



# INSTITUTO POLITÉCNICO NACIONAL
# ESCUELA SUPERIOR DE CÓMPUTO
# INTELIGENCIA ARTIFICIAL
# ING. EN SISTEMAS COMPUTACIONALES

## Laboratorio 3: Búsqueda no informada parte 2

**Integrantes:**

Hurtado Morales Emiliano - 2021630390

Maestro: Andrés García Floriano

Grupo: 6CV3

Fecha de entrega: 26/09/2024

Ciclo Escolar: 2025 – 1

## Introducción

El uso de estructuras de datos adecuadas es fundamental en la implementación eficiente de algoritmos, especialmente en el ámbito de la Inteligencia Artificial y la resolución de problemas complejos. En esta práctica, se desarrolló una biblioteca en Python que implementa tres estructuras de datos fundamentales: la Pila (Stack), la Cola (Queue), y la Lista. Estas estructuras forman la base de muchas aplicaciones en computación, ya que permiten organizar y gestionar información de manera eficiente para la realización de tareas como la búsqueda, la inserción, y la eliminación de elementos.

Una vez implementadas estas estructuras, se aplicaron en la resolución de dos problemas clásicos utilizando el algoritmo de Búsqueda en Anchura (BFS). El primero de estos problemas es el 4-puzzle, un desafío que consiste en ordenar una secuencia desordenada de números intercambiando elementos adyacentes hasta alcanzar la configuración correcta. El segundo problema es la resolución de un laberinto, donde se busca encontrar el camino más corto desde un punto de inicio hasta una salida, navegando a través de una matriz que representa caminos libres y obstáculos.

La elección de BFS como algoritmo de búsqueda es particularmente adecuada para estos problemas, ya que garantiza encontrar la solución óptima en términos de pasos necesarios para alcanzar el objetivo. Al combinar este algoritmo con las estructuras de datos implementadas, se demuestra la importancia de contar con herramientas eficientes para manejar el conjunto de estados y caminos que deben explorarse durante la ejecución del algoritmo.

## Explicación de la Biblioteca

**Enlace de GitHub:** https://github.com/EmilianoHM/Bender-IA/tree/main/Lab_3

Antes de entrar en los algoritmos específicos, es importante entender la **biblioteca** utilizada. Esta proporciona las estructuras de datos necesarias para implementar DFS y BFS.

### 1. Clase Pila (Stack)

La **pila** es fundamental para la implementación de DFS, ya que nos permite gestionar los caminos que se exploran en el algoritmo. La pila sigue una estructura **LIFO** (Last In, First Out), lo que significa que el último estado insertado es el primero en salir.

- **apilar(objeto)**: Agrega un objeto al final de la pila (lo que representa un nuevo estado que queremos explorar).
- **desapilar()**: Retira y devuelve el último elemento de la pila.
- **esta_vacia()**: Devuelve True si la pila no contiene ningún elemento.
- **cima()**: Devuelve el elemento en la parte superior de la pila sin retirarlo.

### 2. Clase Lista

La clase **Lista** se usa para almacenar los estados que ya han sido visitados, evitando así revisitar los mismos nodos o configuraciones. Esto es crucial para evitar ciclos en los problemas de búsqueda.

- **insertar(objeto)**: Inserta un objeto al final de la lista.
- **buscar(objeto)**: Busca si un objeto ya existe en la lista. Es fundamental para saber si un estado ya ha sido explorado.

### 3. Clase Cola (Queue)

La cola es una estructura de datos de tipo **FIFO** (First In, First Out), lo que significa que el primer elemento en entrar es el primero en salir. Aunque no se usa directamente en DFS, es útil para implementar algoritmos como BFS (Búsqueda en Amplitud).

- **insertar(objeto)**: Agrega un objeto al final de la cola.
- **quitar()**: Retira y devuelve el objeto al frente de la cola.
- **esta_vacia()**: Devuelve True si la cola está vacía.
- **recorrer()**: Recorre e imprime todos los elementos de la cola.

**Explicación de los Algoritmos**

**1. 4-Puzzle en una línea**

El 4-puzzle es un problema donde se busca ordenar una secuencia de números [1, 2, 3, 4] a partir de una configuración inicial dada. El algoritmo BFS explora todas las posibles permutaciones, intercambiando elementos adyacentes hasta encontrar la secuencia correcta.

**Descripción del Código**:

```
def bfs_4_puzzle(start, objetivo):
    # Usamos la Cola implementada
    cola = Cola()
    cola.insertar((start, []))  # (estado actual, lista de movimientos)

    # Usamos la Lista genérica para los visitados
    visitados = Lista()
    visitados.insertar(start)

    while not cola.esta_vacia():
        estado_actual, movimientos = cola.quitar()

        # Si hemos alcanzado el objetivo
        if estado_actual == objetivo:
            return movimientos

        # Generar todos los posibles estados adyacentes (intercambiar elementos
adyacentes)
        for i in range(len(estado_actual) - 1):
            nuevo_estado = estado_actual[:]
            # Intercambiar elementos adyacentes
            nuevo_estado[i], nuevo_estado[i + 1] = nuevo_estado[i + 1],
nuevo_estado[i]

            if not visitados.buscar(nuevo_estado):
                visitados.insertar(nuevo_estado)
                cola.insertar((nuevo_estado, movimientos + [nuevo_estado]))  #
Agregar el nuevo estado y el camino

    return None  # Si no hay solución
```

1. Inicializamos la cola con el estado inicial del puzzle y una lista vacía que representa los movimientos realizados para llegar a ese estado.
2. Creamos la lista de visitados que almacena las configuraciones que ya han sido exploradas para evitar ciclos y redundancias.
3. En el ciclo while, el algoritmo toma un estado de la cola y verifica si es el estado objetivo. Si es así, retorna la secuencia de movimientos.
4. Para generar los estados adyacentes, se intercambian elementos adyacentes en la configuración actual, creando nuevas configuraciones.
5. Si una nueva configuración no ha sido visitada, se agrega a la lista de visitados y se inserta en la cola junto con el camino recorrido hasta ese momento.
6. Este proceso continúa hasta que se encuentra la solución o se terminan las posibles configuraciones.

**2. Laberinto**

El objetivo es encontrar un camino desde la posición inicial hasta la posición final dentro de un laberinto representado como una matriz, donde 0 representa un camino libre y 1 representa un obstáculo. BFS se utiliza para explorar las celdas adyacentes hasta encontrar la salida.

**Descripción del Código**:

```
def bfs_laberinto(maze, start, end):
    filas = len(maze)
    columnas = len(maze[0])

    # Movimientos posibles: arriba, abajo, izquierda, derecha
    movimientos = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # Usamos la Cola implementada
    cola = Cola()
    cola.insertar((start, [start]))  # (posición actual, camino recorrido)

    # Usamos la Lista genérica para los visitados
    visitados = Lista()
    visitados.insertar(start)

    while not cola.esta_vacia():
        (posicion_actual, camino) = cola.quitar()
        fila, columna = posicion_actual

        # Si llegamos al final
        if posicion_actual == end:
            return camino

        # Explorar los vecinos
        for movimiento in movimientos:
```

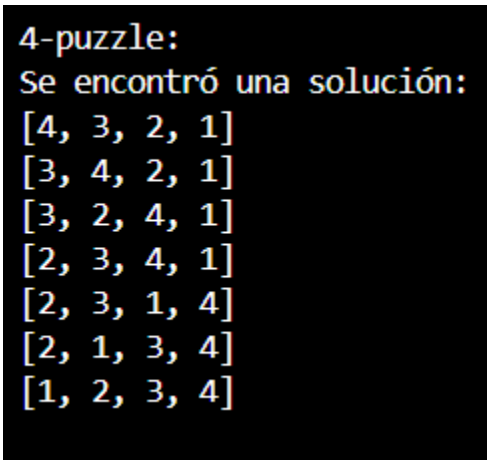
```
            nueva_fila, nueva_columna = fila + movimiento[0], columna +
movimiento[1]

            if 0 <= nueva_fila < filas and 0 <= nueva_columna < columnas and
maze[nueva_fila][nueva_columna] == 0:
                nueva_posicion = (nueva_fila, nueva_columna)
                if not visitados.buscar(nueva_posicion):
                    visitados.insertar(nueva_posicion)
                    cola.insertar((nueva_posicion, camino + [nueva_posicion]))

    return None  # No se encontró un camino
```

1. El laberinto es representado como una matriz donde 0 indica un camino libre y 1 una pared.
2. Inicializamos la cola con la posición de inicio y el camino recorrido hasta ese punto, que inicialmente es solo la posición de inicio.
3. Utilizamos una lista de visitados para registrar las posiciones ya exploradas y así evitar repetir pasos.
4. En el ciclo while, extraemos la posición actual y el camino de la cola. Si llegamos a la posición final (end), el camino recorrido es retornado como la solución.
5. El algoritmo explora las cuatro direcciones posibles (arriba, abajo, izquierda, derecha) a partir de la posición actual.
6. Si una nueva posición es válida (dentro del rango de la matriz y no es una pared), y no ha sido visitada, se agrega a la lista de visitados y se inserta en la cola junto con el camino actualizado.
7. El proceso continúa hasta que se encuentra el camino al final del laberinto o se determinan que no existen más opciones.

## Resultados Esperados

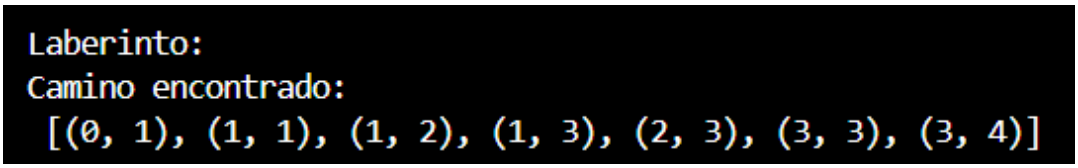
### 1. 4-Puzzle

El algoritmo debería encontrar y mostrar la secuencia de pasos necesarios para transformar la configuración inicial [4, 3, 2, 1] en la configuración final [1, 2, 3, 4].




### 2. Laberinto

El algoritmo debe encontrar un camino desde la posición (0, 1) hasta la posición (3, 4), siguiendo los caminos libres (0) de la matriz.



## Conclusión

La práctica realizada evidenció la importancia de las estructuras de datos y los algoritmos de búsqueda en la resolución de problemas de Inteligencia Artificial. Al implementar las clases de Pila, Cola, y Lista genérica, se obtuvo una comprensión más profunda de cómo estas estructuras permiten organizar y procesar información de manera eficiente. Esto se reflejó claramente en la resolución de los problemas del 4-puzzle y el laberinto, donde el uso de la Cola y la Lista garantizó que el algoritmo BFS pudiera explorar todos los caminos posibles de manera sistemática y sin redundancias.

El uso de BFS demostró ser efectivo para encontrar soluciones óptimas en ambos problemas, cumpliendo su objetivo de explorar los estados de manera completa y en orden de menor a mayor costo. Esta propiedad es crucial en la Inteligencia Artificial, ya que permite encontrar soluciones eficientes y precisas a problemas que involucran múltiples decisiones o rutas posibles.

En conclusión, esta práctica reforzó la importancia de seleccionar y aplicar las estructuras de datos adecuadas para el desarrollo de algoritmos eficientes. Además, mostró cómo estas herramientas pueden ser aplicadas a problemas reales y complejos, permitiendo a los sistemas de inteligencia resolverlos de manera óptima y eficiente. La combinación de una buena selección de estructuras de datos con la implementación de algoritmos robustos, como BFS, es fundamental para abordar los desafíos que surgen en el campo de la computación y la Inteligencia Artificial.