Imagen que contiene Icono

Descripción generada automáticamenteDibujo en blanco y negro

Descripción generada automáticamente con confianza bajaINSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

INTELIGENCIA ARTIFICIAL

**PRÁCTICA 4**

**Búsqueda informada**

NOMBRE DEL ALUMNO: GARCÍA QUIROZ GUSTAVO IVAN

NOMBRE DEL PROFESOR: GARCÍA FLORIANO ANDRES

FECHA DE ENTREGA DEL REPORTE: 04/10/2024

Índice

[Introducción 1](#_Toc178201068)

[Material y equipo. 1](#_Toc178201069)

[Hardware 1](#_Toc178201070)

[Software 1](#_Toc178201071)

[Desarrollo de la práctica. 2](#_Toc178201072)

[Actividades desarrolladas 2](#_Toc178201073)

[Estructuras de datos 2](#_Toc178201074)

[Pila 2](#_Toc178201075)

[Cola 3](#_Toc178201076)

[Lista Genérica 4](#_Toc178201077)

[Problema del 4-puzzle 5](#_Toc178201078)

[Descripción del Problema 5](#_Toc178201079)

[Implementación del Algoritmo 5](#_Toc178201080)

[Problema del laberinto 7](#_Toc178201081)

[Descripción del Problema 7](#_Toc178201082)

[Implementación del Algoritmo 7](#_Toc178201083)

[Resultados obtenidos 9](#_Toc178201084)

[Problema del 4-puzzle 9](#_Toc178201085)

[Problema del laberinto 10](#_Toc178201086)

[Conclusiones. 11](#_Toc178201087)

[Referencias. 12](#_Toc178201088)

[Código 12](#_Toc178201089)

# Introducción

La práctica se presenta la aplicación de técnicas de búsqueda no informada para resolver problemas. Utilizando Python como lenguaje de programación principal, se aprovechan sus librerías para implementar soluciones. La elección de Python se basa en su capacidad para simplificar el código y proporcionar funciones predefinidas que facilitan el desarrollo de algoritmos.

**Instrucciones**

1. Mediante el algoritmo A\* resuelve el problema de búsqueda en el siguiente laberinto.

# Representación del laberinto

# 0 = camino, 1 = pared

maze = [

    [1, 0, 1, 1, 1],

    [1, 0, 0, 0, 1],

    [1, 1, 1, 0, 1],

    [1, 0, 0, 0, 0],

    [1, 1, 1, 1, 1]

]

# Posición de inicio y salida

start = (0, 1)  # Coordenadas (fila, columna) de inicio

end = (3, 4)    # Coordenadas (fila, columna) de salida

Mide el tiempo de ejecución.

1. Propón un escenario adicional, en dónde el laberinto sea más complejo y de mayor tamaño.

Mide el tiempo de ejecución.

# Material y equipo.

Para esta práctica se usaron las siguientes herramientas de software y hardware necesarias para realizar la práctica.

### Hardware

* Computadora.

### Software

* Visual Studio Code.
* Phyton 3.

# Desarrollo de la práctica.

En esta práctica, estudiamos la implementación de la búsqueda informada utilizando el algoritmo A\* para resolver problemas de laberinto. El lenguaje de programación elegido fue Python porque puede simplificar el código y utilizar funciones de manera más rápida.

## Actividades desarrolladas

### Estructuras de datos

La primera fase de la práctica consistió en la creación de una biblioteca de estructuras de datos fundamentales. Se desarrolló un archivo llamado "estructuras\_datos.py" que contiene las implementaciones de Pila, Cola y Lista genérica. Estas estructuras se diseñaron para almacenar objetos, no solo datos simples.

### Pila

La Pila se implementó siguiendo el principio de "último en entrar, primero en salir" (LIFO). Se creó una clase Pila que utiliza una estructura de nodos enlazados para almacenar los elementos. La clase incluye métodos para las operaciones estándar: push para añadir un elemento al tope de la pila, pop para remover y devolver el elemento del tope, y peek para observar el elemento del tope sin removerlo. Además, se implementó un método esta\_vacia para verificar si la pila no contiene elementos. Esta implementación permite una gestión eficiente de la memoria, ya que los nodos se crean y destruyen dinámicamente según se necesitan.

Texto

Descripción generada automáticamente

Figura 1 Implementación de Pila.

### Cola

La Cola se diseñó siguiendo el principio de "primero en entrar, primero en salir" (FIFO). Al igual que la Pila, se utilizó una estructura de nodos enlazados, pero en este caso se mantienen referencias tanto al frente como al final de la cola. La clase Cola incluye métodos para insertar elementos al final de la cola, quitar elementos del frente, recorrer todos los elementos de la cola, y buscar un elemento específico. La implementación de dos punteros (frente y final) permite realizar operaciones de inserción y eliminación en tiempo constante O(1), lo que hace que esta estructura sea eficiente para escenarios donde se requiere un procesamiento en orden de llegada.

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamente

Figura 2 Implementación de Cola.

### Lista Genérica

La Lista Genérica se implementó como una estructura de datos flexible que permite la inserción de elementos en cualquier posición y la búsqueda eficiente de elementos. A diferencia de la Pila y la Cola, que tienen restricciones en cuanto a dónde se pueden insertar o eliminar elementos, la Lista Genérica ofrece mayor flexibilidad. Se implementaron métodos para insertar elementos (en este caso, al principio de la lista por simplicidad) y para buscar elementos en la lista..

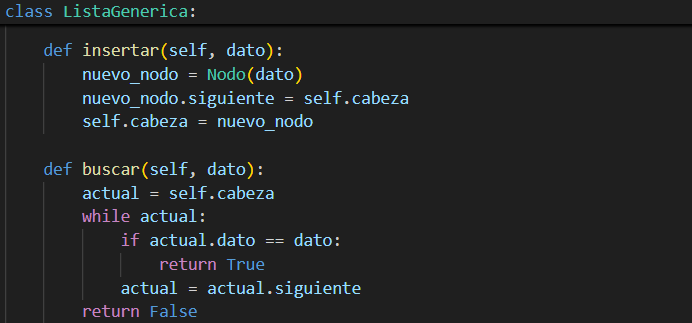


Figura 3 Implementación de Lista.

Para la Pila, se implementaron las operaciones estándar como push, pop y peek. La Cola, por su parte, incluyó funciones para insertar, quitar, recorrer y buscar elementos. La Lista genérica se dotó de capacidades para insertar y buscar objetos. Todas estas estructuras se basaron en una clase Nodo común, que permite el almacenamiento de cualquier tipo de dato, facilitando así el trabajo con objetos complejos.

Forma

Descripción generada automáticamente con confianza baja

Figura 4 Implementación de Nodo

## Problema del laberinto usando el algoritmo A\*

### Descripción del Problema

El 4-puzzle es una versión simplificada del clásico juego de deslizamiento 15-puzzle. Consiste en un tablero de 2x2 casillas, donde hay tres fichas numeradas (1, 2 y 3) y un espacio vacío. El objetivo del juego es deslizar las fichas en el tablero, utilizando el espacio vacío, para alcanzar una configuración específica desde un estado inicial dado. Los movimientos permitidos son deslizar una ficha adyacente al espacio vacío hacia dicho espacio, lo que resulta en cuatro posibles movimientos: arriba, abajo, izquierda o derecha, dependiendo de la posición del espacio vacío.

### Implementación del Algoritmo

La implementación del algoritmo A\* se realizó en Python, utilizando las mismas estructuras y mecanismos para ambos laberintos. El algoritmo se estructuró de la siguiente manera:

### Heurística

Se definió una clase Nodo para representar cada celda del laberinto. Cada Nodo contiene su posición (x, y), los valores g (costo acumulado desde el inicio), h (heurística estimada hasta el objetivo), f (suma de g y h), y una referencia al nodo padre.

La función heurística implementada fue la distancia Manhattan, calculada como |x2 - x1| + |y2 - y1|, donde (x1, y1) es la posición actual y (x2, y2) es la posición objetivo. Esta heurística se eligió por su simplicidad y eficacia en laberintos con movimientos.



Figura 5 Distancia Manhattan.

El algoritmo principal utiliza dos estructuras de datos: una cola de prioridad (implementada con el módulo heapq de Python) para la lista abierta, y un conjunto para la lista cerrada. La cola de prioridad ordena los nodos según su valor f, permitiendo seleccionar eficientemente el nodo con menor f en cada iteración.

El proceso de búsqueda comienza inicializando el nodo de inicio con g=0 y h calculada hasta el objetivo. Este nodo se agrega a la cola de prioridad. El bucle principal del algoritmo continúa mientras haya nodos en la cola de prioridad:

1. Se extrae el nodo con menor f de la cola de prioridad.
2. Si este nodo es el objetivo, se reconstruye y devuelve el camino.
3. Si no, se marca como visitado (se agrega a la lista cerrada) y se exploran sus vecinos.
4. Para cada vecino no visitado: a. Se calcula su nuevo costo g. b. Si el vecino no está en la lista abierta, se agrega. c. Si está en la lista abierta pero el nuevo camino es mejor (menor g), se actualiza. d. Se calcula f = g + h y se agrega o actualiza en la cola de prioridad.

La función para obtener vecinos verifica las cuatro direcciones posibles (arriba, abajo, izquierda, derecha), asegurándose de que las nuevas posiciones estén dentro de los límites del laberinto y no sean paredes.

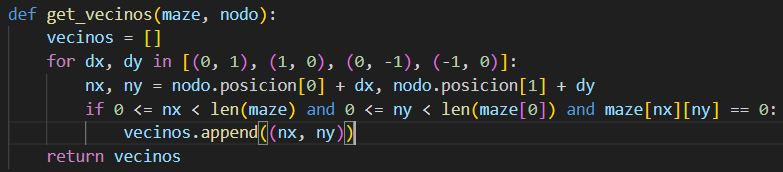


Figura 6 Función para obtener vecinos.

## Problema del laberinto

### Descripción del Problema

El problema del laberinto implica encontrar un camino desde un punto de entrada hasta un punto de salida en una estructura bidimensional que contiene obstáculos. En este caso específico, el laberinto se representa como una matriz de 5x5, donde el valor 0 indica un camino libre y el valor 1 representa una pared o obstáculo. El punto de inicio se define en las coordenadas (0, 1) y el punto de salida en (3, 4). El objetivo es desarrollar un algoritmo que pueda navegar a través del laberinto, evitando las paredes y encontrando una ruta válida desde el inicio hasta la salida. El algoritmo debe ser capaz de explorar diferentes caminos, retroceder cuando se encuentra en un callejón sin salida y, finalmente, determinar si existe una solución y, de ser así, proporcionar la secuencia de movimientos que conducen a la salida.

### Implementación del Algoritmo

El algoritmo principal para resolver el laberinto se implementa en la función resolver\_laberinto, utilizando una Búsqueda en Anchura (BFS). Esta función toma como entrada la matriz del laberinto, las coordenadas de inicio y las coordenadas de salida. Se utiliza una cola (implementada en la biblioteca de estructuras de datos creada previamente) para mantener las posiciones por explorar, siguiendo el principio FIFO (First In, First Out) característico del BFS. El algoritmo comienza insertando la posición inicial en la cola, junto con una lista que contiene solo esta posición como camino inicial.

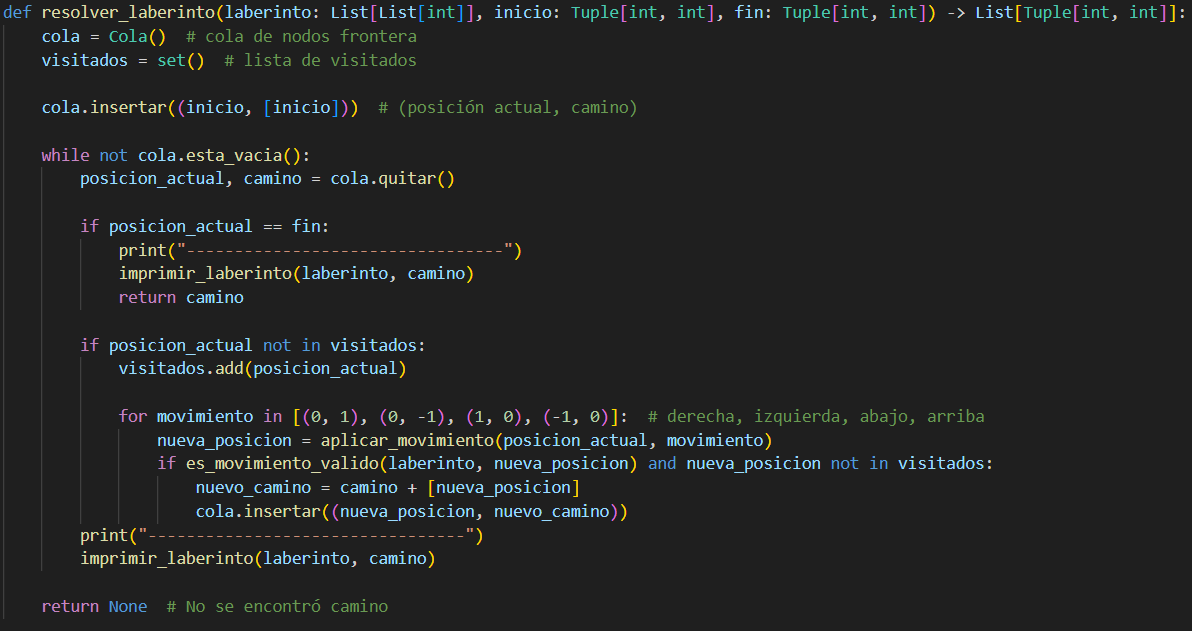


Figura 9 Función resolver\_laberinto().

En cada iteración, se extrae una posición y su camino correspondiente de la cola. Si esta posición es la salida, se ha encontrado una solución y se retorna el camino. En caso contrario, se exploran las cuatro direcciones posibles (arriba, abajo, izquierda, derecha) desde la posición actual. Para cada nueva posición válida (dentro de los límites del laberinto y que sea un camino transitable), se crea un nuevo camino añadiendo esta posición al camino actual, y se inserta en la cola. Para evitar ciclos y posiciones repetidas, se mantiene un conjunto de posiciones visitadas. El proceso continúa hasta encontrar la salida o hasta que se agoten todas las posiciones posibles. Si se encuentra una solución, el algoritmo devuelve la secuencia de posiciones que lleva desde el inicio hasta la salida. En caso contrario, retorna None, indicando que no se encontró solución. Esta implementación garantiza encontrar el camino más corto si existe, debido a la naturaleza de la búsqueda en anchura.

En la función principal se declaran los siguientes datos de entrada: el arreglo bidimensional del laberinto, el inicio y el fin. Después de declarar variables, la solución va almacenar el resultado de la función resolver\_laberinto y se va a encontrar la respuesta.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Figura 10 Programa principal.

## Resultados obtenidos

## Problema del 4-puzzle

Texto

Descripción generada automáticamente

Figura 11 Ejemplo 1 del problema del 4-puzzle.

Texto

Descripción generada automáticamente

Figura 12 Ejemplo 2 del problema del 4-puzzle.

## Problema del laberinto

Calendario

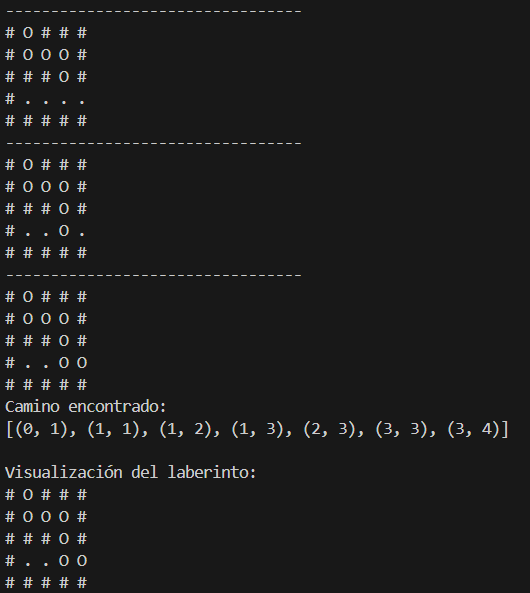
Descripción generada automáticamente 

Figura 13 Problema del laberinto.

# Conclusiones.

Esta práctica permitió comprender la implementación y aplicación de estructuras de datos como pilas, colas y listas genéricas en Python, así como su utilización en algoritmos de búsqueda no informada, específicamente BFS. Se practicó en la resolución de problemas de búsqueda en espacios de estados, como el 4-puzzle y el laberinto, aplicando conceptos teóricos. Para un mejor aprovechamiento de la práctica, ha sido beneficioso repasar previamente en clase los conceptos de las propiedades específicas de BFS en comparación con otras estrategias de búsqueda, y técnicas avanzadas de optimización de código en Python, como el uso de estructuras de datos eficientes para conjuntos y colas de prioridad.

# Referencias.

[1] “Uninformed search algorithms in AI”, GeeksforGeeks, 20-jun-2024. [En línea]. Disponible en: <https://www.geeksforgeeks.org/uniformed-search-algorithms-in-ai/>. [Consultado: 17-sep-2024].

[2] G. Improve, “Depth first search or DFS for a graph”, GeeksforGeeks, 15-mar-2012. [En línea]. Disponible en: <https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>. [Consultado: 17-sep-2024].

[3] 2\_uninformed\_search at main · GUSTAVOIVANGQ/AI. .

# Código

[3] <https://github.com/GUSTAVOIVANGQ/AI/tree/main/3_Uninformed_Search>