t+1) = x_i\ (t) + v_i\ (t+1)$

Este nuevo valor para la posición de cada individuo depende de la actual posición y de la nueva velocidad que se acaba de calcular.

3.2.4 Métodos de cálculo para la inercia (w)

Como se ha mencionado antes, el parámetro de inercia en el PSO es recomendable que vaya decreciendo con el tiempo, ya que este parámetro se encarga de regular la velocidad asociada a cada partícula. Esta demostrado que disminuir este parámetro durante la ejecución del algoritmo mejora el resultado, aunque originalmente no se contemplaba.

En 1998, Shi y Eberhart introdujeron el concepto de masa inercial al aplicar un coeficiente de inercia para limitar la velocidad de las partículas en el algoritmo PSO; a su vez establecieron que dicho coeficiente facilitaba la búsqueda global cuando se trataba de un número grande, mientras que si se trataba de un número pequeño se facilitaba la búsqueda local.

Sin embargo, se ha demostrado en numerosos estudios que el parámetro de inercia, con ajuste dinámico en el tiempo, puede aumentar significativamente la convergencia de una solución, comparándolo con un valor constante

Para las pruebas realizadas se consideraron dos métodos de descenso de la inercia: lineal y caótico.

1.Lineal

Este método ajusta la inercia siguiendo la fórmula:

$$w(t+1) = (w_{max} - w_{min})\left(\frac{t_{max} - t}{t_{max}}\right) + w_{min}$$

De donde w_{max} es el valor inicial del parámetro de inercia al iniciar el algoritmo y w_{min} representa el valor final. Se ha reportado que el rango idóneo para el parámetro w es de [0.9 - 0.4]. El método de descenso lineal habitualmente presenta problemas de convergencia con funciones que tiene una mayor cantidad de variables en su solución.

2-Caótico

La inercia caótica fue propuesta por Feng en donde agrega al método de descenso lineal un factor que genera un mapeo caótico en el rango [0,1]:

$$w(t+1) = \mu(z)(z-1)$$

Donde 3.57< μ ≤4 y z es un número aleatorio de [0,1]. Sin embargo, cuando μ = 4 se cubre el intervalo de [0,1]. Se ajusta entonces con esta fórmula:

$$w(t+1) = (w_{max} - w_{min})\left(\frac{t_{max} - t}{t_{max}}\right) + w_{min}(z)$$

[7]

3.2.5 Parámetros PSO

Como se ha hablado antes en el PSO existen varios parámetros principales que influyen en el comportamiento y en la eficacia del algoritmo, son: el coeficiente de **inercia**, y los parámetros **cognitivo** y **social**.

Inercia:

Este coeficiente afecta a la velocidad a la que los individuos se mueven en el espacio de búsqueda. Ayuda a controlar los resultados de la velocidad anterior del individuo sobre su velocidad actual. Un valor alto ayuda a que el individuo mantenga su dirección y velocidad anterior, lo que se traduce en una mejor exploración global. Por otro lado, un valor bajo hace que el individuo pierda velocidad por lo que ayuda a una exploración más local y detallada.

Parámetro cognitivo:

También conocido como parámetro personal o local, determina el grado en el que cada individuo se ve influido por si mismo. Influye en cuánto se moverá un individuo hacia la mejor posición encontrada

individualmente. Un valor alto de este componente dará a cada individuo una mayor motivación para seguir su propio camio hacia lo que percibe como una mejor solución, dando así lugar a una mayor variedad de soluciones exploradas por el enjambre.

Parámetro social:

Determina el grado de influencia de la mejor posición encontrada por todo el enjambre sobre el movimiento de cada individuo. Valores mas altos de este componente resultan en una mayor convergencia del enjambre hacia las áreas del espacio de búsqueda que parecen más prometedoras según la experiencia del enjambre, es decir, los individuos preferirán moverse hacia la mejor solución encontrada por cualquier individuo.

3.3 Luciérnagas (Firefly)

3.3.1 Introducción

El algoritmo firefly, es un algoritmo metaheurístico, inspirado en el comportamiento del centelleo de las luciérnagas. El propósito primario de una luciérnaga es generar destellos de luz para actuar como sistema de señal para atraer a otras luciérnagas. Xin-She Yang formuló este algoritmo con las siguientes premisas:

1. Todas las luciérnagas son "unisexuales", de modo que cualquier luciérnaga individual será atraída por todas las demás.
2. La atracción es proporcional a su brillo, y para cualquier par de luciérnagas, la menos brillante será atraída por (y por lo tanto se desplazará hacia) la más brillante; aun así, la intensidad (el brillo aparente) decrece cuando aumenta la distancia entre ambas;
3. Si no hay luciérnagas más brillantes que una dada, esta se mueve aleatoriamente.

El brillo es asociado con los valores de una función objetivo.

El algoritmo firefly es un procedimiento metaheurístico de optimización inspirado en la naturaleza.[8]

3.3.2 Autor

El autor de este algoritmo fue **Xin-She Yang** en 2008.

Yang es un dedicado investigador en el campo de la ingeniería y la informática, escribió sobre este algoritmo al intentar utilizar diversos métodos inspirados en la naturaleza para resolver problemas que son difíciles de resolver usando técnicas de optimización convencionales.[9]

Xin-She Yang es un matemático británico de origen chino, investigador experto del Laboratorio Físico Nacional, es conocido por desarrollar varios algoritmos heurísticos para optimización en ingeniería.

Ideó el algoritmo firefly (2008), la búsqueda cuckoo (2009), el algoritmo de murciélago (2010), y el algoritmo de polinización floral.

Estos algoritmos se han convertido en herramientas importantes en inteligencia artificial, aprendizaje de máquinas, informática neuronal y aplicaciones de ingeniería. Desde 2009, más de 1.000 artículos científicos de publicaciones acreditadas han citado el algoritmo firefly y la búsqueda cuckoo. Además, desarrolló la hipótesis de Van Flandern-Yang en colaboración con Tom Van Flandern para explicar las variaciones de gravedad durante el eclipse solar de 1977,13 y con otros fenómenos físicos. [10]

3.3.3 Variables

En este algoritmo se considera un conjunto de I individuos (luciérnagas). Cada uno de estos individuos representa una posible solución al problema. La calidad de esa solución dependerá del **brillo** que esté asociado a cada individuo.

Posición → $f_i = (x_{i1} , ...x_{ir})$

Brillo → $L_i \approx f(f_i)$

En este contexto el **brillo** de cada individuo será el fitness de este algoritmo, es decir, el individuo que posea el mayor brillo será la mejor solución al problema. Como en este algoritmo los individuos se mueven hacia el individuo con un mayor brillo, todos ellos se acercan a la mejor solución, menos la más brillante.

La **atracción** entre los individuos disminuye con la distancia debido a la absorción de la luz. Este valor normalmente va cambiando para representar la disminución (visibilidad) del brillo con la distancia.

3.3.4 Definición del algoritmo

PSEUDOCODIGO

Generar la población inicial de luciérnagas

REPETIR

Mover cada luciérnaga hacia las más brillantes

Mover la luciérnaga más brillante

Actualizar el brillo de las luciérnagas

Ordenarlas por brillo y encontrar la mejor

HASTA (condición de parada)

Generar la población inicial de luciérnagas

En este primer paso repartimos los individuos aleatoriamente en el espacio de búsqueda y calculamos el brillo inicial de cada uno.

Por cada iteración:

Mover cada luciérnaga hacia las más brillantes

Cada individuo (excepto la más brillante) mueve su posición hacia la más brillante, debido a esto se calcula una nueva posición para cada luciérnaga, en función de su posición previa y de la atracción ejercida por las otras luciérnagas.

Un individuo **I** se mueve hacia otro individuo **k**, que es mas brillante que él, siguiendo esta ecuación:

$f_i(t+1) = f_i(t) + \beta(r_{ik})[f_k(t) - f_i(t)\ ] + \varepsilon_i$

Donde:

$f_i(t+1)$ -> es la siguiente posición del individuo i

$f_i(t)$ -> es la posición actual del individuo i

$\beta(r_{ik})$ -> es el atractivo del individuo k sobre el i

ε -> un valor aleatorio

Donde $\beta(r_{ik})$ se calcula mediante:

$$\beta(r_{\mathrm{ik}}) = \beta_0 e^{-\gamma r_{ij}^2}$$

β_0 -> atractivo a distancia cero

γ ->coeficiente de absorción de la luz del medio

r_{ij} -> distancia entre los individuos i y j

Mover la luciérnaga más brillante

El individuo más brillante no se ve atraído por ningún otro, por lo que se mueve aleatoriamente.

Su nueva posición será:

$f_{best}(t+1) = f_{best}(t) + \varepsilon$

Actualizar el brillo de las luciérnagas

En este paso se calcula el fitness (brillo) de cada individuo mediante la función objetivo del problema.

Ordenarlas por brillo y encontrar la mejor

En este ultimo paso se ordenan los individuos por brillo y se determina el mejor (el que más brillo tenga). Para ello se utilizan diversos algoritmos de ordenación comunes (esta elección dependerá del número de individuos, la complejidad del algoritmo, y si el conjunto de datos esta parcialmente ordenado). Algunos de ellos son:

- Quicksort.
- Mergesort.
- Heapsort.
- Insertion Sort ….

Como se ha dicho, esta elección depende de la implementación y entorno de ejecución. Por ejemplo, si se utiliza Python es posible usar el método de ordenamiento integrado que es TimSort bajo el método .short() de las listas, que suele ser suficiente para la mayoría de las aplicaciones prácticas.

3.4 Lobos / Grey Wolf optimization (GWO)

3.4.1 Introducción

Este algoritmo de optimización metaheurística está basado en el comportamiento social y de caza de los lobos grises. Fue desarrollado por **Seyedeli Mirjalili** en 2014, simula tanto la estructura social de estos lobos como sus tácticas de caza para resolver complejos problemas de optimización. Estos lobos son conocidos por su comportamiento de caza estratégico y su jerarquía social, estos componentes se incluyen en este algoritmo para resolver estos problemas.

En esta jerarquía social, los individuos se clasifican en varias categorías: Alfas, Betas, Deltas y Omegas. Los Alfas son los que lideran la manada, estos toman las decisiones sobre la caza, el movimiento del grupo y su descanso. Los Betas, que son los colideres, ayudan a los Alfas en la toma de decisiones y en la comunicación con el resto de los individuos de la manada. Los Deltas ocupan un rango inferior, estos se ocupan de cuidar o vigilar. Por último, los Omegas están en la base de la pirámide jerárquica, estos siguen las decisiones de los miembros de rango superior sin cuestionar sus decisiones.

Este algoritmo utiliza esta jerarquía para modelar la cooperación entre los lobos en la resolución de problemas de optimización. Los mejores candidatos que son los Alfas lideran la búsqueda, estos son seguidos por los Betas y los Deltas, mientras que los Omegas siguen las direcciones establecidas por los miembros superiores. Durante el proceso de optimización, todos los lobos, que son todas las posibles soluciones, van ajustando sus posiciones respecto a los lobos de mejor desempeño en su entorno (espacio de búsqueda).

Este enfoque le permite al GWO explorar eficazmente el espacio de búsqueda y explotar las áreas mas prometedoras que se acerquen a las mejores soluciones encontradas. Posee un gran equilibrio entre la exploración y la explotación de las soluciones, y una gran capacidad para no quedarse estancado en mínimos locales, el GWO ha demostrado ser efectivo en una gran variedad de aplicaciones prácticas, desde la ingeniería hasta la ciencia de datos. Desde sus inicios, este algoritmo ha generado un gran interés en la comunidad de investigación por su robustez y rendimiento superior en comparación con otras técnicas de optimización.[11]

3.4.2 Fundamentos

El GWO introduce N lobos en la manda, desde aquí llamados N individuos.

-Cada único individuo representa una posible solución al problema.

-La calidad o fitness de esa solución se calcula mediante la función objetivo del problema.

Esta jerarquía se representa de la siguiente manera:

Los tres mejores lobos de la manada estarán en los tres niveles superiores de la jerarquía, es decir, en cada uno de estos tres niveles habrá un solo lobo.

-El mejor lobo (mejor solución) se denota como α.

-El segundo mejor lobo se denota como β.

-El tercer mejor lobo se denota como δ.

-El resto de los lobos se consideran lobos ω.

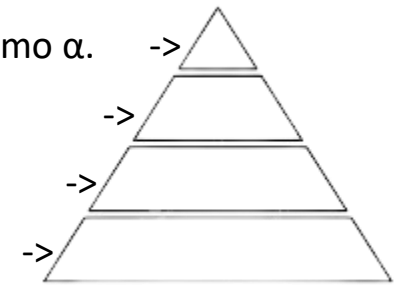


Ilustración 3 Pirámide

El proceso de optimización está representado por la operación de caza, en esta actividad los lobos de los tres primeros niveles guían, mientras que el resto siguen a éstos.

Este algoritmo posee varios coeficientes sociales (a, A y C).

El parámetro a o **factor de exploración** es el que guía el proceso de búsqueda. Este parámetro varía linealmente desde 2 hacia 0 con las iteraciones y se utiliza para calcular dos vectores aleatorios (A y C) que serán los condicionantes de la exploración/explotación del algoritmo.

Estos vectores al depender de a también disminuirán con las iteraciones. Estos se calculan de la siguiente manera:

$A = 2ar_1 - a$ (Ec 1) $C = 2r_2$ (Ec 2)

Donde:

r_1, r_2: son vectores aleatorios en [0, 1].

a: es el parámetro del algoritmo (factor de exploración)

Cuando el valor absoluto de estos vectores es mayor que uno se entiende que los individuos divergen respecto de la presa, se dice que están en fase de exploración.

Al converger a la presa, este valor será menor que uno, en este caso están en fase de explotación.

[12]

3.4.3 Definición del algoritmo

En el GWO la búsqueda de la presa, es decir, la solución del problema se inicia con la creación de una población de N individuos al azar, igual que en el resto de los algoritmos.

Después de esto se realiza un proceso iterativo en el cual:

1. Los lobos: alfa, beta y delta, estiman la posición probable de la presa.
2. Cada lobo del grupo actualiza su distancia respecto de la presa.
3. Se decrementa el parámetro a.

PSEUDOCODIGO

Inicializar la población de lobos

Calcular el fitness de los lobos

Determinar los lobos alfa, beta y delta

REPETIR

Actualizar la posición de cada lobo

Ajustar el parámetro a

Calcular el fitness de los lobos

Determinar los lobos alfa, beta y delta

HASTA (condición de parada)

En el primer paso, inicializar la población de individuos, se escogen unos valores aleatorios del espacio de solución para cada individuo X_i

Aparte de esto también se inicializa el parámetro **a**, este factor de exploración comienza con el valor de 2.

En el segundo paso, se calcula el **fitness** de cada individuo, esto se hace en base al valor de la función objetivo en X_i

$f_i \approx f(X_i)$

Lo siguiente es decidir los individuos alfa, beta y delta que serán los individuos con mejor fitness.

X_α, X_β y X_δ

En el bucle, el primer paso será actualizar la posición de cada individuo. Esta nueva posición será:

$X_p(t+1) = (X_1 + X_2 + X_3) / 3$

$X_1 = X_\alpha - A_1 D_\alpha$ $\quad\quad$ $D_\alpha = |C_1 X_\alpha - X_p(t)|$

$X_2 = X_\beta - A_2 D_\beta$ $\quad\quad$ $D_\beta = |C_2 X_\beta - X_p(t)|$

$X_3 = X_\delta - A_1 D_\delta$ $\quad\quad$ $D_\delta = |C_3 X_\delta - X_p(t)|$

En donde:

T: es la iteración actual.

A_i , C_i (i=1, 2, 3): vectores de coeficientes, calculados usando la Ec1 y la Ec2.

En el siguiente paso, se **actualiza el parámetro** del algoritmo.

El factor de exploración (a) se reduce linealmente en el intervalo [2,0] al avanzar las iteraciones, según:

$$\mathrm{a} = 2 - 2\frac{t}{max}$$

En donde:

t: es la iteración actual del algoritmo.

Max: es el número máximo de iteraciones del algoritmo.

[13]

3.4.4 Variables

Como en cualquiera de estos algoritmos tenemos **el tamaño de la población** de individuos (N), en este caso es el número de lobos que existen en la población. Un mayor numero de individuos puede mejorar la exploración del espacio de búsqueda, pero a su vez aumenta el costo computacional.

Ilustración 4 Lobos Grises

En el GWO existen 3 parámetros o coeficientes sociales los cuales ya se han mencionado que son a, A y C.

- a: Disminuye linealmente de 2 a 0 a lo largo de las generaciones y se utiliza para calcular los coeficientes A y C.
- A: Calculado de la forma (Ec1), es un numero aleatorio entre [0,1]. Este coeficiente afecta a la amplitud del movimiento del individuo (lobo) hacia o desde la presa.
- C: Calculado de la forma (Ec2), este coeficiente influye en el reconocimiento y la localización de la presa, proporcionando un elemento de aleatoriedad en el acercamiento de los lobos a la presa.

Jerarquía de Lobos (Alpha, Beta, Delta)

- Alpha (α): El lobo líder, que posee la mejor solución encontrada hasta el momento.
- Beta (β): El segundo mejor lobo, apoya al Alpha en la toma de decisiones y mantiene la disciplina dentro del grupo.
- Delta (δ): El tercer en la jerarquía, juega un rol subordinado pero esencial, facilitando la comunicación entre el Alpha y Beta con el resto de la manada.
- Omega (ω): El resto de los lobos, siguen las direcciones de los lobos con rango superior.

La función objetivo, de la misma manera que el resto de los algoritmos poseen una función objetivo, este también la tiene, es la función que cada individuo intenta optimizar, determinando la calidad de cada solución.

Por último, los criterios de terminación del problema, que son las condiciones bajo las cuales el algoritmo debe finalizar, ya pueden ser una solución concreta, el número máximo de iteraciones o una limitación del tiempo de ejecución.

3.4.5 Autor: **Seyedali Mirjalili**

El profesor Seyedali Mirjalili (Ali) fundó el Centro de Investigación y Optimización en Inteligencia Artificial en 2019. Actualmente, es profesor de Inteligencia Artificial en la Universidad de Torrens, Australia. Es reconocido internacionalmente por sus avances en Optimización e Inteligencia de Enjambre, incluyendo el primer conjunto de algoritmos desde una perspectiva de inteligencia sintética, un cambio radical de cómo se entienden típicamente los sistemas naturales, y un marco de diseño sistemático para evaluar, validar y proponer algoritmos de optimización robustos y computacionalmente económicos.

Ali ha publicado más de 500 publicaciones con más de 80,000 citas y un índice H de 95. Como el investigador más citado en Optimización Robusta, ha estado en la lista del 1% de los investigadores más citados y ha sido nombrado como uno de los investigadores más influyentes del mundo por Web of Science durante tres años consecutivos desde 2019. En 2020, fue clasificado en el puesto 21 en todas las disciplinas y 4º en Inteligencia Artificial y Procesamiento de Imágenes en la lista de la Universidad de Stanford de los Científicos Más Destacados del Mundo. En 2021, el periódico The Australian lo nombró como el investigador principal en Australia en tres campos de la Inteligencia Artificial, Computación Evolutiva y Sistemas Difusos. Ali es miembro senior del IEEE y editor asociado de varias revistas de IA, incluyendo Neurocomputing, Applied Soft Computing, Advances in Engineering Software, Computers in Biology and Medicine, Healthcare Analytics, Applied Intelligence y IEEE Access. Sus intereses de investigación incluyen Optimización, Inteligencia de Enjambre, Algoritmos Evolutivos y Aprendizaje Automático.

Seyedali Mirjalili es un investigador altamente influyente en el campo de la inteligencia artificial y la optimización. Algunas de sus otras contribuciones y logros son:

Desarrollo de Algoritmos de Optimización: Además del Algoritmo de Optimización por Manada de Lobos Grises (Grey Wolf Optimizer, GWO), Mirjalili ha desarrollado otros algoritmos influyentes basados en la naturaleza, como el Algoritmo de la Salamandra, el Algoritmo de la Ballena (Whale Optimization Algorithm), y el Algoritmo de Optimización del Águila (Eagle Strategy). Estos algoritmos también están inspirados en comportamientos animales y se utilizan para resolver complejos problemas de optimización en diversas áreas de la ingeniería y la ciencia.

Publicaciones y Citas: Con más de 500 publicaciones científicas y una cantidad significativa de citas, Mirjalili ha contribuido considerablemente a la literatura de inteligencia artificial y optimización. Su trabajo ha sido fundamental en avanzar la comprensión y aplicación de algoritmos metaheurísticos en problemas reales.

Actividad Editorial: Mirjalili sirve como editor asociado y revisor en diversas revistas científicas de alto impacto. Su trabajo editorial ayuda a dar forma a la dirección de la investigación en inteligencia artificial y campos relacionados, asegurando la calidad y relevancia de las investigaciones publicadas.

Reconocimientos y Rankings: Los reconocimientos de Mirjalili incluyen estar en la lista de los científicos más citados y ser reconocido por su contribución a campos específicos como la inteligencia artificial, computación evolutiva y sistemas difusos. Estos

honores reflejan su influencia y estatus en la comunidad científica global.

Mentoría y Enseñanza: Como académico en la Universidad de Torrens y en otros institutos antes de eso, Mirjalili también ha desempeñado un papel significativo en la formación de la próxima generación de científicos e ingenieros. Su enfoque en la enseñanza y la mentoría asegura que su conocimiento y experiencia se transmitan a los futuros investigadores.

Contribuciones a la Conferencia y la Comunidad: Mirjalili es un participante activo en conferencias científicas internacionales, donde presenta sus investigaciones, comparte ideas y colabora con otros expertos en su campo. Esto no solo amplía el alcance de su trabajo, sino que también fomenta colaboraciones que pueden llevar a nuevas innovaciones.

Estas actividades reflejan un compromiso profundo no solo con la investigación individual, sino también con el avance del campo en su conjunto, impactando positivamente en la academia y en la industria mediante la aplicación de soluciones innovadoras a problemas complejos.

[14], [15]

3.5 El algoritmo de Ballenas (Whale Swarm Algorithm - WSA)

3.5.1 Introducción

El WSA parte de la idea del WOA (Whale Optimization Algorithm), este algoritmo inspirado en el comportamiento de caza de las ballenas jorobadas fue desarrollado por Seyedali Mirjalili en 2016 (Lobos).

El WOA simula la técnica de burbujeo empleada por estos animales, es una estrategia predadora que utiliza la creación de burbujas en círculos o redes alrededor de su presa antes de atraparla. Este comportamiento es interesante desde un punto de vista de optimización, ya que engloba tanto los elementos de exploración como de explotación, que son las claves para la eficacia en la búsqueda de soluciones optimas en espacios complejos y multidimensionales.

Este algoritmo posee una estructura simple pero potente que permite resolver una amplia gama de problemas de optimización en ingeniería, informática y más allá. La adaptabilidad y eficiencia del WOA son resultados gracias a su capacidad para imitar la dinámica natural de estas ballenas, ajustando sus posiciones en el espacio de búsqueda de manera que maximiza las posibilidades de localizar y converger hacia un óptimo global.

3.5.2 Fundamentos

El WOA se basa en el comportamiento de caza de las ballenas jorobadas, particularmente en la técnica conocida como "burbujeo", donde las ballenas crean burbujas en círculos concéntricos alrededor de su presa. Esta técnica no solo confina a la presa, sino que también reduce sus vías de escape, facilitando a la ballena capturarla eficientemente.

Modelo Matemático del WOA

El WOA utiliza dos mecanismos principales basados en este comportamiento: la encerrona por burbujeo y los movimientos en espiral hacia la presa. La elección entre estos dos comportamientos se modela mediante un enfoque probabilístico que depende de la proximidad de la solución candidata al mejor candidato actual (la presa).

Encerrona por Burbujeo:

En este enfoque, las posiciones de las ballenas se ajustan según la posición del mejor candidato actual. La actualización de la posición se realiza utilizando las siguientes ecuaciones, donde la posición se mueve hacia el mejor candidato o se aleja ligeramente de él basándose en un coeficiente aleatorio.

Movimiento en Espiral:

Este comportamiento simula el movimiento en espiral de las ballenas alrededor de su presa. Se utiliza una ecuación helicoidal para actualizar la posición de la ballena en dirección a la presa.

Características del WOA

-Exploración y Explotación:

El WOA equilibra eficazmente la exploración (buscando en nuevas áreas del espacio de búsqueda) y la explotación (intensificando la búsqueda cerca del óptimo conocido). La alternancia entre el comportamiento de encerrona por burbujeo y el movimiento en espiral permite este equilibrio, haciendo que el WOA sea robusto en encontrar soluciones globales.

-Flexibilidad y Aplicabilidad:

El algoritmo se ha aplicado en una variedad de problemas, desde optimización de funciones continuas hasta problemas de diseño industrial, demostrando su versatilidad y efectividad.

-Simplicidad de Implementación:

A pesar de su eficacia, el WOA es relativamente simple de implementar comparado con otros algoritmos metaheurísticos, lo que lo hace accesible para investigadores y profesionales.

[16]

Una vez definido de donde parte el WSA se comienza a definir esta nueva técnica de optimización, su principio radica en acercarse a su presa siguiendo estrictamente al líder del grupo. Para ello, en primer lugar, se obtienen los valores promedio de posición del enjambre en cada iteración. Luego, cuando el parámetro p, que se utiliza para añadir aleatoriedad al progreso de los miembros del enjambre, esta por debajo de un cierto valor, se utiliza el promedio del enjambre para que cada individuo se moviera hacia la nueva posición. Así se elimina la lenta convergencia y frecuente caída en el optimo local, que se considera la mayor desventaja del algoritmo. La distancia de las ballenas entre sí y de la presa se modeló como una función de aptitud y se utilizó la fórmula de distancia euclidiana para ello. Se eligió un problema de ingeniería complejo para revelar el poder tanto del algoritmo de optimización de ballenas clásico como del algoritmo que incluye la nueva técnica propuesta. Como resultado, esta nueva técnica introducida ha proporcionado una mejora de 10 millones de veces en la solución de este complejo problema.

El WSA se basa en la capacidad natural en la que las ballenas encuentran su comida, migran y se aparean usando ultrasonidos. Gracias a los ultrasonidos avisan a otras ballenas de esto a grandes distancias.

Cuando una ballena encuentra una fuente de comida, emitirá ultrasonidos para notificar a otras ballenas cercanas de la calidad y cantidad de comida que hay, por lo que cada ballena sabrá donde ir.

3.5.3 Definición del algoritmo

Para representar este comportamiento se conformaron las siguientes reglas:

- Todas las ballenas se comunican con otras mediante ultrasonidos en el área de solución.
- Cada ballena tiene la capacidad de calcular que distancia hay de una a otra.
- La calidad y cantidad de comida encontrada por la ballena se representa mediante su fitness.
- El movimiento de cada ballena depende de la ballena con un mejor fitness que ella y más cercana.

PSEUDOCODIGO

Iniciar población de Ballenas

REPETIR

  REPETIR

    Encontrar mejor y más cercana ballena.

    SI SE ENCUENTRA:

      Movemos ballena (Eq.1)

    FIN SI

    Evaluamos ballenas

  HASTA (Nº Ballenas)

HASTA (condición de parada)

Eq 1: Mover ballenas $x_i^{t+1} = x_i^t + \mathbf{rand}\left(0, \rho_0 \cdot e^{-\eta \cdot d_{\mathbf{X},\mathbf{Y}}}\right) * \left(y_i^t - x_i^t\right)$

Se mueve cada ballena siguiendo la ecuación

X_i^{t+1} -> Siguiente posición de la ballena i

X_i^t -> Posición actual de la ballena i.

N.º aleatorio entre 0 y $\rho_0 \cdot e^{-\eta \cdot d_{\mathbf{X},\mathbf{Y}}}$

Donde:

P_0 es la intensidad del ultrasonido.

n es la probabilidad de distorsión del mensaje (causada por el agua)

d es la distancia euclídea entre la ballena x e y.

Y_i^t -> Posición actual de la ballena Y.

[17]

3.6 Abejas – Algoritmo Colonia de Abejas Artificiales (ABC)

3.6.1 Introducción

El Algoritmo de Colonia de Abejas Artificiales (Artificial Bee Colony, ABC) es una técnica de optimización metaheurística que fue desarrollada por Karaboga en 2005. Inspirado por el inteligente comportamiento forrajero de las abejas, el ABC simula las dinámicas de búsqueda de comida de una colmena de abejas para encontrar soluciones óptimas en problemas complejos. Este algoritmo se basa en el modelo de división del trabajo y el reclutamiento mutuo entre abejas, características que permiten una exploración eficiente del espacio de soluciones.

La colonia de abejas se divide en tres grupos: **exploradoras**, **empleadas** y **observadoras**.

-Exploradoras: Buscan aleatoriamente nuevas fuentes de alimentos (soluciones potenciales).

-Empleadas: Explotan las fuentes conocidas y comunican su calidad a las abejas observadoras en la colmena a través de una danza que simula una danza real que las abejas hacen para transmitir la ubicación y la calidad de las fuentes a explotar.

-Observadoras: Evalúan la calidad de las fuentes de alimento ya conocidas y eligen seguir a las abejas empleadas hacia las mejores fuentes.

Este proceso iterativo de exploración y explotación sigue hasta alcanzar una condición de terminación.

Este algoritmo ha demostrado ser efectivo en problemas de optimización numérica y ha sido aplicado en muchos campos como la ingeniería, la investigación operativa y la inteligencia artificial. La simplicidad de este algoritmo sumada a su potente capacidad de optimización ha ayudado en gran parte a su popularidad y gran aceptación en la comunidad científica.[18], [19]

3.6.2- Autor

Dervis Karaboga es un prominente académico e investigador en el campo de la ingeniería eléctrica y la ciencia de la computación. Ha hecho contribuciones significativas al desarrollo de algoritmos de optimización, particularmente en el área de los algoritmos metaheurísticos inspirados en la naturaleza. Karaboga es más conocido por desarrollar el Algoritmo de Colonia de Abejas Artificiales (ABC) en 2005, un método que ha sido ampliamente adoptado para resolver problemas de optimización complejos en diversas áreas como la ingeniería, la optimización de procesos y la inteligencia artificial.

Karaboga ha estado afiliado con la Universidad de Erciyes en Kayseri, Turquía, donde ha trabajado en el departamento de ingeniería informática. En esta institución, ha llevado a cabo una extensa investigación en algoritmos y computación inteligente, y ha supervisado a numerosos estudiantes de posgrado en proyectos relacionados con la optimización y la inteligencia artificial.

El impacto de su trabajo se extiende más allá del desarrollo del ABC, Karaboga ha explorado y refinado una variedad de técnicas relacionadas con la inteligencia de enjambre y otras formas de inteligencia computacional. Sus investigaciones han resultado en numerosas publicaciones en revistas y conferencias científicas de alto impacto, consolidando su reputación como uno de los líderes en el campo de los algoritmos bioinspirados.

Actualmente continúa su investigación y desarrollo en algoritmos avanzados de optimización, buscando nuevas aplicaciones y mejorando las existentes en la ciencia de datos y la ingeniería. [20], [21]

3.6.3- Variables

En el ABC existen F **fuentes de alimento**.

-La **posición** de la fuente es una solución del problema.

-La **calidad o fitness** de una fuente representa el néctar de dicha fuente.

-Las fuentes se agotan con el paso del tiempo. En el contexto del algoritmo se representa mediante una variable asociada a cada fuente que se irá incrementando su valor cuando se intenta mejorar esa fuente, pero no se consigue.

Como se ha mencionado antes existen tres tipos de abejas, las abejas **exploradoras**, que eligen las fuentes de alimento de forma aleatoria, las **empleadas**, que explotan la fuente de alimento y buscan cerca de esa fuente otra mejor, y las **observadoras** que analizan la información de las empleadas y eligen una de ellas para explotarla.

3.6.4- Fundamentos del algoritmo

Como punto de partida se pone a continuación el pseudocódigo 'resumido' del algoritmo.

Inicializar el conjunto de fuentes de alimento

REPETIR

Operaciones de abejas empleadas

Operaciones de abejas observadoras

Operaciones de abejas exploradoras

Actualizar la mejor solución hasta el momento

HASTA QUE (condición de parada)

El primer paso es **inicializar las fuentes de alimento**.

Se seleccionan las F fuentes de alimento iniciales y se les asigna una posición de forma aleatoria dentro del espacio de búsqueda del algoritmo y una variable (limit) que estará a 0 en este momento.

$$x_{ij} = x_j^{min} + \gamma\,(x_j^{max} - x_j^{min})$$

Dónde:

γ: número aleatorio en [0, 1]

Operaciones de abejas empleadas.

Hay una abeja empleada asociada a cada fuente de alimento.

Cada abeja busca alguna fuente próxima y en caso de que sea mejor que la actual, se va a esa fuente.

El pseudocódigo de este tipo de abejas sería:

Para cada abeja empleada i

    Obtener una fuente candidata v_i próxima a la actual x_i.

    Si fitness(v_i) < fitness(x_i)

        Moverse a la nueva fuente.

    Sino

        Incrementar en 1 $limit_i$

    Fin-si

Fin-para

Estas abejas para elegir la fuente candidata próxima también llamada fuente vecina (v_i), siguen esta ecuación:

$$v_i = x_i + \alpha_i (x_i - x_k)$$

Donde:

x_k: Fuente vecina elegida al azar con $k \neq i$

α_i: número aleatorio en [-1,1]

Operaciones de abejas observadoras.

Cada uno de estos individuos elige una fuente de alimento en función del fitness de todas las fuentes que están siendo explotadas por las abejas empleadas.

Una vez que se ha hecho esta selección, sigue como una abeja empleada.

Para cada abeja observadora

Elegir una fuente para explotar, en base a una probabilidad $\mathbf{p_i}$

Obtener una fuente candidata v_i próxima a la actual x_i.

Si fitness(v_i) < fitness(x_i)

Moverse a la nueva fuente.

Sino

Incrementar en 1 $limit_i$

Fin-si

Fin-para

La probabilidad $\mathbf{p_i}$ mencionada anteriormente es la probabilidad de selección de dicha fuente i, sigue la siguiente ecuación:

$$p_i = \frac{fitness_i}{\sum_j fitness_j}$$

$Fitness_i$: fitness de la fuente i-ésima

J: contador con valores de 1 y el número de fuentes.

Operaciones de abejas exploradoras.

Las fuentes de alimento agotadas son sustituidas por otras aleatorias, de igual manera que al comienzo del algoritmo.

La abeja empleada asociada a una fuente que se ha agotado se convierte en exploradora para elegir una nueva fuente.

Se eligen nuevos valores para una la i-ésima fuente, inicializando su posición y su variable limit.

Para la abeja empleada que estuviera asignada en una de esas fuentes agotadas se olvida la fuente x_y y se recuerda la v_i.

Por último, se actualiza la mejor solución hasta el momento, es decir, se guarda la posición de la fuente con mejor fitness.

4. Tecnologías empleadas

En este apartado se describirán las tecnologías que se han utilizado para el desarrollo de este proyecto.

4.1 Python

Lenguaje de alto nivel elegido para la implementación de los algoritmos de optimización por enjambre y procesamiento de imágenes debido a su facilidad de uso, su rica colección de bibliotecas científicas, y su soporte para la creación de entornos virtuales.

Fue creado a finales de los años ochenta por Guido van Rossum en Stichting Mathematisch Centrum (CWI), en Países Bajos, fue creado como sucesor del lenguaje ABC, capaz de manejar excepciones e interactuar con el sistema operativo Amoeba.[22], [23]

Es un lenguaje interpretado y de propósito general, conocido por su simpleza y legibilidad, lo que lo convierte en una opción popular tanto para principiantes como para desarrolladores experimentados. Se utiliza en una gran variedad de ámbitos, desde el desarrollo web hasta ciencia de datos, inteligencia artificial, blockchain …

Ilustración 5 Logo de python

4.2 Numphy

NumPy es una librería de Python especializada en el cálculo numérico y el análisis de datos, especialmente para un gran volumen de datos.

Incorpora una nueva clase de objetos llamados arrays que permite representar colecciones de datos de un mismo tipo en varias dimensiones, y funciones muy eficientes para su manipulación.

La ventaja de Numpy frente a las listas predefinidas en Python es que el procesamiento de los arrays se realiza mucho más rápido (hasta 50 veces más) que las listas, lo cual la hace ideal para el procesamiento de vectores y matrices de grandes dimensiones.[24]

Ilustración 6 Numphy

4.3 OpenCv

OpenCV es una librería de computación visual para el procesamiento de imágenes en Python. Esta biblioteca proporciona herramientas para realizar operaciones de procesamiento de imágenes, como el filtrado, la detección de bordes, el reconocimiento de características, el seguimiento de objetos, etc. Estas herramientas nos permiten desarrollar aplicaciones de visión artificial, como el reconocimiento facial, el seguimiento de objetos, etc. [25]

Esta librería ha sido muy útil en este proyecto debido a que permite el procesamiento y manipulación de imágenes. Es fundamental para la cuantificación de color en imágenes.

4.4 Scikit-learn

Es una biblioteca de Python que proporciona acceso a versiones eficaces de muchos algoritmos comunes. También proporciona una API propia y estandarizada. Por tanto, una de las grandes ventajas de Scikit-Learn es que una vez que se entiende el uso básico y su sintaxis para un tipo de modelo, cambiar a un nuevo modelo o algoritmo es muy sencillo. La biblioteca no solo permite hacer el modelado, sino que también puede garantizar los pasos de preprocesamiento.

La gran variedad de algoritmos y utilidades de Scikit-learn la convierten en la herramienta básica para empezar a programar y estructurar los sistemas de análisis datos y modelado estadístico. Los algoritmos de Scikit-Learn se combinan y depuran con otras estructuras de datos y aplicaciones externas como Pandas o PyBrain.

La ventaja de la programación en Python, y Scikit-Learn en concreto, es la variedad de módulos y algoritmos que facilitan el aprendizaje y trabajo del científico de datos en las primeras fases de su desarrollo.[26]

4.4 Scikit-image

Esta es otra librería de procesamiento de imágenes de código abierto que está diseñada para operar con las bibliotecas numéricas y científicas de Python como Numpy o SciPy.

El proyecto scikit-image comenzó como scikits.image, de Stéfan van der Walt. Su nombre proviene de la idea de que es un "SciKit" (SciPy Toolkit), una extensión de terceros desarrollada y distribuida por separado para SciPy. El código base original fue posteriormente reescrito en profundidad por otros desarrolladores. De los diversos scikits, scikit-image y scikit-learn fueron descritos como "bien mantenidos y populares" en noviembre de 2012. [27]

4.5 Entornos virtuales

Esta característica me ha permitido aislar las dependencias del proyecto, asegurando que las versiones correctas de las librerías se utilicen sin conflictos con otros proyectos.

Una vez que se tiene el entorno creado hay que activarlo para poder ejecutar el programa, esto se hace con source entorno/bin/activate

4.5 Control de versiones

Para el desarrollo del proyecto se ha utilizado en todo momento la herramienta de GitHub lo que me ha permitido tener un control visual de los cambios que realizaba en el código y poder probar en diferentes entornos de trabajo con rapidez.

Ilustración 7 GitHub

4.6 Visual Studio Code

Ha sido el editor de código que he usado en todo el proyecto, permitiéndome la integración de los entornos virtuales de Python y la depuración del código.

También me facilitó una manera de pasarle diferentes argumentos requeridos por el programa desde el propio editor formando un JSON con todos ellos.

Ilustración 8 Visual Studio Code

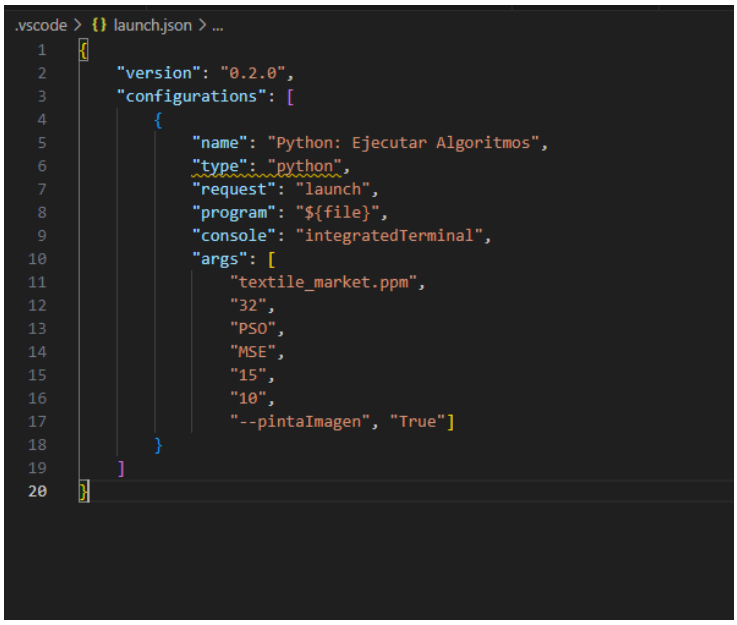


Ilustración 9 JSON de argumentos

4.7 Otros

Aparte de todo lo anterior se realizó un script en bash para automatizar el proceso de ejecución de los algoritmos y la generación de resultados.

Dicho script lanza el programa con diferentes argumentos y vuelca los resultados en diferentes txt para recuperar los datos para formar los resultados. En estas sucesivas llamadas al programa se cambia el algoritmo que se usa, la función con la calcula el fitness y también las imágenes que procesa.

5. Desarrollo del programa

Para el desarrollo como se ha mencionado antes se realizó un programa en Python que realiza las operaciones de los algoritmos de enjambre y genera imágenes cuantizadas con las que calcula un error con 4 índices de error diferentes (MAE, MSE, SSIM, MS-SIM), este programa se llama con diferentes argumentos que se explicarán en este apartado.

Por otro lado, también se realizó un script para la automatización de la obtención de resultados que se encarga de llamar al programa repetidas veces para ejecutarlo con todos los algoritmos, todas las funciones y 4 paletas de colores diferentes, estas 4 paletas son de 32 colores, de 64, de 128 y de 256.

5.1 Archivo main o ejecutor

Este archivo es el principal del programa encargado de recibir los argumentos que se le pasen y de elegir o "mapear" estos argumentos con el algoritmo que toque junto con la función que toque.

Para recibir estos argumentos se ha utilizado la librería por defecto de Python llamada argparse.

```
parser= argparse.ArgumentParser()

#Agrego argumento de la imagen
parser.add_argument('imagen', type=str, help="Nombre de la imagen a procesar")
parser.add_argument('numeroColores', type=int, help="Numero de colores de la nueva imagen")
parser.add_argument('algoritmo', type=str, help="Algoritmo a procesar")
parser.add_argument('funcion', type=str, help="Funcion a usar por algoritmo (ej. MSE)")
parser.add_argument('iteraciones', type=int, help="Numero de iteraciones que realizara el algoritmo")
parser.add_argument('individuos', type=int, help="Numero de individuos del algoritmo")
parser.add_argument('--pintaImagen', type=bool, default=False,help="Argumento para saber si pinta imagen al final del algoritmo
args = parser.parse_args()
```

Ilustración 10 Argumentos

En la ilustración 10 se ve como se añaden los argumentos de tipo cadena o de tipo entero, estos argumentos por orden son:

- Imagen: imagen que se va a procesar en el algoritmo.
- NumeroColores: el numero de colores de la nueva imagen o la paleta que se va a usar.
- Algoritmo: algoritmo que usará el programa para la cuantificación de color. Estos pueden ser: PSO, FA, BA, GWO, ABA que respectivamente son: pso, luciérnagas, ballenas, lobos y abejas.
- Función: función que calculará el fitness de cada iteración del algoritmo, las opciones son: MSE, MAE, SSIM, MSSIM.
- Iteraciones: numero entero que representa el numero de iteraciones que realizara el algoritmo escogido.
- Individuos: numero entero que representa la población de individuos que realizarán el algoritmo.
- PintaImagen: este argumento booleano solo lo usé para mis pruebas, si viene a True el programa mostrará por pantalla las dos imágenes, tanto la original como la resultante, me fue útil al principio para comprobar visualmente que la imagen resultante realmente estaba modificada.

```
# Obtener la ruta completa de la imagen
ruta_imagen = os.path.join(os.path.dirname(__file__), 'images', args.imagen)
```

Ilustración 11 Obtener ruta de la imagen

En la ilustración 11 se ve la línea que obtiene la ruta de la imagen que se le ha pasado. Se ha usado el módulo os para obtenerla de manera que funcione bien en cualquier sistema operativo. Esta operación se realiza debido a que las imágenes que se procesan en el algoritmo están en una carpeta images que está ubicada al mismo nivel que el archivo ejecutor, si estuviesen al mismo nivel que ejecutor no haría falta, pero esto sería más caótico. args.imagen es la forma de obtener la imagen que se le ha pasado como argumento.

```python
# Verificar si el archivo existe
if not os.path.isfile(ruta_imagen):
    print(f"Error: No se puede encontrar el archivo '{ruta_imagen}'")
    quit()

# Intentar leer la imagen con OpenCV
img = cv2.imread(ruta_imagen, cv2.IMREAD_COLOR)
if img is None:
    print(f"Error: No se puede abrir o leer el archivo '{ruta_imagen}'")
    quit()
```

Ilustración 12 Comprobaciones

En la ilustración 12 se ven algunas comprobaciones necesarias por si no se pudiese encontrar el archivo en la carpeta images y la comprobación de que se puede leer o abrir la imagen pasada.

```python
match args.algoritmo:
    case "PSO":
        #PSO
        matchFuncionPso(args, ruta_imagen, args.individuos, args.iteraciones)
    case "FA":
        #Luciernagas
        matchFuncionFa(args, ruta_imagen, args.individuos, args.iteraciones)
    case "BA":
        #Ballenas
        matchFuncionBa(args, ruta_imagen, args.individuos, args.iteraciones)
    case "GWO":
        #Lobos
        matchFuncionGwo(args, ruta_imagen, args.individuos, args.iteraciones)
    case "ABA":
        #Abejas
        matchFuncionAba(args, ruta_imagen, args.individuos, args.iteraciones)
    case _:
            print("Algoritmo no reconocido")
```

Ilustración 13 Cuerpo del programa

En la ilustración 13 se muestra el cuerpo del programa, donde se evalúa que algoritmo ha pedido el usuario y se llama a la función que mapea que función va a usar el propio algoritmo.

```
def matchFuncionAba(args, ruta_imagen, individuos, iteraciones):
    match(args.funcion):
        case "MSE":
            alh=SwarmPackagePy.aba(individuos, func.getMse, 0, 255, 3, iteraciones,args.numeroColores,args.pintaImagen,imagen=ruta_imagen)
        case "MAE":
            alh=SwarmPackagePy.aba(individuos, func.getMae, 0, 255, 3, iteraciones,args.numeroColores,args.pintaImagen,imagen=ruta_imagen)
        case "SSIM":
            alh=SwarmPackagePy.aba(individuos, func.getSsim, 0, 255, 3, iteraciones,args.numeroColores,args.pintaImagen,imagen=ruta_imagen)
        case "MSSIM":
            alh=SwarmPackagePy.aba(individuos, func.getMsSsim, 0, 255, 3, iteraciones,args.numeroColores,args.pintaImagen,imagen=ruta_image
```

Ilustración 14 Mapeo de funciones

En la ilustración 14 se ve una de las funciones que mapean que función va a usar el algoritmo y llama al algoritmo que se haya pedido con la función que se necesite.

Todas estas operaciones constituyen la primera parte del programa, donde se reciben argumentos, se comprueban y se “escoge” el algoritmo a usar junto a la función seleccionada.

5.2 Algoritmos

A continuación, se explica el desarrollo de los algoritmos utilizados.

5.2.1 PSO

```
import os
import numpy as np
from . import intelligence
from . import misfunciones as fn

import sys

# Limites para la velocidad de la particula
V_MAX = 4
V_MIN = -4
```

Ilustración 15 Comienzo PSO

En la ilustración 15 se muestran los imports de este algoritmo y dos variables globales que definen el comportamiento de la velocidad máxima y mínima para cada individuo del algoritmo en este caso 4 y -4.

```
# Clase para el PSO (hereda de intelligence)
class pso(intelligence.sw):
    """
    Particle Swarm Optimization

    """

    # Constructor para el pso
    def __init__(self, n, function, lb, ub, dimension, iteration,numeroColores,pintor, w=0.5, c1=1,
                 c2=1, imagen=""):
        """
```

Ilustración 16 Constructor PSO

En la ilustración 16 comienza la clase que define el PSO y su constructor, sus parámetros son los siguientes:

- n: número de individuos.
- function: función que se aplica en el algoritmo.
- lb: límite inferior del espacio de búsqueda.
- ub: límite superior del espacio de búsqueda.
- dimension: dimensión del espacio de solución (r).
- iteration: número de iteraciones.
- numeroColores: número de colores de la nueva imagen.
- pintor: booleano que se usa para saber si pintamos imagen al final o no (Es el argumento pintaImagen del programa).
- w: parámetro inercia.
- c1: parámetro cognitivo.
- c2: paramero social.
- imagen: ruta de la imagen a procesar por el algoritmo.

```
# Empezamos a inicializar la poblacion de particulas con su velocidad y posicion
# LLama al constructor de intelligence, así inicializa las
# posiciones y la mejor posicion.
super(pso, self).__init__()

# generamos numeros aletaorios uniformemente distribuidos.
# Generamos entre lb y ub (limite inferior y limite superior)
# Generamos un total de n*dimension numeros
self.__agents = np.random.uniform(lb, ub, (n,numeroColores, dimension))

velocity = np.zeros((n,numeroColores, dimension)) #Llenamos el vector de velocidad con 0
self._points(self.__agents)

# Inicializar Pbest, es decir, inicialmente las mejores posiciones de las particulas son las
# primeras halladas.
Pbest = self.__agents


# Iniciamos Gbest con el valor de la particula con menor fitness
Gbest = Pbest[np.array([function(x,numeroColores,imagen) for x in Pbest]).argmin()]
# hasta aqui hemos inicializado el PSO
```

Ilustración 17 Inicialización PSO

En la anterior ilustración se ve como se inicializa cada uno de los parámetros.

Gracias al constructor de intelligence se inicializan posiciones, mejores posiciones y mejor fitness, todo a 0 en un principio (Ilustración 18)

También se generan números aleatorios gracias a la librería numpy de manera uniforme y aleatoria, esto da las primeras posiciones de cada individuo.

Se inicializa el vector de velocidad a 0 y las mejores posiciones de los individuos que son las primeras que obtuvimos antes.

Por ultimo se inicializa la mejor posición global llamando a nuestra función objetivo (MSE, MAE, SSIM o MSSIM) para cada una de las

posiciones de cada individuo, el resultado de esta función es el fitness de cada individuo por lo que con argmin() obtenemos el menor resultado, es decir, la mejor posición de todos los individuos.

```
# Define las posiciones y las mejores posiciones
def __init__(self):
    self.__Positions = []  #Se inicializa una lista vacia para almacenar las posiciones de los individuos
    self.__Gbest = []      #Se inicializa una lista vacía para almacenar las mejor posicion global
    self.mejorFitnes = 0   #Se inicializa el mejor fitness a 0
```

Ilustración 18 Constructor de Intelligence

```
67          for t in range(iteration):
68              #print("Iteración ", t+1)
69              """
70              ESQUEMA PSO
71              1.1- evaluar fitness de cada particula
72              1.2 - actualizar mejor solucion personal de cada particula
73              1.3 - actualizar la mejor solucion global
74
75              2 - Actualizar velocidad y posicion de cada particula
76              3 - mostrar resultado al acabar bucle
77              """
78              r1 = np.random.rand(n,numeroColores,dimension)
79              r2 = np.random.rand(n,numeroColores,dimension)
80
81              #  Calculamos la nueva velocidad
82              velocity = w * velocity + c1 * r1 * (
83                   Pbest - self.__agents) + c2 * r2 * (
84                   Gbest - self.__agents)
85              # Ajustamos la velocidad para que no se salga de los limites.
86              velocity= np.clip(velocity, V_MIN, V_MAX)
87              #print(velocity)
88
89              #Ajustamos la posicion de las particulas sumando la velocidad
90              self.__agents += velocity
91              # Ajusta esta posicion a los limites del espacio
92              self.__agents = np.clip(self.__agents, lb, ub)
93              #Y lo convertimos en lista
94              self._points(self.__agents)
95
```

Ilustración 19 Comienzo bucle PSO

En la ilustración 19 comienza el bucle del PSO, es decir, las instrucciones que va a realizar el algoritmo en cada una de las iteraciones.

Comienza por calcular r1 y r2 llamando a numpy para obtener una matriz de números aleatorios entre 0 y 1 (np.random.rand()) de tamaño (n, numeroColores, dimension), donde para cada individuo n se generan numeroColores valores aleatorios en un espacio de dimension dimensiones, es decir, si tenemos 10 individuos y una paleta de colores de 64 el resultado seria una matriz de tamaño (10, 64, 3) y sus valores serian números aleatorios entre 0 y 1.

En la línea 62 se calcula la velocidad siguiendo la formula de la velocidad del PSO definida en la página 19, después se ajusta esta velocidad con clip() a los valores permitidos (entre -4 y 4).

Se ajusta la posición de las partículas sumando la velocidad, se acota esta posición a los límites del espacio permitidos y por ultimo se convierte en una lista (Ilustración 20).

```
#Convierte a las particulas pasadas (posiciones) en una lista
def _points(self, agents):
    self.__Positions.append([list(i) for i in agents])
```

Ilustración 20 Método de Intelligence para paso a lista

```
#Actualizar mejor solucion particular
#Para todas las particulas ...
for i in range(n):
   # Si el fitness de esta posicion es menor que el almacenado lo actualizamos
   if(function(self.__agents[i],numeroColores,imagen) < function(Pbest[i],numeroColores,imagen)):
      Pbest[i] = self.__agents[i]
```

Ilustración 21 Actualizacion mejor solucion particular PSO

En esta ilustración se muestra un bucle que se ejecuta tantas veces como individuos tenga el algoritmo en el que se actualiza la mejor solución de cada individuo si se cumple una condición, esta condición es que el resultado obtenido por la función objetivo con la nueva posición del individuo sea menor que su mejor posición particular almacenada, de esta manera se mantiene siempre la mejor solución para cada individuo.

```
# Actualizar mejor solucion global
Gbest = Pbest[np.array([function(x,numeroColores,imagen) for x in Pbest]).argmin()]

self.setMejorFitness(function(Gbest,numeroColores,imagen))
print(self.getMejorFitness(), end= ' ')
```

Ilustración 22 Actualizacion mejor solucion global PSO

Por ultimo en el bucle, según como se muestra en la ilustración 22 se actualiza la mejor solución global recorriendo todas las mejores soluciones particulares obtenidas por la función objetivo y cogiendo el menor valor.

Se actualiza el mejor fitness hallado y se imprime.

```
        ##########################################################################################################
        #Guardamos la mejor solucion encontrada por el algoritmo
         Gbest = np.int_(Gbest)
         self._set_Gbest(Gbest)
         # Generamos la imagen cuantizada para imprimirla con el mejor valor final global.
         reducida = fn.generaCuantizada(Gbest,numeroColores,imagen)

         #Pintamos imagen
         fn.pintaImagen(reducida, imagen,pintor,"PSO", numeroColores)
```

Ilustración 23 Fin PSO

En esta ilustración se muestra el final del PSO en el que se guarda la mejor solución global obtenida por el algoritmo, y se genera una imagen cuantizada con esa mejor solución (mejor paleta de colores) para luego pintarla. Esta última generación de cuantizada no hace falta si no se pinta.

5.2.2 Luciérnagas

De igual manera que en el anterior algoritmo se comienza con el constructor.

```
def __init__(self, n, function, lb, ub, dimension, iteration,numeroColores,pintor, csi=1, psi=1,
             alpha0=1, alpha1=0.1, norm0=0, norm1=0.1,imagen=""):
```

Ilustración 24 Constructor Luciernagas

En esta ilustración se muestra el constructor cuyos parámetros son:

- n: número de partículas
- function: función a optimizar
- lb: límite inferior del espacio (0 para imágenes)
- ub: límite superior del espacio (255 para imágenes)
- dimension: dimensiones del espacio
- iteration: número de iteraciones
- numeroColores: número de colores de la nueva imagen
- pintor: booleano que se usa para saber si pintamos imagen al final o no.
- csi: atracción mutua
- psi: Coeficiente de absorción de la luz del medio
- alpha0: valor inicial del parámetro aleatorio alpha
- alpha1: valor final del parámetro aleatorio alpha
- norm0: primer parámetro para una distribución normal (Gaussiana)
- norm1: segundo parámetro para una distribución normal (Gaussiana)
- imagen: ruta de la imagen a procesar por el algoritmo

```
super(fa, self).__init__()

# Inicia la poblacion de luciernagas
self.__agents = np.random.uniform(lb, ub, (n,numeroColores, dimension))
self._points(self.__agents)

# Iniciamos las mejores posiciones de cada particula con las calculadas antes
Pbest = self.__agents

# Calculamos el fitness de las mejores posiciones encontradas por las luciernagas
fitnessP = [function(x,numeroColores,imagen) for x in self.__agents]
fitnessA = fitnessP # Lo igualamos al de la posicion ACTUAL
```

Ilustración 25 Inicio Luciernagas

Se continúa inicializando intelligence, la población de individuos y las mejores posiciones de cada individuo de la misma manera que en el anterior algoritmo.

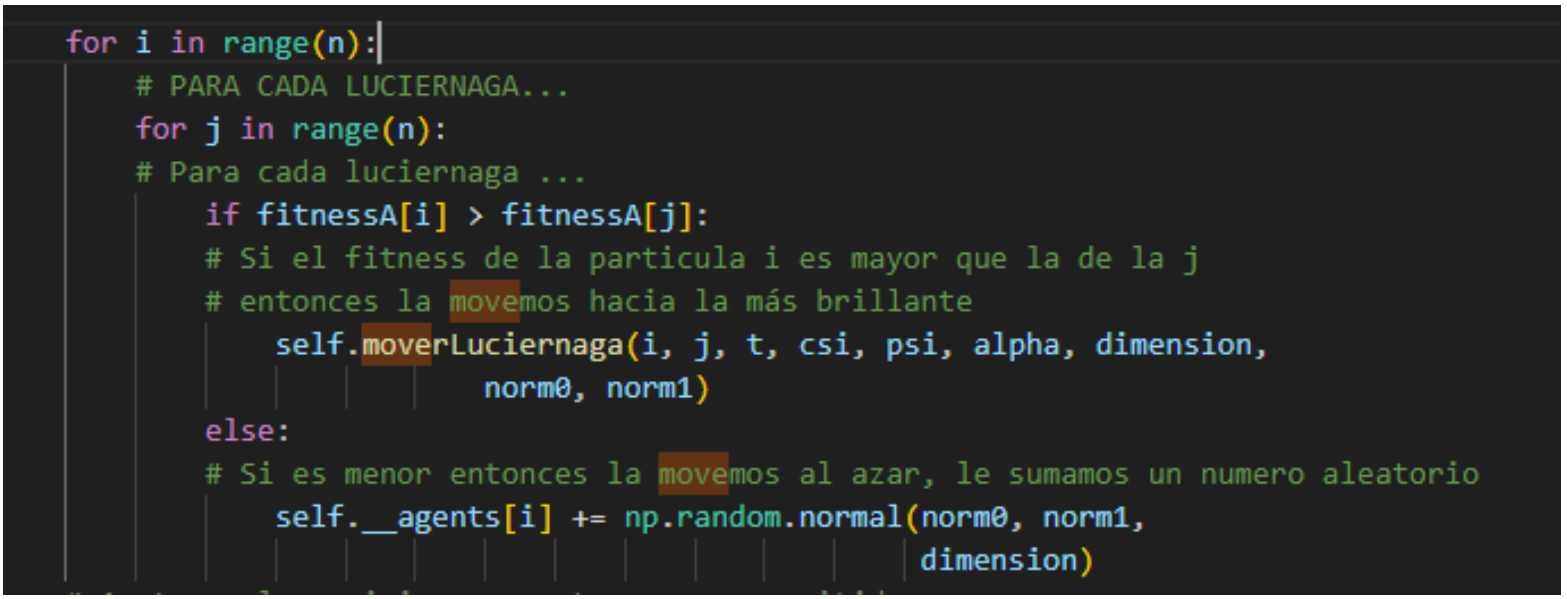
Por último, se calculan todos los fitness, fitnessP representa los mejores valores de fitness (mejores posiciones) que se han encontrado y fitnessA el fitness actual, como se esta inicializando no importa que no sea del todo correcto ya que en un principio tanto la mejor posición como el mejor fitness coincide con la primera posición y el primer fitness hallado.

```
for t in range(iteration):
    #print("Iteración ", t+1)
    # Esto se usa en la funcion mover para el calculo de un numero aleatorio
    alpha = alpha1 + (alpha0 - alpha1) * exp(-t)
```

Ilustración 26 Comienzo del bucle Luciernagas

En esta última ilustración se comienza el bucle del algoritmo, y se calcula alpha que será necesario para la funcion de mover individuos.

Alpha = valor inicial del parámetro alpha + (valor final del parámetro alpha) * exponcial(-numero iteracion).



```
for i in range(n):
    # PARA CADA LUCIERNAGA...
    for j in range(n):
    # Para cada luciernaga ...
        if fitnessA[i] > fitnessA[j]:
        # Si el fitness de la particula i es mayor que la de la j
        # entonces la movemos hacia la más brillante
            self.moverLuciernaga(i, j, t, csi, psi, alpha, dimension,
                        norm0, norm1)
        else:
        # Si es menor entonces la movemos al azar, le sumamos un numero aleatorio
            self.__agents[i] += np.random.normal(norm0, norm1,
                                                 dimension)
```

Ilustración 27 Movimientos de luciernagas

En esta ilustración se muestran dos bucles anidados que recorren cada individuo y por cada individuo se recorre de nuevo cada individuo en el algoritmo por lo que se compara cada individuo con todos y cada uno de los individuos. Se hace para que si el fitness actual de un individuo i es mayor el que el de un individuo j se mueve este individuo i hacia el individuo j ya que el individuo j es mas atractivo, si no cumple esa condición se mueve aleatoriamente por el espacio.

Estas operaciones se realizan para mover cada luciérnaga a las que son mas brillantes que ellas para realizar una explotación global.

```
# Acotamos la posicion a nuestro rango permitido
self.__agents = np.clip(self.__agents, lb, ub)
self._points(self.__agents)
# Luciernagas movidas hasta aqui
#Calculamos el fitness actual de cada luciernaga
fitnessA = [function(x,numeroColores,imagen) for x in self.__agents]
```

Ilustración 28 Cálculos Luciérnagas

En esta ilustración se ve como se acota la posición al rango permitido y se transforma a lista (de igual manera que en el algoritmo anterior).

Con estos pasos concluye el movimiento de luciérnagas por lo que se puede empezar a calcular el fitness actual (fitnessA) de manera igual que en el PSO, para cada individuo se llama a la funcion objetivo para obtener el fitness.

```
# Actualizar la mejor solucion particular
# Si el fitness de la posicion actual es menor (mejor) que el mejor guardado ...
for i in range(n):
#Para cada luciernaga
    if(fitnessA[i] < fitnessP[i]):
    # Actualizamos su posicion y su mejor fitness
        Pbest[i] = self.__agents[i] # posicion
        fitnessP[i] = function(Pbest[i],numeroColores,imagen) # fitness
```

Ilustración 29 Actualización posición y fitness luciérnagas

En la ilustración 29 se muestra como se actualiza tanto la mejor solución particular de cada individuo y su mejor fitness.

```
# Actualizar mejor solucion global
Gbest = Pbest[np.array([function(x,numeroColores,imagen) for x in Pbest]).argmin()]

self.setMejorFitness(function(Gbest,numeroColores,imagen))
print(self.getMejorFitness(), end= ' ')
```

Ilustración 30 Actualización mejor solución global luciérnagas

También se actualiza la mejor solución global y el mejor fitness, por ultimo se imprime este valor.

```
# Esta funcion mueve luciernagas ...
def moverLuciernaga(self, i, j, t, csi, psi, alpha, dimension, norm0, norm1):

    # Calculo de la distancia entre dos luciernagas
    r = np.linalg.norm(self.__agents[i] - self.__agents[j])
    # Calculamos el ATRACTIVO de la luciernga
    # beta = atraccion mutua /
            # (1 + coeficiente de absorcion de la luz del medio * distancia luciernagas al cuadrado)
    beta = csi / (1 + psi * r ** 2) # atractivo de la luciernaga i sobre k


# Calculamos la nueva posicion aplicando esta formula
# fi(t+1) = fj(t) + beta(rik) * (fi(t) - fj(t)) + alpha * exp(-t) * aleatorio
# el aleatorio se calcula usando una distribucion normal Gaussiana, entre los limites propuestos y con la
# dimension dicha
    self.__agents[i] = self.__agents[j] + beta * (
        self.__agents[i] - self.__agents[j]) + alpha * exp(-t) * \
                                               np.random.normal(norm0,
                                                                norm1,
                                                                dimension)
```

Ilustración 31 Función mover luciérnaga

En esta última ilustración se muestra como se mueven las luciérnagas siguiendo las fórmulas que están en los comentarios.

Para el calculo de la distancia (r) se utiliza la biblioteca numpy, en este caso se calcula la distancia euclidiana que es la distancia recta entre dos puntos en el espacio.

Esta función devuelve las nuevas posiciones de cada individuo.

5.2.3 Lobos

```
def __init__(self, n, function, lb, ub, dimension, iteration, numeroColores ,pintor,imagen=""):
    """
    :param n: numero de individuos
    :param function: funcion del algoritmo
    :param lb: limite inferior del espacio de busqueda
    :param ub: limite superior del espacio de busqueda
    :param dimension: dimension del espacio
    :param iteration: numero de iteraciones
    """

    super(gwo, self).__init__()

    #Inicio de la poblacion
    self.__agents = np.random.uniform(lb, ub, (n,numeroColores, dimension))
    self._points(self.__agents)
    #Buscamos los mejores lobos
    alpha, beta, delta = self.getABD(n, function, numeroColores, imagen)

    Gbest = alpha
```

Ilustración 32 Inicialización Lobos

Como en el resto de los algoritmos se empieza con el constructor del algoritmo e inicializando la población.

En este algoritmo los parámetros son:

- n: numero de individuos
- function: función del algoritmo
- lb: limite inferior del espacio de búsqueda
- ub: limite superior del espacio de búsqueda
- dimension: dimensión del espacio
- iteration: número de iteraciones

Después de inicializar intelligence y la población se buscan los mejores lobos (alfa beta y delta ABD), alpha representa el individuo con mejor posición por lo cual se asigna Gbest (mejor posición global) con esta posición.

```python
def getABD(self, n, function, numeroColores, imagen):

    result = []

    # Calcula el fitness de cada agente y los guarda junto con su índice en una lista de tuplas
    fitness = [(function(self.__agents[i],  numeroColores,imagen), i) for i in range(n)]

    # Ordena la lista de fitness en orden ascendente (menor fitness es mejor)
    fitness.sort()

    # Selecciona los tres agentes con mejor fitness
    for i in range(3):
        result.append(self.__agents[fitness[i][1]])

    return result
```

Ilustración 33 Get mejores lobos

Esta función retorna los tres mejores individuos de todos los individuos del algoritmo. Para ello primero inicializa result como una lista vacia, después se calcula el fitnes de cada individuo y se guarda en una lista junto con su índice. Después se ordena esta lista en orden ascendente.

Por ultimo se recorren los tres mejores individuos para añadirlos a result, se selecciona de la lista de individuos el individuo cuyo índice coincide con el índice almacenado en la lista fitness.

```python
    for t in range(iteration):
        #print("Iteración ", t+1)
        #Actualizo el parámetro del algoritmo (a)
        a = 2 - 2 * t / iteration
```

Ilustración 34 Inicio bucle Lobos

En esta ilustración se muestra el inicio del bucle del algoritmo y el calculo de a siguiendo la ecuación de la página 34:

$$\mathrm{a} = 2 - 2\frac{t}{max}$$

```
#Cálculo de los vectores aleatorios entre [0, 1], y de A y C para lobo alpha
r1 = np.random.rand(n,numeroColores,dimension)
r2 = np.random.rand(n,numeroColores,dimension)
A1 = 2 * r1 * a - a
C1 = 2 * r2

#Cálculo de los vectores aleatorios entre [0, 1], y de A y C para lobo beta
r1 = np.random.rand(n,numeroColores,dimension)
r2 = np.random.rand(n,numeroColores,dimension)
A2 = 2 * r1 * a - a
C2 = 2 * r2


#Cálculo de los vectores aleatorios entre [0, 1], y de A y C para lobo delta
r1 = np.random.rand(n,numeroColores,dimension)
r2 = np.random.rand(n,numeroColores,dimension)
A3 = 2 * r1 * a - a
C3 = 2 * r2
```

Ilustración 35 Calculo de aleatorios, A y C

En esta ilustración se muestra el cálculo de números aleatorios para hallar A y C de cada individuo siguiendo las ecuaciones 1 y 2 de la página 31.

```
#Cálculo de D (Para los lobos alpha, beta y delta)
Dalpha = abs(C1 * alpha - self.__agents)
Dbeta = abs(C2 * beta - self.__agents)
Ddelta = abs(C3 * delta - self.__agents)
```

Ilustración 36 Cálculo de D

Se calculan las D de cada individuo alfa beta y delta siguiendo las ecuaciones de la pagina 33.

```
#Cálculo de X para la nueva posicion
X1 = alpha - A1 * Dalpha
X2 = beta - A2 * Dbeta
X3 = delta - A3 * Ddelta
```

Ilustración 37 Cálculo de X para cada lobo

En esta ilustración se muestra el calculo de X para cada individuo siguiendo las ecuaciones de la pagina 33.

```
#Nueva posicion de cada lobo
self.__agents = (X1 + X2 + X3) / 3

#Ajuste de estas nuevas posiciones al limite del espacio
self.__agents = np.clip(self.__agents, lb, ub)
self._points(self.__agents)

#Cálculo de los lobos alfa, beta y delta
alpha, beta, delta = self.getABD(n, function, numeroColores, imagen)
#Cálculo de Gbest (Mejor solucion)
if function(alpha, numeroColores,imagen) < function(Gbest, numeroColores,imagen):
    Gbest = alpha
#Conseguimos el mejor fitness y lo mostramos en pantalla
self.setMejorFitness(function(Gbest,  numeroColores,imagen))
print(self.getMejorFitness(), end= ' ')
```

Ilustración 38 Final bucle lobos

Se ajustan las posiciones de los individuos siguiendo la ecuación de la página 33.

Se acotan las nuevas posiciones de cada individuo a los límites permitidos.

Llamando de nuevo a la función getABD() se obtienen los tres mejores individuos.

Después se calcula la mejor posición global comprobando si la posición actual del mejor individuo es mejor que la almacenada como mejor posición global hasta el momento, si esto se cumple se asigna el valor del mejor individuo a Gbest (mejor posición global).

Por ultimo se guarda el mejor fitness y se imprime.

5.2.4 Abejas

```
def __init__(self, n, function, lb, ub, dimension, iteration, numeroColores ,pintor,imagen=""):
```

Ilustración 39 Constructor abejas

Aquí se muestra el constructor de abejas, estos son sus parámetros:

- n: número de individuos
- function: función
- lb: límite inferior del espacio
- ub: límite superior del espacio
- dimension: dimensión del espacio
- iteration: número de iteraciones

```
super(aba, self).__init__()

#Iniciamos la poblacion y la pasamos a una lista
self.__agents = np.random.uniform(lb, ub, (n,numeroColores, dimension))
self._points(self.__agents)

#Búsqueda de la mejor solucion personal de cada individuo, para devolve
Pbest = self.__agents[np.array([function(x,numeroColores,imagen)
                                for x in self.__agents]).argmin()]
Gbest = Pbest
```

Ilustración 40 inicialización abejas

En esta ilustración se realiza la inicialización del algoritmo, como en el resto se inicia intelligence y la población de individuos.

Se busca la mejor solución de la misma manera que en los anteriores y se asigna a Gbest.

```
#Division de los individuos por grupos basandose en el numero de individuos presentes en el algoritmo
if n <= 10:
    #Si el conjunto total de individuos es menor que 10...
    #Se divide en un grupo grande n -(n//10) ((Division entera))
    count = n - n // 2, 1, 1, 1
else:
    #Si no se dividen en estos grupos:
    a = n // 10
    b = 5
    c = (n - a * b - a) // 2
    d = 2
    count = a, b, c, d
```

Ilustración 41División de individuos

Se dividen los individuos en varios grupos, si el numero de individuos es igual o menor a 10 se dividen en 4 grupos el primero que será aproximadamente la mitad y tres grupos de 1.

Si es mayor a 10 se dividen en 4, el primero es un 10% de los individuos, el segundo son 5 individuos, el tercero que será (numero de individuos - individuos en el primer grupo * individuos en el segundo grupo – individuos en el primer grupo) / 2, y un cuarto grupo de dos individuos.

Esta división se hace para que el algoritmo pueda explorar diferentes áreas del espacio de búsqueda de manera más eficiente. Los grupos grandes tienden a centrarse más en la explotación de soluciones mientras que los mas pequeños se enfocan en la exploración.

```
for t in range(iteration):
    #print("Iteración ", t+1)
    #Calculo del fitness de cada individuo
    fitness = [function(x,numeroColores, imagen) for x in self.__agents]
    # Ordenación de los índices basados en los valores de fitness
    sorted_indices = np.argsort(fitness)

    best_indices = sorted_indices[:count[0]]
    selected_indices = sorted_indices[count[0]:count[0] + count[2]]
```

Ilustración 42 Inicio del bucle abejas

Se comienza el bucle del algoritmo calculando el fitness de cada individuo y almacenándolo en una lista.

Después se ordenan de menor a mayor y se almacenan sus índices en sorted_indices gracias a argsort de Numphy.

Best_indices representa los índices de los individuos con los mejores valores de fitness en la población. Estos son los agentes que se consideran las fuentes de alimento más ricas, y son explotados intensamente por las abejas.

Selected_indices representa los índices de otro conjunto de individuos en la población. Estas no son las mejores soluciones, pero siguen siendo importantes, y se consideran como fuentes de alimento adicionales que las abejas observadoras pueden explorar.

```
#Se generan nuevos individuos a partir de los mejores y los otros
newbee = self.__new(best, count[1], lb, ub) + self.__new(selected,
                                                         count[3],
                                                         lb, ub)
m = len(newbee)
```

Ilustración 43 Nuevos individuos

Se generan nuevos individuos gracias a __new() y se obtiene el numero de individuos nuevos.

```
def __new(self, l, c, lb, ub):
    bee = []
    for i in l:
        new = [self.__neighbor(i, lb, ub) for k in range(c)]
        bee += new
    bee += l

    return bee
```

Ilustración 44 función para nuevos individuos

Esta función genera nuevos individuos para cada individuo en l moviéndose a posiciones vecinas.

```
def __neighbor(self, who, lb, ub):

    neighbor = np.array(who) + uniform(-1, 1) * (
        np.array(who) - np.array(
            self.__agents[randint(0, len(self.__agents) - 1)]))
    neighbor = np.clip(neighbor, lb, ub)

    return list(neighbor)
```

Ilustración 45 función vecinos

Genera un vecino aleatorio para un individuo who ajustando su posición a los límites.

```
if n - m > 0:
    #Se actualiza la poblacion de individuos y se ajustan sus posiciones a los limites de busqueda
    additional_bees = list(np.random.uniform(lb, ub, (n - m, numeroColores, dimension)))
    self.__agents = newbee + additional_bees
else:
    #Se actualiza la poblacion de individuos
    self.__agents = newbee[:n]
```

Ilustración 46 Comprobación abejas

Se comprueba si n-m es positivo antes de generar nuevos individuos y se ajustan sus posiciones a los límites de búsqueda.

```
self.__agents = np.clip(self.__agents, lb, ub)
self._points(self.__agents)

#Se actualiza la mejor solucion personal de cada individuo y luego la mejor solucion global
Pbest = self.__agents[
    np.array([function(x,numeroColores, imagen) for x in self.__agents]).argmin()]
if function(Pbest,numeroColores, imagen) < function(Gbest,numeroColores, imagen):
    Gbest = Pbest

#Set del mejor fitness e impresion del fitnes de la iteracion
self.setMejorFitness(function(Gbest,numeroColores, imagen))
print(self.getMejorFitness(), end= ' ')
```

Ilustración 47 Fin bucle abejas

En esta ilustración se muestra el final del bucle del algoritmo en el que se ajustan las posiciones a los limites del espacio de búsqueda de los individuos.

Se actualiza la mejor solución personal de cada individuo.

Se actualiza la mejor solución global si se cumple la condición.

Por ultimo se guarda el valor del mejor fitness y se imprime.

5.2.5 Ballenas

```
def __init__(self, n, function, lb, ub, dimension, iteration,numeroColores,pintor, ro0=2,
             eta=0.005,imagen=""):
```

Ilustración 48 Constructor ballenas

En esta ilustración se muestra el constructor del algoritmo cuyos parámetros son:

- n: número de ballenas
- function: función a optimizar
- lb: límite inferior del espacio
- ub: límite superior del espacio
- dimension: dimensión del espacio
- iteration: número de iteraciones
- numeroColores: número de colores de la nueva imagen
- pintor: booleano que se usa para saber si pintamos imagen al final o no.
- ro0: intensidad de ultrasonido en la fuente de origen
- eta: probabilidad de distorsión de mensaje a largas distancias
- imagen: ruta de la imagen a procesar por el algoritmo

```
super(ballena, self).__init__()


# Inicializar la poblacion de ballenas
self.__agents = np.random.uniform(lb, ub, (n, numeroColores, dimension))
self._points(self.__agents)

# Inicializamos el vector de las mejores posiciones de las ballenas
Pbest = self.__agents

# Vectores con el fitness actual, y el mejor fitness hallado.
fitActual = [function(x,numeroColores,imagen) for x in self.__agents]
fitMejor = fitActual   # Fitness de la iteracion actual
```

Ilustración 49 inicialización ballenas

En esta ilustración se muestra la inicialización del algoritmo de igual manera que el resto.

Se inicializan los valores del fitness actual y del mejor fitness.

```
for t in range(iteration):



    #print("Iteración ", t+1)
    new_agents = self.__agents
```

Ilustración 50 Inicio del bucle ballenas

Se inicia el bucle del algoritmo.

```
#Para cada particula ...
for i in range(n):
    #Buscamos una ballena con mejor fitness y que sea la mas cercana
    y = self.__better_and_nearest_whale(i, n, fitActual, fitMejor)
    # Si hemos encontrado otra ballena ...
    if y:
    # Movemos a la ballena i hacia otra mejor y cercana (depende de los dos parametros del algoritmo)
        new_agents[i] += np.dot(
            np.random.uniform(0, ro0 *
                np.exp(-eta * self.__whale_dist(i, y))),
            self.__agents[y] - self.__agents[i])
```

Ilustración 51 movimiento de ballenas

En esta ilustración se muestra como se mueven cada uno de los individuos. Para cada individuo se busca un individuo mejor que él mismo y que sea mas cercano.

Si se encuentra se mueve el individuo siguiendo esta ecuación:

$$x_i^{t+1} = x_i^t + \text{rand}\left(0, \rho_0 \cdot e^{-\eta \cdot d_{\mathbf{X},\mathbf{Y}}}\right) * \left(y_i^t - x_i^t\right)$$

```python
def __better_and_nearest_whale(self, u, n, fitActual, fitMejor):

    # Usado para comparar y quedarnos con la menor distancia
    temp = float("inf")

    v = None

    for i in range(n):
    #Para todas las ballenas ...

        if fitActual[i] < fitActual[u]:
        # Si el fitness de i es menor que el fitness de u ...
            # la distancia de i a u es
            dist_iu = self.__whale_dist(i, u)
            if dist_iu < temp:
                v = i
                temp = dist_iu
    return v #Devolvemos ballena
```

Ilustración 52 función buscar mejor y mas cercana ballena

Esta ilustración muestra como se busca un individuo mejor y más cercano.

Se recorren todos los individuos y para cada uno se compara el fitness del individuo desde el que se busca (u) y el fitness del individuo que toque en el bucle, si es menor se comprueba la distancia.

Se calcula la distancia entre individuos y si esta distancia es menor que la anterior que se encontró se almacena para comparar esta ultima con las siguientes iteraciones del bucle.

Al finalizar el bucle se devuelve el individuo con un mejor fitness que el del individuo que se evalúa y que sea más cercano.

```python
def __whale_dist(self, i, j):
    # Calculamos la distancia entre las ballenas, restando el vector ...
    return np.linalg.norm(self.__agents[i] - self.__agents[j])
```

Ilustración 53 calcular distancia entre individuos

En esta ilustración se muestra como se halla la distancia entre individuos usando la librería de numphy.

```
# Movemos Ballenas
self.__agents = new_agents
# Acotamos al espacio de solucion
self.__agents = np.clip(self.__agents, lb, ub)
self._points(self.__agents)


#Actualizamos su fitness actual
fitActual = [function(x,numeroColores,imagen) for x in self.__agents]

#Actualizamos la mejor solucion particular

# Para cada particula ...
for i in range(n):
    # Si el fitness de esta posicion es menor que el almacenado lo actualizamos
    #if(function(self.__agents[i],r) < function(Pbest[i],r)):
    if(fitActual[i] < fitMejor[i]):
        Pbest[i] = self.__agents[i]
        fitMejor[i] = fitActual[i]

#Actualizamos la mejor solucion global
Gbest = Pbest[np.array([function(x,numeroColores,imagen) for x in Pbest]).argmin()]

self.setMejorFitness(function(Gbest,numeroColores,imagen))
print(self.getMejorFitness(), end= ' ')
```

Ilustración 54 Fin bucle ballenas

Por último, se mueven los individuos, se acotan las posiciones al espacio de solución, se actualiza el fitness actual de cada uno, se actualiza la mejor solución de cada individuo comparando el fitness actual con el mejor fitness encontrado, si el actual es mejor que el mejor almacenado se actualiza la mejor posición particular y su mejor fitness.

Se actualiza la mejor solución global y se guarda el mejor fitness para imprimirlo.

En algunos algoritmos no solo se esta guardando la mejor posición global y particular, sino que también se guardan los valores de fitness, ya que a la hora de comparar todos y en caso de mejor guardarlos penaliza el rendimiento si se calculan siempre que se quiera acceder a ellos.

5.3 Funciones

En este apartado se documentan las funciones que se han utilizado en cada uno de los algoritmos.

5.3.1 pintaImagen

```python
def pintaImagen(cuantizada,nombreImagen,pintor,algoritmo,numeroColores):
        # Obtener la ruta del directorio donde se está ejecutando el script
        ruta_script = os.path.dirname(os.path.abspath(__file__))
        # Subir un nivel para obtener el directorio raiz
        ruta_prueba = os.path.dirname(ruta_script)
        #Voy a la carpeta images
        rutaImagen= os.path.join(ruta_prueba, 'images')

        # doy nombre a la imagen de salida en formato ALGORITMO_NUMCOLORES_IMAGEN
        nombreSalida = algoritmo + "_" + str(numeroColores) + "_" + os.path.basename(nombreImagen)
        #Contruyo la ruta hacia el directorio de destino
        rutaDestino = os.path.join(rutaImagen, nombreSalida)

         # Guarda la imagen cuantizada en un archivo
        resultado_guardado = cv2.imwrite(rutaDestino, cuantizada)

         # Verificar si la imagen se guardó correctamente
        if not resultado_guardado:
                print(f"Error: No se pudo guardar la imagen en '{nombreSalida}'")
                return

        # Lee la imagen original y la imagen cuantizada
        imagenori = cv2.imread(nombreImagen, cv2.IMREAD_COLOR)
        imagenresu = cv2.imread(rutaDestino, cv2.IMREAD_COLOR)

         # Si la imagen es un archivo JPEG, redimensionarla
        if nombreImagen.lower().endswith('.jpg') or nombreImagen.lower().endswith('.jpeg'):
                imagenori = redimensionar_imagen(imagenori, 800, 800)
                imagenresu = redimensionar_imagen(imagenresu, 800, 800)

        if(pintor):
                #Muestra la imagen original y la imagen cuantizada en ventanas separadas
                cv2.imshow('Imagen Original', imagenori)
                cv2.imshow('Nueva Imagen cuantizada', imagenresu)

                cv2.waitKey(0) #Esperamos a pulsar una tecla
                cv2.destroyAllWindows() #Cerramos
```

Ilustración 55 pintaImagen

Esta ultima ilustración muestra como el programa pinta las imágenes resultantes del algoritmo.

Primero se obtiene la ruta del directorio actual y se sube un nivel para estar en el directorio raíz, esto es debido a que este archivo esta en un nivel diferente al de la carpeta que contiene las imágenes.

Luego se accede a la carpeta de imágenes.

Se forma el nombre a la imagen resultante que se va a almacenar en formato ALGORITMO_NUMCOLORES_IMAGEN.

Se guarda la imagen cuantizada, se hace una comprobación para en caso de que no se haya podido guardar de error y no intente leer la imagen.

Después se leen las dos imágenes, la original y la cuantizada.

La comprobación de si es un jpg la redimensiona es porque al hacer las pruebas y pasarle una imagen personal hecha con un teléfono, el programa me pintaba las imágenes en un formato muy grande, se salía de la pantalla.

Si se el parámetro de pintar la imagen se ha pasado a true pinta las dos imágenes, después el programa espera a que se pulse cualquier tecla, al pulsarla se destruyen las ventanas y termina.

Este método se llama al final de cada algoritmo.

5.3.2 preparaImagen

```python
def preparaImagen(nombreImagen):
	# Leemos la imagen
	img=cv2.imread(nombreImagen, cv2.IMREAD_COLOR)

	# Si la imagen es un archivo JPEG, redimensionarla
	if nombreImagen.lower().endswith('.jpg') or nombreImagen.lower().endswith('.jpeg'):
		img = redimensionar_imagen(img, 800, 800)

	# Redimensiona la imagen a una matriz 2D de pixeles (cada fila es un pixel, cada columna es una de las componentes RGB)
	# Modificamos img para ir de punto en punto
	z= img.reshape((-1,3))
	z = np.float32(z)

	return z,img
```

Ilustración 56 preparaImagen

Esta ilustración muestra el método que se llama para preparar las imágenes para operar con ellas, se encarga de leer la imagen, redimensionarla si es jpg y redimensionar la imagen a una matriz 2D de pixeles y luego se devuelve la matriz junto con la imagen.

5.3.3 generaCuantizada

```python
def generaCuantizada(x,tam_paleta,nombreImagen):
        #Preparamos imagen
        z,img=preparaImagen(nombreImagen)

        # Elimina los pixeles duplicados para evitar problemas con KMeans
        z_unique = np.unique(z, axis=0)

        # Ajusta el número de clusters al número de pixeles únicos si es menor que tam_paleta
        tam_paleta = min(tam_paleta, len(z_unique))

        # Aplica KMeans a los pixeles únicos para encontrar los clusters (colores representativos)
        k_means = KMeans(n_clusters=tam_paleta,init=x, n_init=1,max_iter=1,algorithm="lloyd").fit(z_unique)

        # Guardamos el valor de los centroides y de las etiquetas de cada pixel
        labels = k_means.predict(z)

        # Obtiene los centroides de los clusters (los colores representativos)
        paleta = k_means.cluster_centers_
        # con np.uint8 los convertimos a enteros de 8 bits
        paleta = np.uint8(paleta)

        # Reemplaza cada pixel en la imagen original por el color del cluster al que pertenece
        img_cuantizada = paleta[labels.flatten()]
        img_cuantizada2 = img_cuantizada.reshape((img.shape))

        return img_cuantizada2
```

Ilustración 57 generaCuantizada

Esta ilustración muestra el método encargado de generar una imagen cuantizada.

Primero llama a preparaImagen para preparar la imagen para su uso.

Se eliminan los pixeles duplicados, esa línea es debida a que a raíz de actualizar la librería sklearn se experimentaron errores que se corrigieron con esto.

Se ajustan el número de clústeres al número de pixeles únicos si es menor que tam_paleta que es el tamaño de la paleta que se pide o el número de colores.

Se aplica el Kmeans de la librería sklearn a los pixeles únicos para encontrar los clústeres.

Se guardan los valores de las etiquetas de cada pixel y se obtienen los centroides para luego reemplazar cada pixel en la imagen original por el color del clúster al que pertenece.

Por ultimo se devuelve la imagen cuantizada.

5.3.4 getMse

```
def getMse(x,tam_paleta,nombreImagen):
        z,img =preparaImagen(nombreImagen)
        img_cuantizada2 = generaCuantizada(x,tam_paleta,nombreImagen)

        # Aplanar img_cuantizada2 para que coincida con la forma de z
        img_cuantizada2_flat = img_cuantizada2.reshape((-1, 3))

        # Calcula el error cuadrático medio entre la imagen original y la imagen cuantizada
        return mean_squared_error(z, img_cuantizada2_flat)
```

Ilustración 58 función MSE

En esta ultima ilustración se muestra como se calcula el mse, se prepara la imagen, se genera la cuantizada y se llama a mean_mean_squared_error de la librería sklearn para calcular el error cuadrático medio

5.3.5 getMae

```python
def getMae(x, tam_paleta,nombreImagen):
    # Prepara la imagen, devolviendo tanto la imagen aplanada (z) como la original (img)
    z, img = preparaImagen(nombreImagen)

    # Genera la imagen cuantizada
    img_cuantizada2 = generaCuantizada(x, tam_paleta,nombreImagen)

    # Aplanar img_cuantizada2 para que coincida con la forma de z
    img_cuantizada2_flat = img_cuantizada2.reshape((-1, 3))

    # Calcular el MAE entre la imagen original (aplanada) y la imagen cuantizada (aplanada)
    mae = np.mean(np.abs(z - img_cuantizada2_flat))

    return mae
```

Ilustración 59 getMae

En esta ilustración se muestra como se calcula el mae, se prepara la imagen, se genera la imagen cuantizada, se redimensiona y se calcula el mae gracias a numphy. Devuelve el error absoluto medio.

5.3.6 getSsim

```python
def getSsim(x, tam_paleta,nombreImagen):
    # Prepara la imagen, devolviendo tanto la imagen aplanada (z) como la original (img)
    z, img = preparaImagen(nombreImagen)

    # Genera la imagen cuantizada
    img_cuantizada2 = generaCuantizada(x, tam_paleta,nombreImagen)

    # Redimensiona la imagen cuantizada a una matriz 2D de píxeles
    img_cuantizada2_plana = img_cuantizada2.reshape((-1, 3))
    img_cuantizada2_plana = np.float32(img_cuantizada2_plana)

    # Calcular el SSIM entre la imagen original y la imagen cuantizada
    ssim_index = ssim(z, img_cuantizada2_plana,multichannel=True,channel_axis=-1,data_range=img.max() - img.min())

    # Utiliza 1 - SSIM como fitness, donde valores más bajos son mejores
    return 1 - ssim_index
```

Ilustración 60 getSsim

En esta ilustración se muestra como se calcula el ssim, igual que en las anteriores se prepara la imagen y se genera la imagen cuantizada, luego se redimensiona la imagen a una matriz de dos dimensiones para luego calcular el ssim gracias a la librería skimage

Como esta librería devuelve el índice de similitud en el que 0 no se parecen y 1 si se parecen, invierto el valor al restar 1 menos lo que devuelve ssim para que se ajuste a los algoritmos.

5.3.7 getMssim

```python
def getMsSsim(x, tam_paleta,nombreImagen):

    # Prepara la imagen, devolviendo tanto la imagen aplanada (z) como la original (img)
    z, img = preparaImagen(nombreImagen)

    # Genera la imagen cuantizada usando la paleta de colores (x)
    img_cuantizada2 = generaCuantizada(x, tam_paleta,nombreImagen)

    # Redimensiona la imagen cuantizada a una matriz 2D de píxeles
    img_cuantizada2_flat = img_cuantizada2.reshape((-1, 3))
    img_cuantizada2_flat = np.float32(img_cuantizada2_flat)

    # Ajuste del tamaño de la ventana, se asegura que el tamaño de la ventana win_size sea menor o igual al tamño de la imagen más pequeña
    win_size = min(img.shape[0], img.shape[1], 7)

    # Calcular el MS-SSIM entre la imagen original y la imagen cuantizada
    ms_ssim_index = ssim(z, img_cuantizada2_flat, multichannel=True, gaussian_weights=True, sigma=1.5, use_sample_covariance=False, win_size=win_size, channel_axis=-1, data_range=img.max() - img.


    # Utilizar 1 - MS-SSIM como fitness, donde valores más bajos son mejores
    fitness = 1 - ms_ssim_index

    return fitness
```

Ilustración 61 getMssim

Esta función calcula el mssim de la misma manera que el ssim, la diferencia son los parámetros que se mandan a ssim lo que en vez de devolver el índice de similitud estructural (ssim) devuelve el índice de similitud multi escalar (mssim).

Como en la función anterior se invierte el índice para ajustarse a los algoritmos.

[1] «Los modos de color de la imagen digital». [En línea]. Disponible en: https://www.fotonostra.com/fotografia/modoscolor.htm

[2] «Espacio de color - Wikipedia, la enciclopedia libre». [En línea]. Disponible en: https://es.wikipedia.org/wiki/Espacio_de_color

[3] «Inteligencia de enjambre e inteligencia artificial - Fundación Aquae».[En línea]. Disponible en: https://www.fundacionaquae.org/la-inteligencia-enjambre-y-la-inteligencia-artificial/

[4] «PSO: Optimización por enjambres de partículas [En línea]. Disponible en: https://www.cs.us.es/~fsancho/Blog/posts/PSO.md

[5] «James Kennedy (social psychologist) - Wikipedia». [En línea]. Disponible en: https://en.wikipedia.org/wiki/James_Kennedy_(social_psychologist)

[6] «Russell C. Eberhart - Wikipedia [En línea]. Disponible en: https://en.wikipedia.org/wiki/Russell_C._Eberhart

[7] V. Álvarez-Garduño, N. Guadiana-Ramírez, y Á. Anzueto-Ríos, «Análisis comparativo de la modificación del parámetro de inercia para la mejora en el desempeño del algoritmo PSO», *Científica*, vol. 25, n.o 1, pp. 104-114, 2021, doi: 10.46842/IPN.CIEN.V25N1A09.

[8] «Algoritmo firefly - Wikipedia, la enciclopedia libre». [En línea]. Disponible en: https://es.wikipedia.org/wiki/Algoritmo_firefly

[9] «Dr Xin-She Yang | Middlesex University». [En línea]. Disponible en: https://www.mdx.ac.uk/about-us/our-people/staff-directory/dr-xin-she-yang/

[10] «Xin-She Yang - Wikipedia, la enciclopedia libre [En línea]. Disponible en: https://es.wikipedia.org/wiki/Xin-She_Yang

[11] S. Mirjalili, S. M. Mirjalili, y A. Lewis, «Grey Wolf Optimizer», *Advances in Engineering Software*, vol. 69, pp. 46-61, mar. 2014, doi: 10.1016/J.ADVENGSOFT.2013.12.007.

[12] «GWO». [En línea]. Disponible en: https://seyedalimirjalili.com/gwo

[13] H. Faris, I. Aljarah, M. A. Al-Betar, y S. Mirjalili, «Grey wolf optimizer: a review of recent variants and applications», *Neural Comput Appl*, vol. 30, n.o 2, pp. 413-435, jul. 2018, doi: 10.1007/S00521-017-3272-5/TABLES/4.

[14] «Seyedali Mirjalili».[En línea]. Disponible en: https://seyedalimirjalili.com/

[15] «Seyedali Mirjalili - Google Scholar». [En línea]. Disponible en: https://scholar.google.com/citations?user=TJHmrREAAAAJ&hl=en

[16] J. Nasiri y F. M. Khiyabani, «A whale optimization algorithm (WOA) approach for clustering», *Cogent Math Stat*, vol. 5, n.o 1, p. 1483565, ene. 2018, doi: 10.1080/25742558.2018.1483565.

[17] B. Zeng, L. Gao, y X. Li, «Whale swarm algorithm for function optimization», *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10361 LNCS, pp. 624-639, 2017, doi: 10.1007/978-3-319-63309-1_55/FIGURES/5.

[18] «AN IDEA BASED ON HONEY BEE SWARM FOR NUMERICAL OPTIMIZATION».

[19] «Algoritmo colonia de abejas artificiales - Wikipedia, la enciclopedia libre». [En línea]. Disponible en: https://es.wikipedia.org/wiki/Algoritmo_colonia_de_abejas_artificiales

[20] «(10) Derviş Karaboğa | LinkedIn». [En línea]. Disponible en: https://www.linkedin.com/in/dervi%C5%9F-karabo%C4%9Fa-886ba23/

[21] «Dervis Karaboga - Google Académico». [En línea]. Disponible en: https://scholar.google.es/citations?user=aC77_cUAAAAJ&hl=es&oi=aohttps://scholar.google.es/citations?user=aC77_cUAAAAJ&hl=es&oi=ao

[22] *Why was Python created in the first place?* General Python FAQ. [En línea]. Disponible en: http://docs.python.org/faq/general#why-was-python-created-in-the-first-place

[23] «artima - The Making of Python», *www.artima.com*, [En línea]. Disponible en: https://www.artima.com/articles/the-making-of-python

[24] «La librería Numpy | Aprende con Alf». [En línea]. Disponible en: https://aprendeconalf.es/docencia/python/manual/numpy/

[25] «Procesamiento de imágenes con OpenCV en Python [En línea]. Disponible en: https://imaginaformacion.com/tutoriales/opencv-en-python

[26] «Scikit-Learn, herramienta básica para el Data Science en Python». [En línea]. Disponible en: https://www.master-data-scientist.com/scikit-learn-data-science/

[27] S. Van Der Walt *et al.*, «scikit-image: image processing in Python», *PeerJ*, vol. 2:e453, n.o 1, p. e453, jul. 2014, doi: 10.7717/peerj.453.