UNIVERSIDAD AUTÓNOMA METROPOLITANA
División de Ciencias Naturales e Ingeniería
Departamento de Matemáticas Aplicadas y Sistemas
$ Licenciatura en Ingeniería en Computación

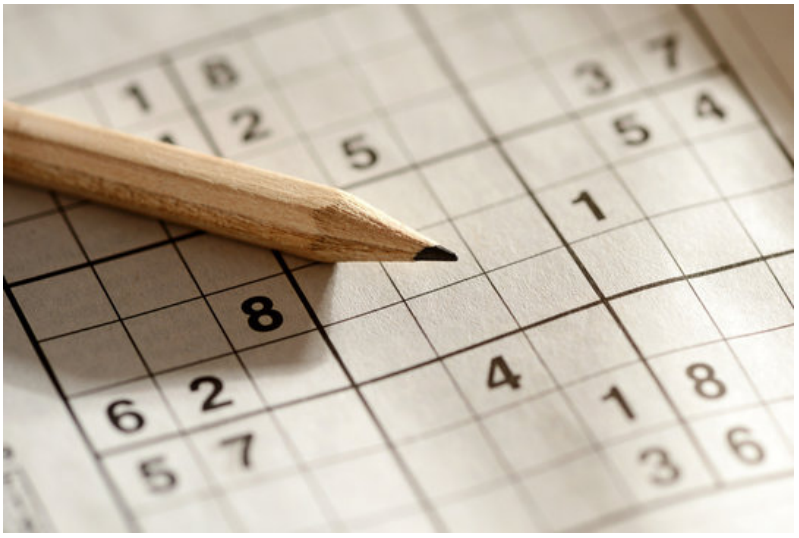
Sistemas Inteligentes / Inteligencia de Enjambre.

PROYECTO 4: Problema Sudoku

Docente: Alejandro Lara Caballero

Autores:

Cortes Lopez Alan Yair
Garcia Nuñez Rodrigo
Chavez Flores Alejandro
Hernandez Flores Aldo Enrique



Índice

Introducción

El Sudoku, un popular juego de lógica, que ha sido objeto de estudio y aplicación en diversas áreas de la tecnología y la inteligencia artificial. Este juego consiste en rellenar una cuadrícula de 9x9 celdas dividida en subcuadrículas de 3x3, de tal manera que cada fila, columna y subcuadrícula contenga todos los dígitos del 1 al 9 sin repetirse. Para resolver un Sudoku plantea un desafío computacional interesante debido a la combinación de restricciones y la búsqueda de soluciones válidas.

Los algoritmos bioinspirados han ganado popularidad como enfoques eficaces para abordar problemas de optimización combinatoria. En este proyecto utilizaremos uno de estos algoritmos que es la Optimización por Enjambre de Partículas (PSO), inspirado en el comportamiento social de los organismos, como los enjambres de pájaros o parvadas, donde cada "partícula" del enjambre ajusta su posición en función de su propia experiencia y de la información compartida con sus vecinos.

Nos enfocaremos en implementar el algoritmo PSO para resolver el problema del Sudoku. Utilizaremos la representación del tablero como una matriz de números enteros, donde cada celda corresponde a un espacio en blanco en el Sudoku que debe ser rellenado con un número válido. La función de evaluación será diseñada para penalizar las soluciones que no cumplan con las reglas del Sudoku.

Además, compararemos el desempeño del algoritmo PSO con otra metaheurística, la estrategia greedy que se caracteriza por tomar decisiones locales óptimas en cada paso sin considerar el impacto a largo plazo. Esta comparación nos permitirá evaluar la eficacia y eficiencia relativa de ambos enfoques en la resolución del Sudoku. Por último mostraremos los análisis de resultados de cada una de las técnicas que utilizamos que es el algoritmo PSO y greedy.

Descripción del problema.

En la actualidad, el sudoku es uno de los pasatiempos más famosos del mundo. Gran parte de las revistas y periódicos de prácticamente todo el mundo tienen un hueco reservado en sus páginas de entretenimiento para este juego. La fácil comprensión de las reglas hace a este pasatiempo atractivo para que personas de todas las edades se atrevan a intentar resolverlo, aunque esto no implica que sea simple de solucionar.

El juego consiste en rellenar una tabla de 9 × 9, 81 casillas o celdas en total, dividida en regiones de 3 × 3, con las cifras del 1 al 9. Inicialmente, algunos números vienen fijados en algunas de las casillas, denominados pistas. La condición esencial del juego se basa en escribir una sola vez cada cifra en cada grupo (columna, fila y región). Desde el punto de vista de la optimización, el sudoku, presenta el fenómeno de explosión combinatoria, puesto que, las dimensiones y complejidad matemática que resulta resolver el problema es grande. Por ello, está clasificado como un problema NP-completo, es decir, no existe un algoritmo que lo resuelva en un tiempo de cómputo polinomial en el tamaño N del problema.

La cuadrícula de 9 × 9 con regiones de 3 × 3 es la más común. También existen otros tamaños, generalmente, de dimensión N × N con regiones de $\sqrt{N} \times \sqrt{N}$, en este caso, se utilizan N números o caracteres para llenar la cuadrícula, como 4×4, 16×16, 25×25. También podemos encontrar cuadrículas con regiones que no son cuadradas.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Imagen 1. Ejemplo de datos de entrada que se dan al inicio del juego.

Generalmente, se dice que, un sudoku está bien planteado cuando solo tiene una única solución, para ello, demostrado por McGuire, Tugemann y Civario, el número mínimo de pistas necesario es 17. Esto no implica que todo sudoku con 17 datos iniciales, o más, esté bien planteado.

Cuando nos disponemos a enfrentarnos a un sudoku, podemos encontrar varios niveles de dificultad. Esta dificultad depende de la relevancia y la posición de los números dados, no de la cantidad de pistas iniciales, puesto que, un jugador ante una cuadrícula con un número de pistas bajo, lo que el vería como una cuadrícula casi vacía, podría resultar más fácil de resolver que uno con un número de pistas iniciales mayor. La dificultad reside en los métodos de resolución que tenga que utilizar el jugador para rellenar la cuadrícula, ante métodos de mayor elaboración se considera un nivel de dificultad más alto.

Si empezamos con la cuadrícula en blanco, es decir, ninguna pista inicial, podemos observar que para un sudoku de orden 1 la cantidad de soluciones es 1, mientras que cuando el orden es 4 hay 288 soluciones. Usando la combinatoria, vemos que, tenemos 4! formas de llenar la primera fila. En cuanto a la segunda fila, una vez fijada la primera fila, hay 2 opciones para las casillas que pertenecen a la primera región y otras 2 para las casillas de la segunda región, en total 4 formas de rellenarla. En la tercera fila, una vez fijadas las dos anteriores, observamos que, dependiendo de la fila dos, puede haber 4 o 2 formas de rellenarla, por tanto, la media de opciones para la fila 3 es 3. La cuarta fila, una vez fijadas las otras tres, solo tiene una opción, con lo que concluimos que 4! ·4·3 = 288.

En el caso de un sudoku de orden 9, se ha estimado que el número de soluciones es 6.670.903.752.021.072.936.960. Su descomposición en factores primos es 220 · 3 8 · 5 · 7 · 27.704.267.971. Un número primo tan grande como el 27.704.267.971 implica que no podemos contar el número de soluciones como un simple problema de combinatoria, para conseguirlo ha hecho falta la computación.

Representación Matemática del Problema

La complejidad de este problema se determina por medio de la cantidad de espacios en blanco que tenga nuestra primera instancia, ya que entre menos espacios en blanco haya o en otras palabras entre más pistas existan (con pistas nos referimos a la cantidad de datos iniciales que nos dan como entrada en la Imagen 1. se puede visualizar este tipo de datos) menor es el espacio de soluciones, en otras palabras, será más sencillo encontrar una respuesta del problema planteado.

Pistas	Cuadros en blanco	Espacio de soluciones
75	6	5,3144E+05
70	11	3,1300E+10
65	16	1,8530E+15
60	21	1,0942E+20
55	26	6,4611E+39
50	31	3,8152E+44
45	36	2,2528E+49
40	41	1,3303E+54
35	46	7,8552E+58
30	51	4,6384E+63
25	56	2,7389E+68
20	61	1,6173E+73
15	66	9,5500E+77
10	71	5,6392E+82
5	76	3,3299E+87
0	81	1,9663E+92

Imagen. Espacio de soluciones según la cantidad de pistas para el sudoku de orden.

Con base a la imagen. Podemos observar que mientras más espacios en blanco existan en el sudoku más grande será nuestro número de soluciones y por ende más complejo será llegar a una solución de nuestro problema.

Técnica de optimización bioinspirada.

La técnica bioinspirada que elegimos fue el algoritmo de optimización por enjambre de partículas (PSO), originalmente desarrollado por James Kennedy y Russell Eberhart en 1995, es un algoritmo del área de la inteligencia artificial de la rama de inteligencia de enjambres. Está inspirada en el comportamiento social de los seres vivos. El algoritmo de optimización por enjambre de partículas se inspira en la evolución en el comportamiento colectivo, principalmente trata de imitar el comportamiento social de varios grupos de animales como lo son los peces, parvadas, manadas etc,. El algoritmo se basa en población de individuos, solo que con diferentes enfoques y han demostrado ser eficientes para la solución de problemas complejos.

El algoritmo es de optimización metaheurísticos, ya que utiliza analogías con otros procesos para resolver el problema, los métodos metaheurísticos no se especializan en resolver un problema en particular, por lo que puede usarse para cualquier problema. Este método no garantiza dar el mejor resultado, pero sí un resultado aceptable. Otra característica de estos algoritmos, es que son no deterministas (estocásticos), esto quiere decir que los resultados obtenidos no siempre serán los mismos aunque se trate de una misma función. En muchas de sus aplicaciones, los algoritmos de enjambre de partículas, ofrecen resultados de calidad del 99%, sin embargo, se ha demostrado que los algoritmos de enjambre de partículas son superiores en cuanto a eficiencia, debido a su bajo costo computacional. El algoritmo de optimización de enjambre de partículas original ha tenido varios cambios y han surgido variaciones del mismo según el problema que se quiera resolver. Sin embargo, en 2006, se trató de establecer un estándar (SPSO), y que posteriormente se le ha contribuido con algunas sugerencias y cambios en el 2007 y 2011. Comparado con el algoritmo clásico, el estándar, agregó el factor social, cognitivo, de inercia y construcción. También se cambió el modo en que las partículas se comunicaban por medio de topologías definidas.

PSO Tradicional.

Esta es la versión más popular del algoritmo PSO y es la que se viene exponiendo a lo largo de esta memoria.

Algoritmo 3 Pseudocódigo del algoritmo PSO básico

```
S ← InicializarCumulo()
while no se alcance la condición de parada do
  for i = 1 to size(S) do
    evaluar cada partícula x_i del cúmulo S
    if fitness(x_i) es mejor que fitness(pBest_i) then
      pBest_i ← x_i; fitness(pBest_i) ← fitness(x_i)
    end if
    if fitness(pBest_i) es mejor que fitness(g_i) then
      g_i ← pBest_i; fitness(g_i) ← fitness(pBest_i)
    end if
  end for
  for i = 1 to size(S) do
    elegir p_j, la partícula con mejor fitness del vecindario p_i
    actualizar la velocidad de la partícula p_i, de acuerdo con los valores de p_i y p_j
    mover la partícula p_i de acuerdo con su nueva velocidad
  end for
end while
```

En esta sección, se añaden algunos comentarios más específicos de la codificación continua, sobre todo en lo que respecta a la representación de las partículas y operadores. En Algoritmo se muestra una versión más específica del pseudocódigo del PSO para codificación continua.

Algoritmo 4 Pseudocódigo del algoritmo PSO para codificación continua

$S \leftarrow InicializarCumulo()$
while no se alcance la condición de parada **do**
  for $i = 1$ to $size(S)$ **do**
    evaluar cada partícula x_i del cúmulo S
    if $fitness(x_i)$ es mejor que $fitness(pBest_i)$ **then**
      $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$
    end if
    if $fitness(pBest_i)$ es mejor que $fitness(g_i)$ **then**
      $g_i \leftarrow pBest_i$; $fitness(g_i) \leftarrow fitness(pBest_i)$
    end if
  end for
  for $i = 1$ to $size(S)$ **do**
    $v_i \leftarrow \omega \cdot v_i + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i) + \varphi_2 \cdot rand_2 \cdot (g_i - x_i)$
    $x_i \leftarrow x_i + v_i$
  end for
end while
Salida: la mejor solución encontrada

Para no confundir el ámbito de actuación de los operadores, es decir, PSO Global o PSO Local, representamos la mejor partícula del vecindario como gi , pudiendo ser ésta bien gBesti o bien lBesti dependiendo de la versión elegida.

El PSO se describe generalmente como una población de vectores (llamadas partículas) cuyas trayectorias oscilan en torno a una región. En el caso más general, teniendo en cuenta la configuración del enjambre (población), hay dos versiones del PSO: globales y locales. En la versión global (gbest) la trayectoria de cada partícula está influenciada por el mejor punto encontrado por cualquier miembro de todo el enjambre. En la versión local (lbest) la trayectoria de cada partícula puede ser influenciada solo por el mejor miembro en el vecindario (el mismo enjambre). Además, cada partícula también se ve influida por su mejor éxito previo.

PSO Para Codificación Binaria.

En la actualidad, se están estudiando una gran cantidad de problemas de optimización que requieren de una representación binaria de sus soluciones. Por este motivo, es necesario proveer una versión del algoritmo PSO que trabaje con este tipo de codificación. En este caso, las posiciones en el espacio de búsqueda se representan mediante cadenas de bits, por lo que el algoritmo y los operadores pueden variar sustancialmente respecto a la versión continua. Kennedy y Eberhart propusieron una adaptación inicial del PSO binario. En esta versión, la posición de la partícula es un vector cadena de bits mientras que la velocidad se representa mediante un vector real.

En su forma más general el PSO binario se describe como: Se denota a Np como el número de partículas en la población. Así que Y t i Y_i^t = (y_{i1}^t,y_{i2}^t,...,y_{iD}^t), y y_i^t ∈ 0, 1, es una partícula i con D bits en la iteración t, donde Y_i, es una solución potencial que tiene una tasa de cambio llamada velocidad.

La velocidad es V_i^t = (v_{i1}^t,v_{i2}^t,...,v_{iD}^t), v t id ∈ R. Así que P_i^t = (p_{i1}^t,p_{i2}^t,...,p_{iD}^t)representa la mejor solución que la partícula i ha obtenido hasta la iteración t, y P_i^t = (p_{i1}^t,p_{i2}^t,...,p_{iD}^t representa la mejor solución obtenida a partir de P_i^t en la población (gbest) o del vecindario local (lbest), en la iteración t.

Cada partícula ajusta su velocidad de acuerdo a la siguiente ecuación: $v_i^t = v_{id}^{t-1} + co_1 r_1 (p_{id}^t - y_{id}^t) + co_2 r_2 (p_{id}^t - y_{id}^t)$ donde co_1 es el factor de aprendizaje cognitivo, co_2 es el factor de aprendizaje social, y r_1 y r_2 Son números generados en forma aleatoria con distribución uniforme en [0, 1]. Los valores co y r determinan los pesos de dos partes donde la suma de ellos es limitada normalmente a 4. La velocidad representa la probabilidad de que el valor actual tome el valor de uno. Para mantener el valor en el intervalo [0, 1] se emplea la función sigmoidal:

$$sig(v_{id}^t) = \frac{1}{1+exp\,(-v_{id}^t)}$$

Algoritmo de PSO

Algoritmo 1 Red de ordenamiento para $n = 4$

Input: x_0, x_1, x_2, x_3

```
1    begin
2        FOR i = 0 hasta Np
3            STATE Calcular la aptitud por cada p_i
4            IF Y_i^t > P_i^t
5                FOR d = 1 hasta D bits
6                    STATE P_id^t = Y_i^t
7                END FOR
8            END IF
9            g = i
10           FOR j=índices de los vecinos
11               IF P_j^t > P_g^t
12                   STATE g = j
13               END IF
14           END FOR
15           FOR d = 1 hasta D
16               STATE v_id^t = v_id^(t-1) + co1 r1 (P_id^t - Y_id^t) + co2 r2 (P_gd^t - Y_id^t)
17               STATE v_id^t ∈ [-V_max, +V_max]
18               IF Número aleatorio [0, 1] < sig(v_id^t)
19               STATE Y_i^(t+1) = 1
20           ELSE
21               STATE Y_i^t = 0
22           END IF
23           END FOR
24       END FOR
```

Output: y_1, y_2, y_3, y_4

Descripción de pseudocódigo de PSO

El algoritmo comienza en el paso 1, donde se define el tamaño del enjambre de partículas (Np) y se calcula la aptitud para cada partícula (pi). En el paso 2, se compara la aptitud total (Yt) con la aptitud de cada partícula (Pit), y si Yt es mayor que Pit, se copia el valor de la partícula (Pid) en el vector de velocidad (vid) de la partícula actual. En el paso 3, se selecciona el índice de la mejor partícula global (g) hasta el momento.

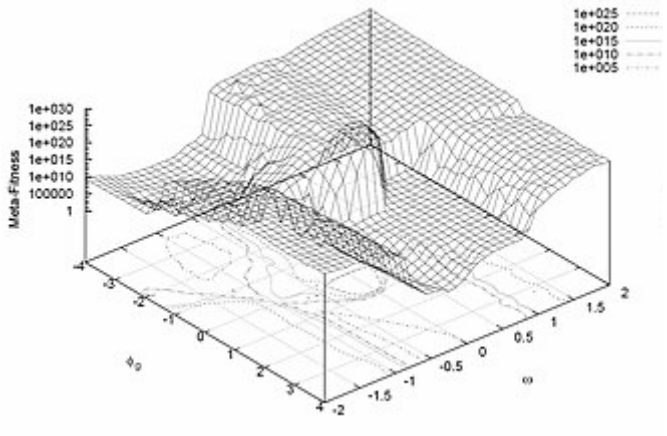
En los pasos 4 y 5, se recorren los índices de los vecinos de cada partícula y se comparan sus aptitudes (Pt) con la del vecino actual. Si la aptitud del vecino es mayor, se actualiza el vector de velocidad (vid) de la partícula actual.

En el paso 6, se recorren los bits del vector de velocidad (vid) de cada partícula y se actualiza el vector de posición (Y) de la partícula actual. La actualización se realiza mediante la fórmula: vid = v^(t-1) + cri * (Pid - Yid) + c2r2 * (Pgd - Yid), donde cri y c2r2 son constantes de aceleración y r1 y r2 son números aleatorios entre 0 y 1. El nuevo valor de Y se limita entre -Vmax y +Vmax.

En el paso 7, se verifica si el número aleatorio entre 0 y 1 es menor que la función de probabilidad de la nueva partícula (sig(vid)). Si se cumple la condición, se selecciona la nueva partícula como la mejor partícula global (g), y se actualiza el vector de posición (Y).

Finalmente, en el paso 8, se devuelve el vector de posición (Y) como la solución al problema de optimización.

En la imagen se muestra también el ejemplo de entrada del algoritmo, con un tamaño de enjambre de 4 partículas, los valores de entrada To, 1, 2, 3 y el vector de aptitudes Yt = [92, 93, 94].

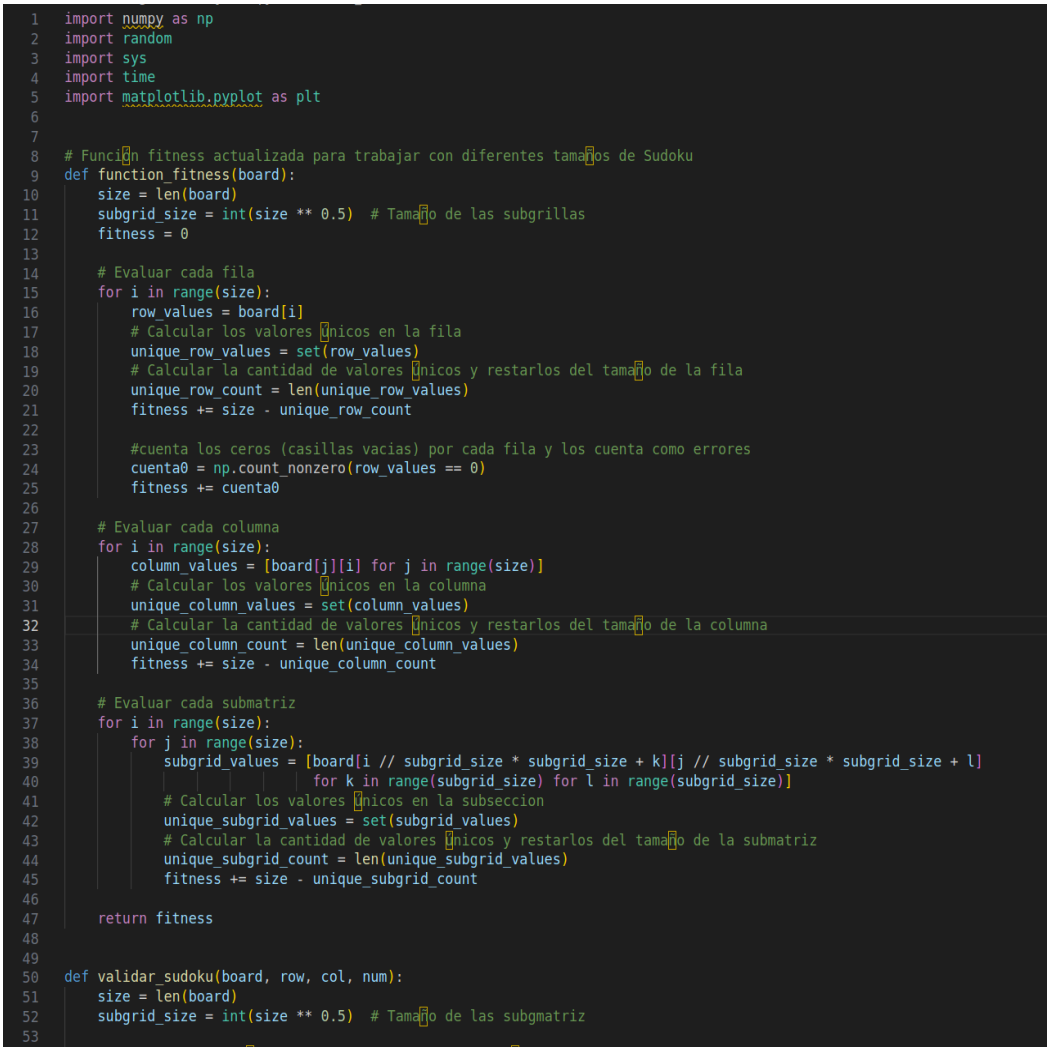


Gráfica bidimensional que muestra el rendimiento de una variante PSO ante un problema en función de dos parámetros.

Implementación del algoritmo.

Para este proyecto, se optó por utilizar el lenguaje de programación Python debido a su versatilidad y la amplia gama de bibliotecas disponibles que facilitan la implementación de algoritmos complejos como el PSO (Optimización por Enjambre de Partículas). El algoritmo PSO se seleccionó por su capacidad para encontrar soluciones óptimas o cercanas a óptimas en problemas de optimización complejos como el Sudoku. La implementación del algoritmo PSO se realizó en Python, aprovechando las características del lenguaje que permiten una sintaxis clara y legible, lo que facilita la comprensión y el mantenimiento del código.

Para desarrollar el algoritmo PSO, se utilizaron las bibliotecas estándar de Python, lo que garantiza la portabilidad del código y su capacidad para ejecutarse en diferentes plataformas sin necesidad de instalar bibliotecas externas. Se implementaron las funciones necesarias para la inicialización de las partículas, la actualización de sus posiciones y velocidades, y la evaluación de la función de aptitud. Además, se incluyeron los parámetros del algoritmo, como el número de partículas, el número de iteraciones y los coeficientes de inercia, que pueden ajustarse según las necesidades específicas del problema a resolver.



```python
import numpy as np
import random
import sys
import time
import matplotlib.pyplot as plt


# Función fitness actualizada para trabajar con diferentes tamaños de Sudoku
def function_fitness(board):
    size = len(board)
    subgrid_size = int(size ** 0.5)  # Tamaño de las subgrillas
    fitness = 0

    # Evaluar cada fila
    for i in range(size):
        row_values = board[i]
        # Calcular los valores únicos en la fila
        unique_row_values = set(row_values)
        # Calcular la cantidad de valores únicos y restarlos del tamaño de la fila
        unique_row_count = len(unique_row_values)
        fitness += size - unique_row_count

        #cuenta los ceros (casillas vacias) por cada fila y los cuenta como errores
        cuenta0 = np.count_nonzero(row_values == 0)
        fitness += cuenta0

    # Evaluar cada columna
    for i in range(size):
        column_values = [board[j][i] for j in range(size)]
        # Calcular los valores únicos en la columna
        unique_column_values = set(column_values)
        # Calcular la cantidad de valores únicos y restarlos del tamaño de la columna
        unique_column_count = len(unique_column_values)
        fitness += size - unique_column_count

    # Evaluar cada submatriz
    for i in range(size):
        for j in range(size):
            subgrid_values = [board[i // subgrid_size * subgrid_size + k][j // subgrid_size * subgrid_size + l]
                              for k in range(subgrid_size) for l in range(subgrid_size)]
            # Calcular los valores únicos en la subseccion
            unique_subgrid_values = set(subgrid_values)
            # Calcular la cantidad de valores únicos y restarlos del tamaño de la submatriz
            unique_subgrid_count = len(unique_subgrid_values)
            fitness += size - unique_subgrid_count

    return fitness


def validar_sudoku(board, row, col, num):
    size = len(board)
    subgrid_size = int(size ** 0.5)  # Tamaño de las subgmatriz

```

Evaluación del algoritmo.

La evaluación del algoritmo PSO se llevó a cabo comparándolo con otros dos enfoques comunes para resolver el Sudoku: un algoritmo genético y una estrategia greedy. Se realizaron experimentos computacionales utilizando diferentes instancias de problemas de Sudoku, lo que permitió comparar las soluciones obtenidas por cada algoritmo en términos de calidad, tiempo de ejecución y robustez.

En términos de calidad de la solución, se analizó la precisión con la que cada algoritmo pudo resolver el Sudoku, evaluando si las soluciones encontradas eran óptimas o cercanas a óptimas. Se calcularon métricas como el fitness de la solución y la cantidad de errores en el tablero resuelto para determinar la calidad de las soluciones generadas por cada algoritmo.

Además, se evaluó el tiempo de ejecución requerido por cada algoritmo para resolver el Sudoku, lo que permitió comparar su eficiencia en términos de velocidad de convergencia. Se registraron los tiempos de ejecución de cada algoritmo y se analizaron los factores que podrían influir en su rendimiento, como el tamaño del problema y los parámetros del algoritmo.

Se analizó su capacidad para encontrar soluciones óptimas o cercanas a óptimas en una variedad de instancias de problemas de Sudoku, incluyendo casos difíciles con múltiples soluciones posibles. Se evaluaron las estrategias de exploración y explotación del espacio de búsqueda de cada algoritmo y se compararon sus capacidades para encontrar soluciones de alta calidad en diferentes escenarios.

Comparación con otros algoritmos.

Para comparar este algoritmo, se hizo uso de otros dos algoritmos(greedy y metaheurística) el algoritmo Greedy toma decisiones inmediatas basadas en la información disponible en cada paso. Es un enfoque simple que elige la mejor opción local en cada etapa sin considerar el impacto a largo plazo. Puede encontrar soluciones rápidas pero no garantiza la optimización global.

PSO está inspirado en el comportamiento social de los animales, donde las partículas (soluciones candidatas) cooperan para encontrar la mejor solución. Utiliza la exploración (movimiento aleatorio) y la explotación (seguir la mejor solución encontrada) para buscar soluciones. Puede encontrar soluciones de alta calidad, pero puede ser computacionalmente costoso debido a su necesidad de iteraciones y ajuste de parámetros.

Los AG están basados en la evolución biológica y operan con una población de soluciones candidatas. Utilizan operadores de selección, cruza y mutación para generar nuevas soluciones y mejorar gradualmente la población. Pueden ser efectivos para encontrar soluciones óptimas o cercanas a óptimas, pero pueden requerir más tiempo y ajuste de parámetros que los enfoques más simples como el Greedy.

El Sudoku es un problema de complejidad NP, lo que significa que encontrar una solución óptima puede requerir un tiempo exponencial en relación con el tamaño del tablero. Los algoritmos heurísticos como el Greedy, PSO y AG son útiles para encontrar soluciones aproximadas en un tiempo razonable, pero no garantizan la óptima en todos los casos.

En teoría, el algoritmo Greedy debería ser más rápido que el PSO y el AG debido a su enfoque simple y local para resolver el problema del Sudoku. Sin embargo, es probable que produzca soluciones subóptimas.

El PSO y el AG, aunque más lentos que el Greedy, tienen el potencial de encontrar soluciones de mayor calidad debido a su capacidad para explorar y explotar el espacio de búsqueda de manera más efectiva.

Para la comparación, utilizaremos 3 semillas diferentes, en nuestro caso las semillas representan un juego de sudoku a resolver, la primera será un sudoku de 9x9, en el cual, los valores con 0 son los valores en el que el algoritmo deberá buscar el número que corresponde en esa posición, los demás números representan pistas. El primer sudoku es este:

5 3 0 0 7 0 0 0 0

6 0 0 1 9 5 0 0 0

0 9 8 0 0 0 0 6 0

8 0 0 0 6 0 0 0 3

4 0 0 8 0 3 0 0 1

7 0 0 0 2 0 0 0 6

0 6 0 0 0 0 2 8 0

0 0 0 4 1 9 0 0 5

0 0 0 0 8 0 0 7 9

Nuestro algoritmo PSO, será comparado con un algoritmo Greedy y con un algoritmo de metaheurística.

Complejidad del Algoritmo:

Citas.

PSO: Tiene una complejidad moderada, con un enfoque basado en la optimización de la función fitness utilizando enjambres de partículas.

Algoritmo Genético: Su complejidad es moderada-alta, ya que involucra operadores de selección, cruzamiento y mutación en una población de cromosomas.

Algoritmo Greedy: Tiene una complejidad baja, ya que sigue un enfoque heurístico simple de seleccionar la mejor opción en cada paso sin considerar consecuencias futuras.

Calidad de la Solución:

PSO: Tiende a encontrar soluciones de alta calidad, ya que explora el espacio de búsqueda de manera global mediante enjambres de partículas.

Algoritmo Genético: Puede obtener soluciones de calidad, pero puede quedar atrapado en óptimos locales debido a su enfoque de búsqueda basado en la población.

Algoritmo Greedy: Tiende a encontrar soluciones subóptimas, ya que toma decisiones localmente óptimas en cada paso sin considerar el panorama general.

Tiempo de Ejecución:

PSO: Tiene un tiempo de ejecución moderado, ya que explora el espacio de búsqueda de manera eficiente utilizando técnicas de optimización inspiradas en la naturaleza.

Algoritmo Genético: Puede tener un tiempo de ejecución alto, especialmente en instancias complejas, debido a la generación y evaluación de múltiples poblaciones de cromosomas.

Algoritmo Greedy: Tiene un tiempo de ejecución bajo, ya que sigue un enfoque de toma de decisiones simple y local en cada paso.

Escalabilidad:

PSO: Es relativamente escalable y puede adaptarse a Sudokus de diferentes tamaños y niveles de dificultad.

Algoritmo Genético: Puede enfrentar problemas de escalabilidad en Sudokus de gran tamaño debido al aumento en la complejidad de la búsqueda en un espacio de soluciones más grande.

Algoritmo Greedy: Tiene dificultades para escalar a Sudokus de mayor tamaño o complejidad, ya que su enfoque local puede ser insuficiente para explorar el espacio de búsqueda de manera efectiva.

Robustez:

PSO: Tiene una buena robustez, ya que puede manejar diferentes configuraciones iniciales de Sudoku y converger hacia soluciones consistentes.

Algoritmo Genético: Puede ser robusto en algunos casos, pero su desempeño puede variar según la configuración de los parámetros y la calidad de la inicialización.

Algoritmo Greedy: Puede ser sensible a las condiciones iniciales y puede no encontrar soluciones óptimas en todas las instancias de Sudoku.

Comparación de Códigos

Inicializar

Los código utilizados para comparar nuestro algoritmo PSO, fueron sacados de github para posteriormente adaptarlos a nuestros problemas y así compararlos.

- **Metaheuristica:** https://github.com/totallyhuman/sudoku-solver/blob/master/SudokuSolver.py

- **Greedy:** https://github.com/MojTabaa4/genetic-algorithm/blob/main/sudoku_solver.ipynb

PSO (Optimización por Enjambre de Partículas):

Variables Relevantes:

Coeficientes de aprendizaje (c1, c2): Controlan la influencia de la mejor posición personal y global en el movimiento de las partículas.

Velocidades (v): Representan la magnitud y dirección del movimiento de cada partícula en el espacio de búsqueda.

Métodos Relevantes:

function_fitness(): Calcula la aptitud (fitness) de una solución Sudoku, evaluando la cantidad de valores únicos en filas, columnas y submatrices, al igual que contando el número de 0's (casillas vacías), contandolas como errores.

```python
def function_fitness(board):
    size = len(board)
    subgrid_size = int(size ** 0.5)  # Tamaño de las subgrillas
    fitness = 0

    # Evaluar cada fila
    for i in range(size):
        row_values = board[i]
        # Calcular los valores únicos en la fila
        unique_row_values = set(row_values)
        # Calcular la cantidad de valores únicos y restarlos del tamaño de la fila
        unique_row_count = len(unique_row_values)
        fitness += size - unique_row_count

        #cuenta los ceros (casillas vacias) por cada fila y los cuenta como errores
        cuenta0 = np.count_nonzero(row_values == 0)
        fitness += cuenta0

    # Evaluar cada columna
    for i in range(size):
        column_values = [board[j][i] for j in range(size)]
        # Calcular los valores únicos en la columna
        unique_column_values = set(column_values)
        # Calcular la cantidad de valores únicos y restarlos del tamaño de la columna
        unique_column_count = len(unique_column_values)
        fitness += size - unique_column_count

    # Evaluar cada submatriz
    for i in range(size):
        for j in range(size):
            subgrid_values = [board[i // subgrid_size * subgrid_size + k][j // subgrid_size * subgrid_size + l]
                              for k in range(subgrid_size) for l in range(subgrid_size)]
            # Calcular los valores únicos en la subseccion
            unique_subgrid_values = set(subgrid_values)
            # Calcular la cantidad de valores únicos y restarlos del tamaño de la submatriz
            unique_subgrid_count = len(unique_subgrid_values)
            fitness += size - unique_subgrid_count

    return fitness
```

generar_sudoku_inicial(): Genera una solución inicial aleatoria para una partícula del PSO, respetando las restricciones del Sudoku.

```python
def generar_sudoku_inicial(particle,size, board):
    #sudoku_inicial = [[0] * size for _ in range(size)]
    sudoku_inicial = particle
    subgrid_size = int(size ** 0.5)  # Tamaño de las subgmatriz

    for i in range(size):
        for j in range(size):
            if board[i][j] == 0:
                possible_values = [n for n in range(1, size + 1)]
                random.shuffle(possible_values)
                for value in possible_values:
                    if validar_sudoku(sudoku_inicial, i, j, value):
                        sudoku_inicial[i][j] = value
                        break

    return sudoku_inicial
```

PSO(): Este método implementa el algoritmo PSO. Primero se genera una población de partículas de soluciones pseudo-aleatorias para el sudoku.

```python
def PSO(sudoku_matriz, size, num_iterations, num_particles):
    # Establecer la semilla
    random.seed(123)
    c1=1
    c2=1
    rand=0.3
    board = leer_archivo(sudoku_matriz, size)
    subgrid_size = int(size ** 0.5)  # Tamaño de las submatrices

    particles = []
    global_best_board = None
    global_best_fitness = float('inf')

    # Inicializar partículas con soluciones aleatorias
    for _ in range(num_particles):
        particle = np.copy(board)
        particle = generar_sudoku_inicial(particle,size, board)
        particles.append(particle)

    Pbest = np.copy(particles)
    Pbest_fitness = np.array([function_fitness(p) for p in Pbest])

    best_index = np.argmin(Pbest_fitness)
    global_best_board = np.copy(Pbest[best_index])
    global_best_fitness = np.min(Pbest_fitness)

    #imprimir_mejor_solucion((p for p in Pbest), 1)
    #print(pbest_fitness)

    start_time = time.time()
```

Después, se calcula la velocidad de la partícula. Esta velocidad es normalizada en los valores. -1, 0 y 1. Cada casilla de cada solución de las partículas será afectada por un V, es decir, el resultado del cálculo de V será una matriz del mismo tamaño del sudoku con los valores mencionados anteriormente.

```python
        k=0

        for particula in particles:
            # Actualización de la velocidad y posición de la partícula
            v = rand * (c1 * np.random.rand() * (Pbest[k] - particula) + c2 * np.random.rand() * (global_best_board - particula))
            # Limitar la velocidad entre [-1, 1]

            #print(vl for vl in v)
            for vl in v:
                for i in range(len(vl)):
                    #print(f"{vc}\n")
                    if vl[i] <= -1:
                        vl[i] = -1
                    elif vl[i] >= 1:
                        vl[i] = 1
                    else:
                        vl[i] = 0

            #print(v)
            #imprimir_mejor_solucion(particula, fitness)
            particula = particula + v
            fitness = function_fitness(particula)
```

Luego de sumar los valores V a las correspondientes casillas de las partículas, nuevamente se evalúa la calidad de la nueva solución obtenida tras el movimiento de la partícula, tal que si la partícula mejoró su mejor solución conocida, entonces su mejor posición y fitness se actualiza.

```
            if fitness < Pbest_fitness[k]:
                print(fitness, Pbest_fitness[k],"!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")
                #print(particle)
                Pbest[k] = np.copy(particle)
                #print(pbest)
                Pbest_fitness[k] = fitness

            imprimir_mejor_solucion(Pbest[k], Pbest_fitness[k])
            k+=1

        best_index = np.argmin(Pbest_fitness)
        global_best_board = np.copy(Pbest[best_index])
        global_best_fitness = np.min(Pbest_fitness)

        if global_best_fitness ==0:
            break


        print(f"////////////////////    {iteration}    //////////////////////////////// {global_best_fitness}")

        #print("Iteración:", iteration, "Mejor fitness:", global_best_fitness)

    end_time = time.time()

    print("Tiempo de ejecución:", end_time - start_time, "segundos")
    imprimir_mejor_solucion(global_best_board, global_best_fitness)
    return Pbest_fitness
```

Algoritmo Genético:

Variables Relevantes:

Probabilidad de mutación (PM) y probabilidad de cruzamiento (PC): Controlan la tasa de mutación y la probabilidad de cruzamiento en el algoritmo genético.

Población (population): Representa el conjunto de individuos (cromosomas) en cada generación del algoritmo genético.

Parámetro	Valor
Tamaño de la población (POPULATION)	1000
Número de repeticiones (REPETITION)	1000
Probabilidad de mutación (PM)	0.1
Probabilidad de cruzamiento (PC)	0.95

Métodos Relevantes:

get_fitness(): Calcula la aptitud de un cromosoma del Sudoku evaluando la cantidad de valores únicos en filas, columnas y subgrillas.

crossover() y mutation(): Implementan los operadores genéticos de cruzamiento y mutación, respectivamente, para generar nuevos individuos en cada generación.

```python
# Parte 5: Implementando el Algoritmo Genético

def read_puzzle(address):
    puzzle = []
    f = open(address, 'r')
    for row in f:
        temp = row.split()
        puzzle.append([int(c) for c in temp])
    return puzzle

def r_get_mating_pool(population):
    fitness_list = []
    pool = []
    for chromosome in population:
        fitness = get_fitness(chromosome)
        fitness_list.append((fitness, chromosome))
    fitness_list.sort()
    weight = list(range(1, len(fitness_list) + 1))
    for _ in range(len(population)):
        ch = rndm.choices(fitness_list, weight)[0]
        pool.append(ch[1])
    return pool

def get_offsprings(population, initial, pm, pc):
    new_pool = []
    i = 0
    while i < len(population):
        ch1 = population[i]
        ch2 = population[(i + 1) % len(population)]
        x = rndm.randint(0, 100)
        if x < pc * 100:
            ch1, ch2 = crossover(ch1, ch2)
        new_pool.append(mutation(ch1, pm, initial))
        new_pool.append(mutation(ch2, pm, initial))
        i += 2
```

Algoritmo Greedy:

Variables Relevantes:

Valores del Sudoku (easy_values): Representan el estado inicial del Sudoku que se va a resolver utilizando el enfoque Greedy.

Grid (grid): Representa la estructura de datos utilizada para almacenar las celdas del Sudoku y sus valores correspondientes.

Métodos Relevantes:

calculate_possibilities(): Calcula las posibles opciones para cada celda vacía en el Sudoku, basándose en los valores presentes en las filas, columnas y subgrillas.

solve(): Implementa el enfoque Greedy para resolver el Sudoku, seleccionando la opción más prometedora en cada paso sin considerar el panorama general.

```
def calculate_possibilities(self):
    """
    calculate_possibilities()

    For each empty cell, find numbers that are not in its units.

    Returns:
    * did_something  -- a boolean which stores if the function made any
                        changes to a cell
    """
    did_something = False
    print("Calculating possibilities...")

    for key, value in self.grid.items():
        if value == 0 or type(value) is list:
            self.grid[key] = []
            location = self.locate_cell(key)

            for i in range(1, 10):
                if (i not in location[0] and i not in location[1] and i
                        not in location[2]):
                    self.grid[key].append(i)

            if len(self.grid[key]) == 1:
                self.grid[key] = self.grid[key][0]
                did_something = True

    return did_something

def solve(self):
```

Comparación de tiempos

Greedy:

El algoritmo Greedy es el más rápido en términos de tiempo de ejecución.Esto se debe a su enfoque directo y local para resolver el problema. El código del Greedy realiza cálculos simples y locales para cada celda del sudoku, sin necesidad de iteraciones complejas.

En el código del Greedy, observamos que simplemente calcula las posibilidades para cada celda y luego muestra la solución.

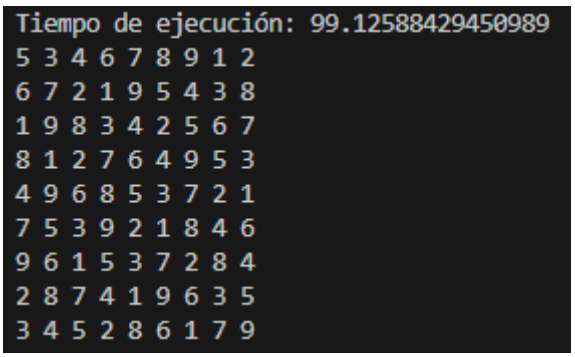
- El algoritmo comienza calculando las posibilidades para cada celda del sudoku.
- Luego, muestra la cuadrícula inicial y la solución encontrada después de calcular las posibilidades.
- La solución encontrada es correcta y coincide con la solución esperada.
- El tiempo transcurrido para encontrar la solución fue de aproximadamente 0.002 segundos.

Metaheurística (AG):

El algoritmo AG es el más lento en términos de tiempo de ejecución.El AG requiere múltiples iteraciones a través de generaciones para mejorar gradualmente la solución, lo que lleva a un tiempo de ejecución más largo.En el código del AG, observamos bucles que generan poblaciones, seleccionan padres, realizan cruzamientos y mutaciones, y evalúan la aptitud de las soluciones.

Estos procesos iterativos y la manipulación de poblaciones hacen que el tiempo de ejecución del AG sea el más largo de los tres algoritmos.

- Muestra la solución encontrada después de ejecutar el algoritmo genético.
- La solución encontrada es correcta y coincide con la solución esperada.
- El tiempo de ejecución fue de aproximadamente 99.13 segundos, mucho más largo que el tiempo del Greedy.



PSO:

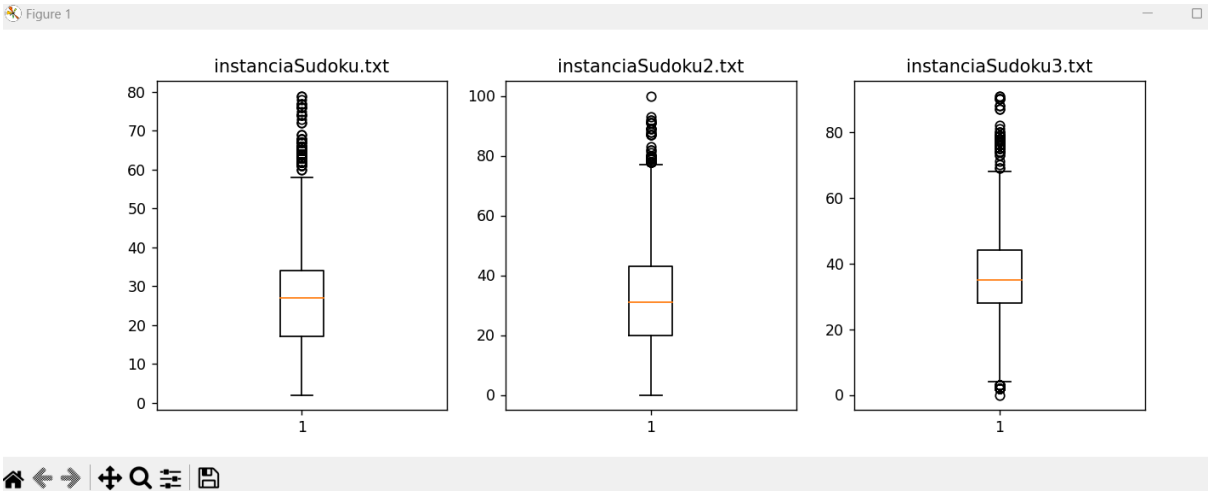
El algoritmo PSO es más lento en comparación con el Greedy pero más rápido que el AG.El tiempo de ejecución del PSO puede ser afectado por parámetros como el número de partículas y el número de iteraciones.En el código del PSO, podemos observar bucles que iteran a través de las partículas y las iteraciones para mejorar la solución.

Aunque el PSO es más rápido que el AG, su tiempo de ejecución sigue siendo significativamente más largo que el del Greedy debido a la complejidad adicional de las iteraciones y la búsqueda en el espacio de soluciones.

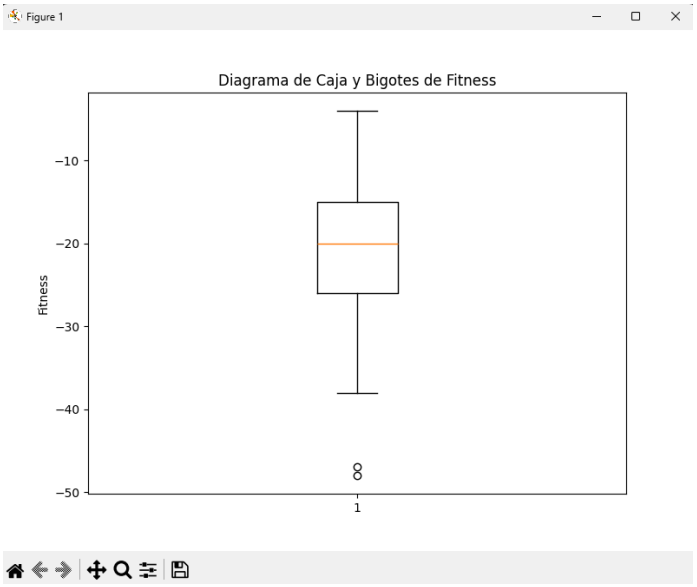
- Muestra la solución encontrada después de ejecutar el algoritmo PSO con un tamaño de 9, 10 iteraciones y 10000 partículas.
- La solución encontrada es correcta y coincide con la solución esperada.
- El tiempo de ejecución fue de aproximadamente 38.57 segundos, más rápido que el AG pero más lento que el Greedy.

```
Tiempo de ejecución: 38.56899905204773 segundos
                    [5 3 4 6 7 8 9 1 2]
                    [6 7 2 1 9 5 3 4 8]
                    [1 9 8 3 4 2 5 6 7]
                    [8 5 9 7 6 1 4 2 3]
                    [4 2 6 8 5 3 7 9 1]
                    [7 1 3 9 2 4 8 5 6]
                    [9 6 1 5 3 7 2 8 4]
                    [2 8 7 4 1 9 6 3 5]
                    [3 4 5 2 8 6 1 7 9]
Fitness:0
```

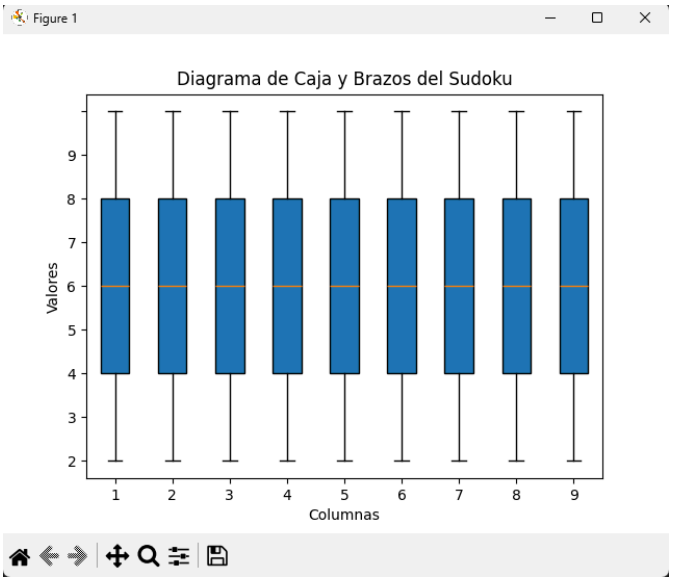
PSO



Metaheurística



Greedy



Análisis de resultados.

- El algoritmo Greedy es más rápido debido a que tiene un enfoque directo y local para resolver el problema. A diferencia del PSO que lo hace por medio de iteraciones más complejas.

Imagen. Tiempo que tomó ejecutar algoritmo greedy.

- El algoritmo de PSO requiere de parámetros por lo que eso lo hace un poco más tardado debido a que depende del número de partículas y el número de iteraciones.

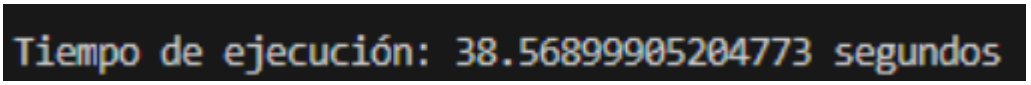


Imagen 2. Tiempo que tomó ejecutar algoritmo PSO.

Conclusiones.

- PSO es un algoritmo de optimización metaheurística basado en la inteligencia de enjambre, inspirado en el comportamiento social de los pájaros o peces. Los individuos, llamados partículas, se mueven a través del espacio de búsqueda en busca de soluciones óptimas mediante la cooperación y comunicación entre sí.
- Greedy es un enfoque de resolución de problemas que sigue una estrategia avariciosa, es decir, en cada paso, elige la opción óptima localmente sin considerar el panorama general o futuro.
- PSO puede ser más eficiente para problemas complejos con múltiples óptimos locales o problemas en los que la función objetivo es muy irregular.
- Greedy puede ser más eficiente para problemas simples o casos donde se sabe que la solución óptima se encuentra cerca de la solución actual.
- Ambos algoritmos funcionan para resolver problemas problemas de optimización solo que de maneras distintas.
- PSO puede ser más computacionalmente costoso ya que implica la actualización continua y la evaluación de múltiples partículas en cada iteración
- Greedy generalmente tiene una complejidad computacional más baja ya que solo involucra tomar decisiones locales en cada paso.
- PSO puede ser más computacionalmente costoso ya que implica la actualización continua y la evaluación de múltiples partículas en cada iteración.

- Greedy generalmente tiene una complejidad computacional más baja ya que solo involucra tomar decisiones locales en cada paso.

Referencias.

1. C.(2013, August 22). *Optimización por enjambre de partículas*. Wikipedia.org; Wikimedia Foundation, Inc. https://es.wikipedia.org/wiki/Optimizaci%C3%B3n_por_enjambre_de_part%C3%ADculas.
2. C.*(2005, June 20). rompecabezas de lógica. Wikipedia.org; Wikimedia Foundation, Inc. https://es.wikipedia.org/wiki/Sudoku*
3. *Rangel-Carrillo, E., Elvira-Ceja, S., Sánchez, E., Alanis, A., & Arana-Daniel, N. (n.d.). Optimización por Enjambre de Partículas para resolver el problema de control óptimo inverso en Seguimiento de Trayectorias. Retrieved February 13, 2024, from https://amca.mx/memorias/amca2014/media/files/0176.pd*
4. *kexugit. (2015, August 14). Inteligencia artificial - optimización del enjambre de partículas. Microsoft.com.https://learn.microsoft.com/es-es/archive/msdn-magazine/2011/august/artificial-intelligence-particle-swarm-optimization*
5. *Introducción — Optimización PSO 0.0.1 documentation. (2019). Github.io. https://joaquinamatrodrigo.github.io/optimizacion_PSO_python/pso.introduccion.html*