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

Ingeniería en Inteligencia Artificial, Mexican Axolotl Optimization
Fecha: 8 de junio de 2025

Mexican Axolotl Optimization

Presenta

Angeles López Erick Jesse[1]

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

Resumen

RESUMEN

En esta practica se describe el comportamiento, las partes esenciales, configuraciones, implementación y comparación de resultados de *Mexican Axolotl Optimization*, como un algoritmo bioinspirado utilizado para la búsqueda de óptimos globales en problemas de optimización.

Palabras clave: Axolote, Evolución, Resultado óptimo

[1] eangelesl1700@alumno.ipn.mx

Índice general

Introducción

Mexican Axolotl Optimization (*MAO*) es una técnica de optimización bio-inspirada que toma como referencia las características biológicas y comportamentales únicas del axolote mexicano. Esta especie se destaca por su notable capacidad de regeneración de órganos, su habilidad para el camuflaje, así como por su complejo sistema reproductivo que involucra poblaciones diferenciadas de machos y hembras, además de mecanismos de cruza genómica. Estas propiedades naturales han servido de modelo para diseñar un algoritmo que simula procesos de supervivencia, adaptación y evolución, proporcionando un enfoque novedoso para resolver problemas de optimización complejos [1].

En esta práctica, el objetivo principal es comprender en profundidad el funcionamiento interno de *MAO* y evaluar cómo sus principales metaparámetros influyen en su desempeño. Entre estos parámetros se encuentran la tasa de "aprendizaje", que controla la velocidad de adaptación del algoritmo; la probabilidad de daño y la probabilidad de sanación, que simulan procesos de deterioro y recuperación biológica; y el tamaño del torneo para cruza, que determina la presión selectiva durante la combinación genética. Estos elementos representan etapas esenciales del algoritmo y su correcta configuración es fundamental para lograr un rendimiento óptimo.

Esta exploración no solo contribuye a afianzar los fundamentos teóricos que sustentan *MAO*, sino que también permite desarrollar un criterio práctico para la selección y ajuste de sus componentes, de modo que se adapten de manera eficiente a distintos tipos de problemas. De esta manera, se busca potenciar la aplicabilidad del algoritmo y facilitar su implementación en contextos reales donde la optimización es un desafío constante.

Capítulo 1

Mexican Axolotl Optimization

Los dioses mexicas se reunieron en Teotihuacan para crear el universo ofreciendo su propia vida en sacrificio. Dioses como Huitzilopochtli, Xochipilli y Tezcatlipoca tomaron el sacrificio, sin embargo, Xolotl, victima del miedo, huyo del ritual.

Xolotl se transformo en múltiples criaturas para evitar se atrapado por el viento, se transformo en diferentes criaturas pero seguía siendo encontrado, por lo que opto por arrojarse al lago convirtiéndose en un anfibio con branquias en forma de cuernos, un axolote. Pudo sobrevivir algunos días en el lago pero finalmente fue atrapado por el viento y llevado al ritual para dar movimiento al universo [2].

1.1. Modelo natural

El axolote mexicano o *Ambystoma mexicanum* es una especie endémica del lago de Xochimilco, son salamandras que nunca superan su fase larvaria. Tiene branquias que brotan de su cabeza, patas palmeadas, una aleta dorsal, cola, pulmones funcionales, camuflaje y una sonrisa.

Los axolotes son un tema de investigación por biólogos dado a su capacidad de regenerar extremidades y órganos sin cicatrices permanentes [3].

1.2. Modelo artificial

El algoritmo *Mexican Axolotl Optimization* (**MAO**) busca imitar el comportamiento de estos anfibios para encontrar la mejor solución en un espacio de búsqueda dadas las siguientes analogías:

- Los ajolotes son soluciones de problemas y sus órganos y extremidades las dimensiones de la solución.

- El lago en donde viven es el espacio de búsqueda.
- El camuflaje del axolote, cuya efectividad depende del estado de búsqueda, determina la capacidad de sobrevivir, es decir, su aptitud.

En otras palabras, sea O un problema de optmización numerica cuyas elementos sea vectores de dimensión D acotados en el rango $[\text{mín}_i, \text{máx}_i]$, los axolotes representan un conjunto de soluciones de tamaño np: $P = \{S_1, \ldots, S_{np}\}$ donde $O(S_j) \in \mathbb{R}\ \forall j \in \{1, \ldots, np\}$.

El algoritmo MAO es un proceso iterativo de 4 etapas definida por el acronimo *TIRA*: transición de larva a adulto (*Transition*), lesión y restauración *Injury and restoration*, reproducción (*Reproduction*) y variedad (*Assortment*).

1.2.1. Transition

La población de axolotes se inicia de manera aleatoria y a cada individuo se le asigna un genero, obteniendo dos subconjuntos. Los individuos de cada grupo transicionan en el agua de larvas a adultos ajustando el color de las partes de su cuerpo para parecerse mas al mejor de cada grupo, como se muestra en el algoritmo 1

Algorithm 1 Transition
Input constante de diferenciación α, Población P, Función de optimización O
Output Población actualizada P'

El mejor axolote: $p_{best} = Best(P)$
for $p_j \in P$ **do**
    Probabilidad inversa de transición: $pp_j = \frac{O(p_j)}{\sum O(p_i)}$
    if $pp_j < r$ **then**
        Se aproxima al mejor: $p_{j,i} = p_{j,i} + (p_{best,i} - p_{j,i}) \cdot \alpha$
    else
        Se mueve de forma aleatoria: $p_{j,i} = \text{mín}_i + (\text{máx}_i - \text{mín}_i) * r_i$
    end if
end for

La habilidad de los ajolotes de cambiar de color esta limitada, por lo que unicamente se aproximan a la nueva solución, ya que no se busca que se adapten de forma perfecta al mejor de ellos. Cuando los individuos son difieren mucho de la mejor solución, medida que se obtiene con la probabilidad inversa, se mueven de manera aleatoria para explorar el espacio.

1.2.2. Injury and restoration

Mientras los axolotes se mueven en el agua pueden sufrir accidentes y ser lastimados. En esta fase los ajolotes pierden partes que posteriormente regene-

ran de forma aleatoria, para esto, se consideran ambas probabilidades donde la primera actuá sobre los axolotes que serán lastimados y la segunda sobre las partes del cuerpo que serán regeneradas, esto se aplica para ambos grupos como se muestra en el algoritmo 2.

Algorithm 2 Injury and Restoration
Input Población P, probabilidad de daño dp, probabilidad de regeneración rp
Output Población actualizada P'

```
for p_j ∈ P do
    if r ≤ dp then
        for i ∈ D do
            if r ≤ rp then
                p_{j,i} = mín_i + (máx_i − mín_i) * r_i
            end if
        end for
    end if
end for
```

1.2.3. Reproduction and Assortment

Los machos y las hembras se reproducen dejando un par de huevos que contienen una mezcla del material genético de los padres de forma uniforme. Se genera una cadena aleatoria de tamaño D y si es un 0 el primer hijo hereda el gen del padre y el dos de la madre, si es un 1 el primer hijo hereda del gen de la madre y el dos del padre, como se muestra en la figura 1.1.

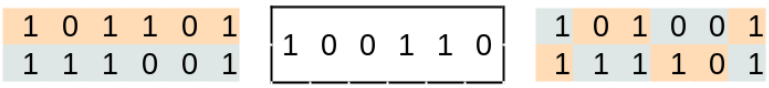

Figura 1.1: Cruza uniforme

El algoritmo 3 selecciona, entre un rango de machos, con quien reproducirse mediante un torneo de tamaño k con cada hembra, cuyos huevos heredan el material genético. Finalmente, entre los 4 individuos se seleccionan los dos mejores y se les asigna el genero de hembra y macho al primero y segundo mejor respectivamente.

Algorithm 3 Reproduction
Input Población P, probabilidad de daño dp, probabilidad de regeneración rp
Output Población actualizada P'

for $f \in F$ **do**
  $m = \{Best(MC) | MC \subset M \wedge |MC| = k\}$
  egg_1
  egg_2
  for $i \in D$ **do**
    if $r \leq 0{,}5$ **then**
      $\text{huevo}_{1,i} = m_i$
      $\text{huevo}_{2,i} = f_i$
    else
      $\text{huevo}_{1,i} = f_i$
      $\text{huevo}_{2,i} = m_i$
    end if
  end for
end for
$V = Sort(function = O, \{f, m, \text{huevo}_1, \text{huevo}_2\})$
$f = V[0]$
$m = V[1]$

1.3. Ventajas

La técnica *Mexican Axolotl Optimization* (*MAO*) presenta varias ventajas importantes:

- **Inspiración biológica robusta:** Su diseño se basa en procesos naturales probados, como la regeneración y adaptación, lo que permite un enfoque balanceado entre exploración y explotación.
- **Capacidad de adaptación dinámica:** Gracias a sus parámetros como la probabilidad de daño y sanación, el algoritmo puede adaptarse a distintos escenarios y escapar de óptimos locales.
- **Manejo efectivo de poblaciones heterogéneas:** La diferenciación entre machos y hembras y la cruza genómica permiten mantener diversidad genética, lo que mejora la búsqueda global.
- **Versatilidad:** Puede ser aplicado a problemas de optimización continua, compleja y de alta dimensión.

1.4. Desventajas

Sin embargo, *MAO* también presenta algunas limitaciones:

- **Sensibilidad a la configuración:** La selección incorrecta de metaparámetros puede degradar significativamente el rendimiento del algoritmo.
- **Costo computacional:** El manejo de poblaciones grandes y múltiples iteraciones puede resultar en tiempos de cómputo elevados, especialmente para problemas complejos.
- **Falta de garantías teóricas:** Como muchas heurísticas bioinspiradas, no ofrece garantías matemáticas de convergencia al óptimo global.
- **Implementación compleja:** La modelación de procesos biológicos complejos puede dificultar la implementación y ajuste del algoritmo para usuarios sin experiencia.

1.5. Aplicaciones

El algoritmo *Mexican Axolotl Optimization* ha demostrado ser útil en diversas áreas de optimización, tales como:

- Optimización de funciones continuas y no lineales. Resuelve problemas complejos con múltiples variables y restricciones.
- Selección de parámetros óptimos en modelos de machine learning.
- Inspiración para simulaciones en biología computacional y estudios de dinámica poblacional.

Capítulo 2

Problemas

2.1. Knapsack problem

Dado un conjunto de n ítems

$$I = \{1, 2, \ldots, n\}$$

Donde cada ítem i tiene un valor $v_i \geq 0$ y un peso $w_i \geq 0$ y dada una mochila con capacidad máxima W, se busca seleccionar un subconjunto de ítems que maximice el valor total sin exceder la capacidad.

Podemos representar los elementos dentro de la mochila como un vector binario:

$$x = (x_1, x_2, \ldots, x_n) \text{ con } x_i \in \{0, 1\}$$

Donde:

- $x_i = 0$ si el ítem no esta en la mochila
- $x_i = 1$ si el ítem si esta en la mochila

Para calcular el valor $v(x)$ y el peso $w(x)$ de la mochila sumamos los valores que si se encuentren dentro de ella:

$$v(x) = \sum_{i=1}^{n} v_i x_i$$

$$w(x) = \sum_{i=1}^{n} w_i x_i$$

El objetivo, es encontrar el mayor $v(x)$ siempre que el peso $w(x)$ no exceda el peso máximo W.

- El conjunto de estados posibles son todas las cadenas binarias de tamaño n:

$$S = \{x \in \{0, 1\}^n\}$$

- El estado inicial puede ser cualquier cadena de tamaño n cuyo peso no exceda el peso máximo:

$$s_0 = \{x \in \{0,1\}^n | w(x) \leq W\}$$

- Se busca maximizar el valor de la mochila. La función objetivo suma los valores de los objetos dentro de la mochila. Si el peso de la mochila excede el limite, entonces se le asigna una ganancia negativa.

$$f(x) = \begin{cases} v(x), & \text{si } w(x) \leq W \\ W - w(x), & \text{si } w(x) > W \end{cases}$$

  Se le asigna la diferencia del peso máximo menos el peso actual (Dando un numero negativo). Esto con el objetivo de que, si por alguna razón esa es la mejor solución actual, sepa encontrar una mejor solución disminuyendo esa diferencia.

- Entonces, un estado x_j es un estado final si genera mayor aptitud en comparación de los demás x_i generados y tiene una aptitud no negativa:

$$f(x_j) \geq 0 \wedge f(x_j) \geq f(x_i)\ \forall x_i \in S$$

- La operación que genere genere el vecino sera *Bit flip* que intercambia un 0 por un 1 y viceversa en una posición aleatoria i).

$$B(x_i) = \begin{cases} 1, & \text{si } x_i = 0 \\ 0, & \text{si } x_i = 1 \end{cases}$$

2.2. Minimizar la función

Obtener los mínimos de la función

$$f(x) = \sum_{i=1}^{D} x_i^2, \text{ con } -10 \geq x_i \geq 10$$

.

Dado un vector de D números en el rango de $[-10, 10]$, se busca obtener el valor mínimo del sumatoria de sus cuadrados.

- El conjunto de estados posibles son todas las cadenas de enteros en dicho intervalo:

$$S = \{x \in [-10, 10]^n\}$$

- El estado inicial se genera de forma arbitraria como un vector de D números en el rango establecido $[-10, 10]$
- La función objetivo unicamente considera los valores dentro del propio vector:
$$f(x)$$
- Un estado de aceptación x_j es aquel que produzca el menor valor de aptitud en la función comparando con los demás x_i generados:
$$f(x_j) \leq f(x_i)\ \forall x_i \in S$$
- La operación que genere los vecinos puede tener multiples interpretaciones. Para este problema se asume un espacio circular donde -10 es el consecutivo del 10 y que $\forall d_i \in D, d_i \in \mathbb{Z}$. Entonces, los vecinos de d_i son los números consecutivos, es decir d_{i-1} y d_{i+1}.

  La operación sera entonces:
$$d_i = min(f(d_{i-1}), f(d_i), f(d_{i+1}))$$

2.3. Problemas de optimización CEC 2017

En el documento [4] se presentan una serie de problemas sobre optimización numérica de parámetros reales. En este reporte se analizan las 10 primeras funciones que cumplen con la siguiente definición:

- Todas las funciones son problemas de minimización definidos de la siguiente manera:
$$minf(x),\ x = [x_1, x_2, \ldots, x_D]^T$$
  Donde:
  - x es el vector de variables de dimensión D que representa la solución del problema.
  - D es el numero de dimensiones del problema.
- El óptimo global (la mejor solución) se encuentra desplazada del origen para evitar respuestas que asumen que la respuesta esta cerca del origen:
$$o = [o_1, o_2, \ldots, o_D]^T$$
  Donde o es el vector del optimo global desplazado.

  El valor óptimo se distribuye de manera aleatoria en el rango de $o \in [-80, 80]^D$

- Las funciones son escalables, es decir, el numero de dimensiones D puede variar.
- El rango de búsqueda de todas las funciones para las variables se delimita por $x \in [-100, 100]^D$
- Implementación de matrices de rotación: Las variables interactúan entre ellas para volver el problema más difícil.
- Para simular problemas reales, las variables se dividen de manera aleatoria en subcomponentes. Cada subcomponente tiene su propia matriz de rotación.

2.3.1. Funciones

A continuación se definen las 10 primeras funciones.

1) Bent Cigar Function

$$f(x) = x_1^2 + 10^6 \sum_{i=2}^{D} x_i^2$$

2) Zakharov Function

$$f(x) = \sum_{i=1}^{D} x_i^2 + \left(0{,}5 \sum_{i=1}^{D} i x_i\right)^2 + \left(0{,}5 \sum_{i=1}^{D} i x_i\right)^4$$

3) Rosenbrock's Function

$$f(x) = \sum_{i=1}^{D-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$

4) Rastrigin's Function

$$f(x) = \sum_{i=1}^{D} \left[x_i^2 - 10\cos(2\pi x_i) + 10\right]$$

5) Expanded Schaffer's F6 Function

$$g(x,y) = 0{,}5 + \frac{\sin^2(\sqrt{x^2+y^2}) - 0{,}5}{(1+0{,}001(x^2+y^2))^2}$$

$$f(x) = \sum_{i=1}^{D-1} g(x_i, x_{i+1})$$

6) Lunacek Bi-Rastrigin Function

$$f(x) = \text{mín}\left(\sum_{i=1}^{D}(x_i-\mu_0)^2, dD + s\sum_{i=1}^{D}(x_i-\mu_1)^2\right) + 10\sum_{i=1}^{D}\left[1-\cos(2\pi z_i)\right]$$

$$\mu_0 = 2{,}5, \quad \mu_1 = -\sqrt{\frac{\mu_0^2}{d}}$$

7) Non-Continuous Rotated Rastrigin's Function

$$f(x) = \sum_{i=1}^{D}\left[z_i^2 - 10\cos(2\pi z_i) + 10\right]$$

$$z_i = \text{Tosz}(\text{Tasy}(x_i))$$

8) Levy Function

$$f(x) = \sin^2(\pi w_1) + \sum_{i=1}^{D-1}(w_i-1)^2\left[1+10\sin^2(\pi w_i+1)\right] + (w_D-1)^2\left[1+\sin^2(2\pi w_D)\right]$$

$$w_i = 1 + \frac{x_i - 1}{4}$$

9) Modified Schwefel's Function

$$f(x) = 418{,}9829D - \sum_{i=1}^{D} x_i \sin(\sqrt{|x_i|})$$

10) High Conditioned Elliptic Function

$$f(x) = \sum_{i=1}^{D} 10^{6\frac{i-1}{D-1}} x_i^2$$

Cuyas graficas se observan en la figura 2.1

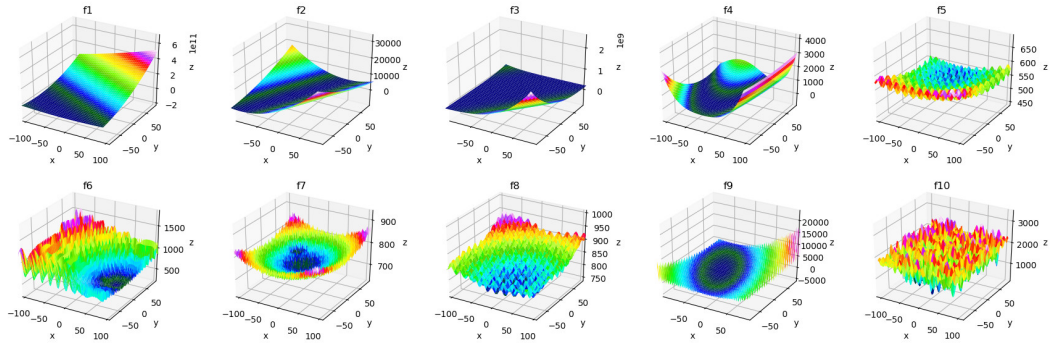

Figura 2.1: Superficies ploteadas de las 10 primeras funciones para dos dimensiones [5]

Capítulo 3

Codigo

3.1. Problem

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

El código 3.1 incluye los siguientes métodos los cuales deben de ser implementados por sus clases hijas:

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

Listing 3.1: Clase *problem*

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

```
6     def generateInformation(self, *args, **kwargs) ->
          None: pass
7
8     @abstractmethod
9     def objective(self, solution: Any) -> float: pass
10
11    @abstractmethod
12    def generateInitialSolution(self) -> Any: pass
13
14    @abstractmethod
15    def normalizeSolution(self, solution: Any) -> Any:
          return solution
16
17    @abstractmethod
18    def getRandomNeighbour(self, solution:Any) -> Any:
          pass
19
20    @abstractmethod
21    def getNextNeighbour(self, solution:Any, *args, **
          kwargs) -> Any: pass
22
23    @abstractmethod
24    def getNeighbours(self, solution: Any) -> list: pass
25
26    @abstractmethod
27    def printInformation(self) -> None: pass
```

3.1.1. Knapsack problem

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

La función 3.2 genera de manera aleatoria un conjunto de elementos en la mochila con pesos y valores aleatorios en un rango de $[1, 10]$. La función 3.3 calcula la energía del sistema la cual suma todos los pesos y valores de los elementos que se encuentran en la solución, si el peso es menor al capacidad de la mochila entonces devuelve el valor de la mochila, en caso contrario devuelve la diferencia de el peso actual menos la capacidad máxima.

La función 3.4 genera soluciones aleatorias de combinaciones y no regresa ninguna de ellas hasta que el peso de la solución sea menor a la capacidad máxima. Finalmente, la función 3.5 se encarga de generar un vecino de forma aleatoria, primero selecciona un elemento aleatorio del vector y después lo invierte.

Listing 3.2: Función *generateInformation* de Knapsack problem

```
    def generate_information(self, items, capacity):
    self.information = {
        "items": items,
        "values": [(random.randint(1,10), random.randint
           (1, 10)) for _ in range(items)],
        "capacity": capacity
    }
```

Listing 3.3: Función *objective* de Knapsack problem

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

Listing 3.4: Función *generateInitialSolution* de Knapsack problem

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

Listing 3.5: Función *getRandomNeighbour* de Knapsack problem

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

Para *MAO*, la función *normalize solution devuelve un valor binario si son mayores o menores 0.5.*

3.1.2. Sum function Problem

La función 3.6 unicamente define el tamaño del vector y los rangos de valores. por otro lado, la función 3.7 calcula la energía del sistema dada por la suma de los cuadrados, dado que es una función de minimización se invierte el signo.

La función 3.8 genera un vector de n elementos aleatorios en los rangos definidos, mientras que la función 3.9 suma o resta en uno a un elemento

aleatorio del vector (dado que se considera una configuración circular, se ajusta el valor si el nuevo valor no se encuentra en el rango).

Listing 3.6: Función *generateInformation* de SumFunctionProblem

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

Listing 3.7: Función *objective* de SumFunctionProblem

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

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

    return -total_sum
```

Listing 3.8: Función *generateInitialSolution* de SumFunctionProblem

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

Listing 3.9: Función *getRandomNeighbour* de CEC 2017

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

    if neighbour[index] > self.information['max']:
    neighbour[index] = self.information['min']

    if neighbour[index] < self.information['min']:
    neighbour[index] = self.information['max']

    return neighbour
```

La función *normalizeSolution* redondea la solución a un valor entero.

3.1.3. CEC 2017

La función 3.10 define la función a optimizar, los rangos de valores, el tamaño de la dimensión y la tasa de cambio (ya que la solución es real). Por otro lado, la función 3.11 calcula la energía del sistema dados los parámetros ya definidos.

La función 3.12 genera un vector de la dimensión definida restringida por la información adicional definida, mientras que la función 3.13 suma o resta en *alpha* a un elemento aleatorio del vector.

Listing 3.10: Función *generateInformation* de CEC 2017

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

Listing 3.11: Función *objective* de CEC 2017

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

Listing 3.12: Función *generateInitialSolution* de CEC 2017

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

Listing 3.13: Función *getRandomNeighbour* de CEC 2017

```
1    def getRandomNeighbour(self, solution):
2    neighbour = solution[:]
3    index = random.randint(0, len(solution) - 1)
4    alpha = random.uniform(-self.information["alpha"],
        self.information["alpha"])
5
6    neighbour[index] += alpha
7    return neighbour
```

3.2. Metaheuristic

De igual manera, el se diseña una clase padre **??** que implemente algunas de las funciones *hill climbing*, *simulated annealing*, *genetic algorithm* y *Mexican axolotl optimization*.

3.2.1. Mexican Axolotl Optimization

Se diseña una clase *Axolotl* que hereda de la clase *Metaheuristic* y recibe los siguiente meta-parámetros:

- Tamaño de la población
- Probabilidad de daño.
- Probabilidad de regeneración.
- Tamaño del torneo.
- Tasa de aprendizaje α.

Se definen la funciones que se ejecutan en cada paso.

Transition

La función 3.14 recibe un subconjunto de la población (machos o hembras) y encuentra los valores de aptitud de todos los individuos y define el mejor para compararlo con el resto. Se calcula la probabilidad inversa de transición y si supera un umbral aleatorio se aproxima al mejor, en caso contrario se mueve de manera aleatoria. Al finalizar es necesario aplicar una función de normalización de resultados, ya que se puede utilizar para resolver problemas en espacios discretos.

Listing 3.14: Función *transition*

```
def transition(self, population):
  best = self.bestIndividual(population)
  values = [ self.problem.objective(individual) for
     individual in population ]
  total = sum(values)

  for i, individual in enumerate(population):
    objective = values[i]
    probability = objective / total if total else 0
    r = random.random() / len(population)

    if probability < r:
```

```
12        for j in range(len(individual)):
13          individual[j] = individual[j] + (best[j] -
              individual[j]) * self.alpha
14      else:
15        population[i] = self.problem.getRandomNeighbour(
            individual)
16
17    population[i] = self.problem.normalizeSolution(
        individual)
```

Injury And Restoration

Después, a los conjuntos de machos y hembras se les aplica la función 3.15 que itera sobre una primera probabilidad para ser candidato a mutaciones y después se itera cada dimensión para modificar su valor dada una probabilidad.

Listing 3.15: Función *injuryAndRestoration*

```
1 def injuryAndRestoration(self, population):
2   for i, individual in enumerate(population):
3     if random.random() < self.damage:
4       for _ in individual:
5         if random.random() < self.regeneration:
6           population[i] = self.problem.getNextNeighbour(
              individual, i)
```

Uniform

Para la reproducción es necesario definir la función *uniform* que define el comportamiento de como se mezclan las dimensiones del macho y la hembra para generar dos nuevas criás. El código 3.16 genera una secuencia binaria aleatoria de tamaño D para después generar los nuevos hijos (Como en la figura 1.1).

Listing 3.16: Función *Uniform*

```
1 def uniform(self, male, female):
2   child1 = []
3   child2 = []
4
5   for i in range(len(male)):
6     if random.random() < 0.5:
7       child1.append(male[i])
8       child2.append(female[i])
9     else:
```

```
10        child1.append(female[i])
11        child2.append(male[i])
12
13    population = [ male, female, child1, child2 ]
14    population.sort(key=self.problem.objective, reverse=
          True)
15    return population[0:2]
```

Reproduction

La función del código 3.17 itera sobre todas las hembras y obtiene una sub-población de machos de forma aleatoria para reproducirse y poner dos huevos. Entre el macho, la hembra y los dos huevos se seleccionan los dos mejores, los cuales parte de la siguiente población de hembras y de machos (respectivamente).

Listing 3.17: Función *Reproduction*

```
1  def reproduction(self):
2    new_males = []
3    new_females = []
4    for female in self.females:
5      males = random.sample(self.males, self.
           tournament_size)
6      best_male = self.bestIndividual(males)
7      [ new_female, new_male ] = self.uniform(best_male,
           female)
8      new_males.append(new_male)
9      new_females.append(new_female)
10
11   self.males = new_males
12   self.females = new_females
```

Capítulo 4

Resultados

A continuación se desarrollan las mejores configuracion de *MAO* y se compara su rendimiento contra *genetic algorithm* (con sus mejores configuraciones) para los problemas descritos.

4.1. Problemas

A continuación se presentan las mejores combinaciones para los problemas considerando 50 entrenamientos diferentes con 50 épocas.

4.1.1. knapsack Problem

Considerando 50 elementos en la mochila, una capacidad máxima de 50, los mejores resultados se obtienen con las configuraciones de la tabla 4.1.

Mejor	***best***	***worst***	***mean***	***std dev***
Damage	0.7	0.0	0.7	0.9
Regeneration	0.1	0.4	0.1	0.4
Tournament	2	2	2	2
Alpha	0.1	0.3	0.1	0.1
best score	133.0000	133.0000	125.0000	132.0000
worst score	110.0000	98.0000	110.0000	114.0000
mean score	125.0000	116.6800	125.0000	124.8000
std dev	5.3871	7.6357	5.3871	4.8906

Tabla 4.1: Mejores configuraciones de parámetros para el problema KSP con el algoritmo *MAO*

Considerando 250 pruebas con 250 epocas de entrenamiento, una población de 32 (ambos) y *genetic algorithm* con la siguiente configuración:

- Estacionario: 0,1 %

- Selección *Negative assortative*
- Cruza: *Two point*
- Reemplazo: *restricted tournament*

Se obtuvieron los siguientes resultados:

Algoritmo	**Best**	**Worst**	**Mean**	**StdDev**
MAO	151	71	111.21	15.5682
GA	146	79	115.87	14.1604

Tabla 4.2: Estadísticas comparativas entre *MAO* y *GA* para el problema *Knapsack problem*

4.1.2. Sum Function problem

Considerando 100 elementos en un rango de $[-100, 100]$, los mejores resultados se obtienen con las configuraciones de la tabla 4.3. Nótese que ninguna de las configuraciones alcanzó la mejor solución de 0, esto es por el numero de épocas de entrenamiento realizadas son menores al tamaño del problema, diseñado de esta forma para poder analizar las diferencias.

Mejor	***best***	***worst***	***mean***	***std dev***
Damage	0.9	0.3	0.6	1.0
Regeneration	1.0	0.2	1.0	0.9
Tournament	2	2	2	2
Alpha	0.5	0.0	0.5	0.5
best score	-12914	-51141	-17622	-21457
worst score	-74623	-82841	-71509	-60826
mean score	-43794.4000	-66939.1400	-39172.7600	-41366.4800
std dev	13824.4655	7925.9986	13103.4330	10032.3798

Tabla 4.3: Mejores configuraciones de parámetros para el problema SFP con el algoritmo *MAO*

Considerando *genetic algorithm* con la siguiente configuración:

- Estacionario: 0,3 %
- Selección *Negative assortative*
- Cruza: *Blend*
- Reemplazo: *restricted tournament*

Se obtuvieron los siguientes resultados:

Algoritmo	**Best**	**Worst**	**Mean**	**StdDev**
MAO	-71	-246	-143.41	27.9164
GA	-278.81	-1061.63	-515.30	120.6011

Tabla 4.4: Estadísticas comparativas entre *MAO* y *GA* para el problema *Sum Function Problem (SFP)*

4.1.3. CEC 2017

Dada la complejidad de lo problemas y el tiempo computacional, se realizaron 100 entrenamientos con 100 épocas cada uno. Las mejores configuraciones para cada problema se encuentra en la tabla 4.5.

F	***D***	**R**	**T**	α	***best score***
f1	1.0	0.8	2.0	0.2	-5.8544e+11
f2	1.0	0.8	10.0	0.6	-1.5942e+78
f3	0.6	0.8	2.0	0.4	-3.0364e+05
f4	1.0	1.0	2.0	0.0	-5.8416e+03
f5	0.8	0.2	6.0	0.6	-1.3505e+03
f6	0.8	0.6	2.0	0.4	-6.5909e+02
f7	0.8	1.0	2.0	0.2	-2.5530e+03
f8	0.4	0.6	4.0	0.2	-1.6529e+03
f9	0.2	0.0	2.0	0.2	-2.8855e+04
f10	0.8	0.2	6.0	0.4	-1.8282e+04

Tabla 4.5: Mejores configuraciones de funciones para los problemas del *cec 2017*.

Ahora, dada la mejor configuración para cada problema, se realizan las pruebas para obtener los resultados y los tiempos de ejecución en las tabla 4.6 y 4.7.

f(x)	Peor	Mejor	Promedio	Mediana	Desviación estándar
f1	$-1{,}02e11$	$-5{,}49e11$	$-3{,}26e11$	$-3{,}17e11$	$1{,}07e11$
f2	$-8{,}79e116$	$-6{,}37e134$	$-3{,}19e133$	$-2{,}28e125$	$1{,}39e134$
f3	$-3{,}83e5$	$-6{,}76e5$	$-5{,}14e5$	$-5{,}26e5$	$7{,}74e4$
f4	$-2{,}73e3$	$-5{,}25e3$	$-3{,}99e3$	$-3{,}95e3$	$7{,}57e2$
f5	$-1{,}34e3$	$-1{,}81e3$	$-1{,}65e3$	$-1{,}68e3$	$1{,}21e2$
f6	$-6{,}71e2$	$-7{,}11e2$	$-6{,}91e2$	$-6{,}93e2$	$1{,}08e1$
f7	$-2{,}15e3$	$-3{,}14e3$	$-2{,}49e3$	$-2{,}38e3$	$2{,}54e2$
f8	$-1{,}60e3$	$-2{,}36e3$	$-1{,}82e3$	$-1{,}80e3$	$1{,}66e2$
f9	$-4{,}12e4$	$-8{,}90e4$	$-5{,}95e4$	$-5{,}68e4$	$1{,}48e4$
f10	$-1{,}72e4$	$-2{,}34e4$	$-2{,}13e4$	$-2{,}15e4$	$1{,}30e3$

Tabla 4.6: Estadísticas de energía por función.

f(x)	Peor	Mejor	Promedio	Mediana	Desviación estándar
f1	0.253	0.233	0.237	0.236	0.0045
f2	0.428	0.416	0.422	0.421	0.0038
f3	0.280	0.271	0.274	0.273	0.0020
f4	0.281	0.276	0.277	0.277	0.0015
f5	0.310	0.293	0.301	0.298	0.0064
f6	0.306	0.293	0.299	0.300	0.0031
f7	0.447	0.428	0.439	0.440	0.0055
f8	0.353	0.344	0.346	0.346	0.0019
f9	0.337	0.311	0.315	0.313	0.0053
f10	0.557	0.532	0.545	0.546	0.0069

Tabla 4.7: Estadísticas de tiempo de ejecución por función.

4.2. Discusión

Para el problema de *knapsack problem* el algoritmo de *MAO* logro encontrar un mejor resultado por encima de *genetic algorithm* con una diferencia de 5. Sin embargo, pese a encontrar un mejor resultado, tiene una media ligeramente menor, lo que significa que en algunos casos particulares es en donde encuentra el mejor resultado.

Para el problema de *sum function problem*, *MAO* encuentra un resultado mucho mejor que *genetic algorithm* cada unocon su mejor configuración. También, entre mayor sea el daño y la regeneración mejores resultados se obtuvieron, lo que indica que la exploración es fundamental para este problema y, de igual forma que con *knapsack problem*, un torneo de 2 resulta la mejor

solución.

Finalmente para los problemas de *CEC 2017* las funciones funciones son aquellas que se dedicaron a la exploración y en casos particulares se incremento el tamaño del torneo (funciones $f2, f3, f5, f8, f10$). Por otro lado, los resultados obtenidos aplicando las configuraciones a cada una de las funciones no garantizan que se obtenga el mejor resultado, ya que comparando los resultados de *genetic algorithm* hubo algunas funciones en donde se obtuvieron mejores resultados y en otras no. Sin embargo, la diferencia del mejor resultado y de tiempo de procesamiento no son muy diferentes.

Conclusiones

En esta practica se exploro el comportamiento del algoritmo *Mexican Axolotl Optimization*, un método de optimización inspirado en el ciclo de vida y las bondades genéticas presentes en los axolotes. Este enfoque considera 4 etapas esenciales: *transition*, *injury and restoration*, *reproduction* y *assortment* y se implementaron para buscar soluciones en espacios de distintos tipos de problemas: *knapsack problem*, *sun function problem* y los problemas del *CEC 2017*.

Adicionalmente se probaron múltiples configuraciones de meta parámetros y mediante fuerza bruta encontrar la mejor configuración para cada problema. Esto indica que, de igual forma que con *genetic algorithm*, cada problema requiere un tratamiento especial y es fundamental entender el porque ocurren ocurren estos resultados. El enfoque de esta practica unicamente busca mostrar los resultados de dichas configuraciones.

Este algoritmo fue diseñado para espacios reales pero, sorpresivamente, tiene un buen rendimiento en problemas enteros y binarios. Lo que abre la posibilidad incluso de resolver problemas de permutación.

MAO tiene un rendimiento similar a *genetic algorithm* (y en ocasiones superior), pero de igual manera su rendimiento requiere de un ajuste de meta parámetros.

Bibliografía

[1] Y. Villuendas-Rey, J. L. Velázquez-Rodríguez, M. D. Alanis-Tamez, M.-A. Moreno-Ibarra, and C. Yáñez-Márquez, "Mexican axolotl optimization: A novel bioinspired heuristic," *Mathematics*, vol. 9, no. 7, p. 781, Apr. 2021, accedido el 8 de junio de 2025. [Online]. Available: https://doi.org/10.3390/math9070781

[2] Acuario Michin Puebla. (2025) La leyenda del axolote mexicano, un dios que se negó a morir. Acuario Michin Puebla. Accedido el 5 de junio de 2025. [Online]. Available: https://puebla-es.acuariomichin.com/la-leyenda-del-axolote-mexicano-un-dios-que-se-nego-a-morir/

[3] National Geographic España. (2025) Axolote mexicano, datos esenciales de un animal único. National Geographic España. Accedido el 5 de junio de 2025. [Online]. Available: https://www.nationalgeographic.es/animales/axolote-mexicano

[4] Y. S. Ong, P. N. Suganthan, and T. Weise, "Definitions of cec2017 benchmark suite," 2017, accedido el 26 de marzo de 2025. [Online]. Available: ruta/local/o/enlace

[5] N. H. Awad, M. Z. Ali, P. N. Suganthan, J. J. Liang, and B. Y. Qu, "Problem definitions and evaluation criteria for the cec 2017 special session and competition on single objective bound constrained real-parameter numerical optimization," Tech. Rep., 2016, accedido el 26 de marzo de 2025.

Anexos

f(x)	Peor	Mejor	Promedio	Mediana	Desviación estándar
f1	$-3{,}92e12$	$-5{,}67e12$	$-5{,}00e12$	$-5{,}11e12$	$4{,}53e11$
f2	$-8{,}71e175$	$-2{,}62e191$	$-1{,}98e190$	$-4{,}65e183$	∞
f3	$-6{,}53e5$	$-1{,}21e6$	$-9{,}24e5$	$-9{,}39e5$	$1{,}43e5$
f4	$-1{,}28e5$	$-2{,}80e5$	$-2{,}22e5$	$-2{,}30e5$	$3{,}88e4$
f5	$-2{,}48e3$	$-2{,}88e3$	$-2{,}71e3$	$-2{,}70e3$	$1{,}01e2$
f6	$-7{,}34e2$	$-7{,}76e2$	$-7{,}51e2$	$-7{,}50e2$	$1{,}05e1$
f7	$-1{,}01e4$	$-1{,}25e4$	$-1{,}14e4$	$-1{,}15e4$	$6{,}89e2$
f8	$-2{,}94e3$	$-3{,}45e3$	$-3{,}10e3$	$-3{,}10e3$	$1{,}17e2$
f9	$-7{,}03e4$	$-1{,}20e5$	$-8{,}76e4$	$-8{,}56e4$	$1{,}13e4$
f10	$-2{,}98e4$	$-3{,}33e4$	$-3{,}12e4$	$-3{,}11e4$	$8{,}43e2$

Tabla 4.8: Estadísticas de energía por función mediante *genetic algorithm*.

f(x)	Peor	Mejor	Promedio	Mediana	Desviación estándar
f1	0.294	0.274	0.282	0.281	0.0053
f2	0.345	0.317	0.328	0.327	0.0076
f3	0.380	0.368	0.374	0.373	0.0039
f4	0.337	0.330	0.334	0.335	0.0021
f5	0.328	0.314	0.317	0.316	0.0029
f6	0.426	0.392	0.398	0.394	0.0083
f7	0.568	0.561	0.564	0.564	0.0020
f8	0.417	0.395	0.399	0.397	0.0048
f9	0.493	0.473	0.475	0.473	0.0044
f10	0.583	0.566	0.571	0.570	0.0045

Tabla 4.9: Estadísticas de tiempo de ejecución por función mediante *genetic algorithm*.