INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

INGENIERÍA EN SISTEMAS COMPUTACIONALES

Inteligencia Artificial

Laboratorio 3: Búsqueda No Informada pt.2

Sebastián Reyes Núñez

Jesús Pérez González

Marco Isaí Vicencio Flores

Grupo:

6CV3

Introducción

En este laboratorio se lleva a cabo el desarrollo y la implementación de diversas estructuras de datos como listas genéricas, colas y pilas utilizando una biblioteca de objetos creada específicamente para esta práctica. El objetivo principal de esta práctica es comprender y aplicar los conceptos fundamentales de estas estructuras de datos, así como su utilidad en la resolución de problemas computacionales complejos. Para ilustrar el uso de estas estructuras, se implementaron dos algoritmos: uno para la resolución de un problema de puzzle utilizando búsqueda en anchura, y otro para resolver un laberinto con el mismo enfoque.

Marco Teórico

Algoritmo de Búsqueda en Anchura (BFS)

El algoritmo de Búsqueda en Anchura es un método de búsqueda o recorrido de grafos que explora un grafo o árbol de manera sistemática, visitando todos los nodos a un mismo nivel de profundidad antes de pasar al siguiente nivel. Se caracteriza por su enfoque exhaustivo, lo que lo convierte en una técnica adecuada para encontrar el camino más corto en grafos no ponderados, como laberintos o problemas de búsqueda de rutas.

El BFS utiliza una cola para gestionar los nodos pendientes por visitar. En cada iteración, el nodo al frente de la cola es procesado y se añaden sus vecinos no visitados al final de la misma. Este proceso garantiza que los nodos se visiten en orden de cercanía, lo que es especialmente útil en problemas donde el objetivo es encontrar la solución más cercana o más corta.

Una característica importante del BFS es que asegura encontrar la solución óptima en términos de número de pasos, siempre y cuando todos los movimientos tengan el mismo "costo". Por esta razón, es muy adecuado para aplicaciones donde la prioridad es la eficiencia y la minimización del número de movimientos o pasos necesarios para alcanzar el estado deseado.

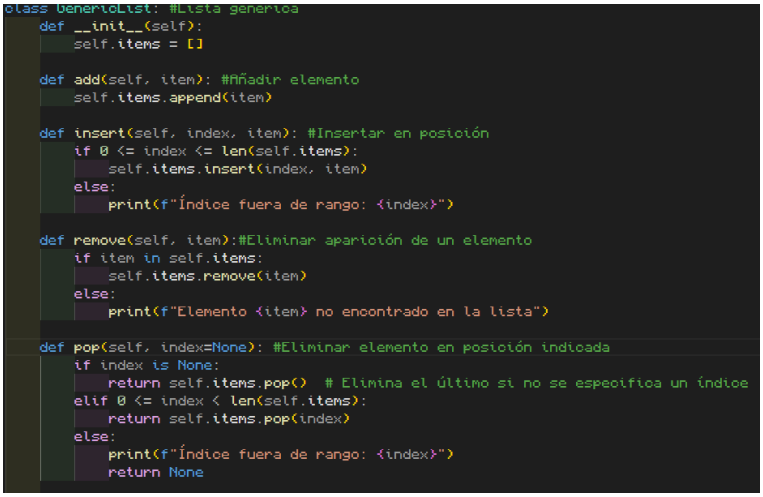
Ventajas y Limitaciones del BFS

Entre las principales ventajas del algoritmo BFS se destaca su capacidad para encontrar el camino más corto en problemas donde todas las acciones tienen el mismo costo. Además, su implementación es relativamente simple y fácil de entender, lo que lo convierte en una opción preferida en problemas de búsqueda donde la solución puede estar cerca del punto de partida.

Sin embargo, el BFS también tiene algunas limitaciones. Dado que explora todos los nodos a un mismo nivel antes de profundizar, el consumo de memoria puede aumentar significativamente, especialmente en grafos o árboles con una gran cantidad de nodos. Esto hace que el BFS no sea adecuado para grafos muy profundos o infinitos, ya que podría requerir un almacenamiento desmesurado de nodos. A pesar de esta limitación, sigue siendo una herramienta eficiente y eficaz en problemas donde la búsqueda de la solución óptima en un espacio acotado es crucial.

Desarrollo

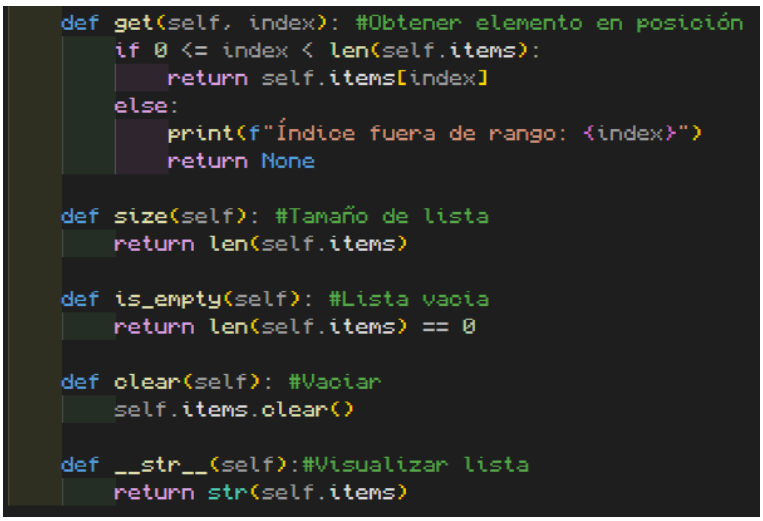
Para el desarrollo de esta practica se utilizo una biblioteca de objetos creada por nosotros, donde se creo una instancia para una lista genérica, una cola FIFO y una fila LIFO, cada una con sus respectivos métodos:



```
class GenericList: #Lista generica
    def __init__(self):
        self.items = []

    def add(self, item): #Añadir elemento
        self.items.append(item)

    def insert(self, index, item): #Insertar en posición
        if 0 <= index <= len(self.items):
            self.items.insert(index, item)
        else:
            print(f"Índice fuera de rango: {index}")

    def remove(self, item):#Eliminar aparición de un elemento
        if item in self.items:
            self.items.remove(item)
        else:
            print(f"Elemento {item} no encontrado en la lista")

    def pop(self, index=None): #Eliminar elemento en posición indicada
        if index is None:
            return self.items.pop()  # Elimina el último si no se especifica un índice
        elif 0 <= index < len(self.items):
            return self.items.pop(index)
        else:
            print(f"Índice fuera de rango: {index}")
            return None
```

Para la lista genérica se realizaron los siguientes métodos:

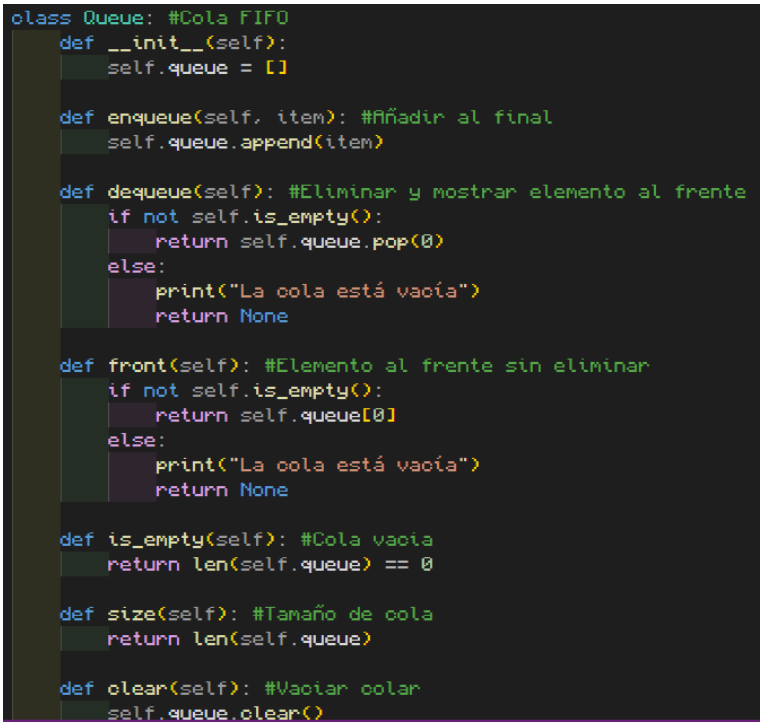
- *add:* Añadir elementos en la lista.
- *insert:* Insertar un elemento en una posición deseada. Si la posición se encuentra fuera del rango de la lista entonces se notifica al usuario.
- *remove:* Elimina la primera aparición de un elemento en la lista. Si el elemento no se encuentra en la lista entonces se notifica al usuario.
- *pop:* Elimina un elemento en una posición deseada, en caso de ingresar un índice de posición se elimina la ultima aparición del elemento. Si la posición se encuentra fuera de rango entonces se notifica al usuario.



```
    def get(self, index): #Obtener elemento en posición
        if 0 <= index < len(self.items):
            return self.items[index]
        else:
            print(f"Índice fuera de rango: {index}")
            return None

    def size(self): #Tamaño de lista
        return len(self.items)

    def is_empty(self): #Lista vacia
        return len(self.items) == 0

    def clear(self): #Vaciar
        self.items.clear()

    def __str__(self):#Visualizar lista
        return str(self.items)
```

- *get:* Obtenemos un elemento dada una posición. Si la posición se encuentra fuera de rango entonces se notifica al usuario.

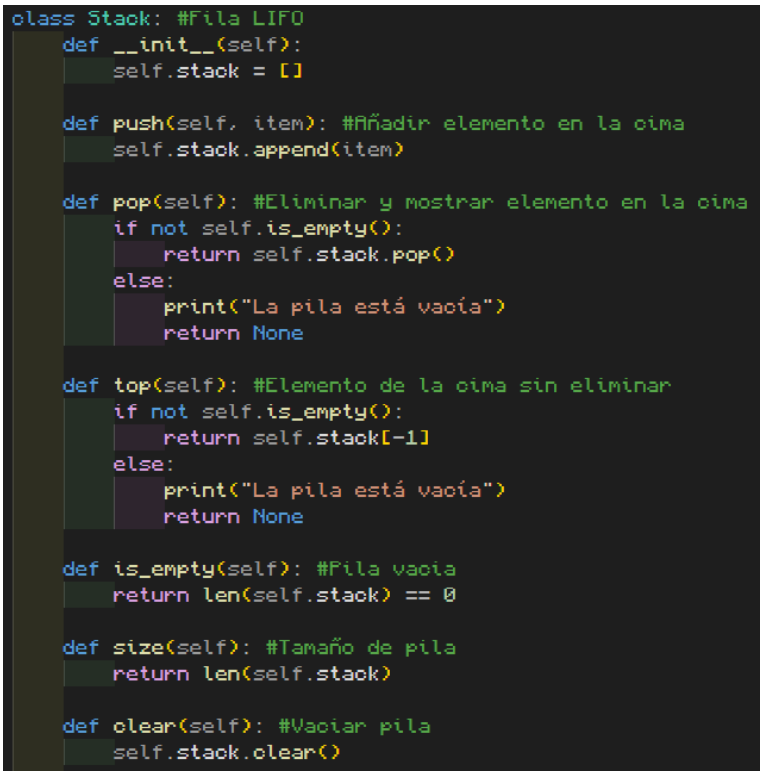
- *size:* Permite conocer el tamaño total de la lista.
- *is_empty:* Permite saber si la lista se encuentra vacía.
- *Clear:* Elimina todos los elementos de la lista.
- *__str__:* Muestra una representación del estado actual de la lista.



```
class Queue: #Cola FIFO
    def __init__(self):
        self.queue = []

    def enqueue(self, item): #Añadir al final
        self.queue.append(item)

    def dequeue(self): #Eliminar y mostrar elemento al frente
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            print("La cola está vacía")
            return None

    def front(self): #Elemento al frente sin eliminar
        if not self.is_empty():
            return self.queue[0]
        else:
            print("La cola está vacía")
            return None

    def is_empty(self): #Cola vacia
        return len(self.queue) == 0

    def size(self): #Tamaño de cola
        return len(self.queue)

    def clear(self): #Vaciar colar
        self.queue.clear()
```

Para las colas de tipo FIFO tenemos los siguientes métodos:

- *enqueue:* Agrega un elemento a la cola, es decir, en la posición final.
- *dequeue:* Elimina el elemento que se encuentra al frente de la cola.
- *front:* Muestra el elemento al frente de la cola sin eliminarlo.
- *is_empty:* Permite conocer si la cola se encuentra vacía.
- *size:* Muestra el tamaño de la cola.
- *clear:* Elimina todos los elementos de la cola para dejarla vacía.
- *__str__:* Muestra el estado actual de la cola.



```python
class Stack: #Fila LIFO
    def __init__(self):
        self.stack = []

    def push(self, item): #Añadir elemento en la cima
        self.stack.append(item)

    def pop(self): #Eliminar y mostrar elemento en la cima
        if not self.is_empty():
            return self.stack.pop()
        else:
            print("La pila está vacía")
            return None

    def top(self): #Elemento de la cima sin eliminar
        if not self.is_empty():
            return self.stack[-1]
        else:
            print("La pila está vacía")
            return None

    def is_empty(self): #Pila vacia
        return len(self.stack) == 0

    def size(self): #Tamaño de pila
        return len(self.stack)

    def clear(self): #Vaciar pila
        self.stack.clear()
```

Para la pila LIFO contamos con los siguientes métodos:

- *push:* Añade un elemento en la cima de la pila.
- *pop:* Elimina y muestra el elemento que se encuentra en la cima de la pila.
- *top:* Muestra el elemento en la cima de la pila, pero sin eliminarlo.
- *is_empty:* Permite conocer si la pila se encuentra vacía.
- *size:* Muestra el tamaño de la pila.
- *clear:* Elimina todos los elementos de la pila para dejarla vacía.
- *__str__:* Muestra el estado actual de la pila.

Puzzle-4

Para resolver el problema de puzzle-4 se utilizó el algoritmo de búsqueda en anchura (BFS por sus siglas en ingles *"Breadht First Search")*, utilizando una cola FIFO de nuestra librería:

```python
from collections import deque
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            return None

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)

    def clear(self):
        self.queue.clear()

    def __str__(self):
        return str(self.queue)
```

Utilizamos el módulo *collections*, que permite implementar colas doblemente enlazadas, que facilitan añadir y eliminar elementos al final de la cola.

Utilizamos los métodos *enqueue, dequeue, is_empty, size, clear* y *__str__* de nuestra cola para agregar y eliminar elementos de nuestra cola. Además de conocer el tamaño de la cola, saber si esta vacía y mostrar el estado actual.

```python
def vecino(estado):
    vecinos = []
    zero_pos = estado.index(0)  # Encuentra la posición del espacio vacío (0)
    movimientos_pos = {
        0: [1, 2],
        1: [0, 3],
        2: [0, 3],
        3: [1, 2],
    }

    for m in movimientos_pos[zero_pos]:
        nuevo = estado[:]
        # Intercambia el espacio vacío (0) con la ficha de la posición adyacente
        nuevo[zero_pos], nuevo[m] = nuevo[m], nuevo[zero_pos]
        vecinos.append(nuevo)

    return vecinos
```

Definimos el método *vecinos*, que constara de una lista genérica donde buscaremos la posición del espacio vacío, representado por el 0, en el tablero y todos los movimientos posibles en sus casillas adyacentes.

Mediante un ciclo *for*, calculamos e intercambiamos la posición del espacio vacío entre sus casillas adyacentes.

```
def bfs_4_puzzle(estado_ini, estado_obj):
    queue = Queue()
    queue.enqueue((estado_ini, []))  # (estado actual, ruta de movimientos)
    visited = set()

    while not queue.is_empty():
        estado_actual, camino = queue.dequeue()

        if estado_actual == estado_obj:
            return camino + [estado_actual]  # Devolvemos la ruta completa

        visited.add(tuple(estado_actual))  # Marcamos el estado como visitado

        # Generar estados vecinos
        for v in vecino(estado_actual):
            if tuple(v) not in visited:
                queue.enqueue((v, camino + [estado_actual]))

    return None  # No se encontró solución
```

Definimos el método *bfs_4_puzzle* que utilizara el algoritmo BFS para resolver el laberinto, tomando como parámetros el estado inicial y estado objetivo.

Utilizamos una cola, para almacenar el estado inicial y la ruta de movimiento, y una lista genérica que servirá para guardar los nodos visitados.

Mientras la cola no este vacía, tomara el estado actual y lo comparara con el estado objetivo. En caso de ser satisfactoria se devuelve la ruta tomada, en otro caso, se almacena la tupla que conforma el estado actual en la lista de nodos visitados. Los estados adyacentes se generan si la tupla actual no se encuentra dentro de los nodos visitados.

```
estado_ini = [1, 2, 0, 3]
estado_obj = [1, 2, 3, 0]

# Resolver el puzzle
res = bfs_4_puzzle(estado_ini, estado_obj)

if res:
    print("Camino de solución encontrado:")
    for step in res:
        print(step)
else:
    print("No se encontró solución para el puzzle.")
```

Finalmente, establecemos el estado inicial y estado objetivo. Llamamos al método *bfs_4_puzzle* con los parámetros establecidos anteriormente. Se muestra el estado de solución en caso de encontrarse.

Resultado:

```
[Running] python -u "c:\Users\marco\Downloads\Practica 3\4Puzzle.py"
Camino de solución encontrado:
[1, 2, 0, 3]
[1, 2, 3, 0]
```

Se muestra cada instancia que se realiza con el algoritmo hasta alcanzar el estado objetivo.

Laberinto

Para resolver el laberinto se utilizó un algoritmo de búsqueda en anchura (BFS por sus siglas en ingles "*Breadth First Search")*, utilizando una cola FIFO de nuestra librería:

```python
from collections import deque

class Queue: #Cola
    def __init__(self):
        self.queue = deque()

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.popleft()
        return None

    def is_empty(self):
        return len(self.queue) == 0
```

Utilizamos el módulo *collections*, que permite implementar colas doblemente enlazadas, que facilitan añadir y eliminar elementos al final de la cola.

Utilizamos los métodos *enqueue*, *dequeue* y *is_empty* para añadir y eliminar elementos en la cola, así como saber si la pila se encuentra vacía.

```python
def bfs_maze(maze, start, end): #Algoritmo BFS para resolver el laberinto
    queue = Queue() #Declaración de cola y lista generica.
    visited = set()
    queue.enqueue((start, [start]))  #(posición actual, ruta recorrida)
```

Declaramos el constructor del BFS para el laberinto, donde se reciben como parámetros el objeto que representa el laberinto, las coordenadas de entrad y las de salida, además de llamar a la cola y una lista genérica para los nodos visitados. Además, en la cola agregamos la posición de inicio y una lista que nos permite conocer la ruta recorrida.

```python
    while not queue.is_empty(): #
        (position, path) = queue.dequeue()
        x, y = position

        # Si llegamos a la salida, devolvemos la ruta seguida
        if position == end:
            return path

        # Si ya visitamos esta posición, la ignoramos
        if position in visited:
            continue

        visited.add(position)
```

Se realiza un bucle *while* mientras la cola no este vacía se van agregando los siguientes nodos según su posición y el camino recorrido. En cada iteración se revisa si la posición actual corresponde a la posición de salida, en otro caso, la posición se agrega a la lista de nodos visitados. En caso de que la posición ya se encuentra en la lista, entonces se ignora y continua la iteración.

```
        # Movimientos posibles
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for move in moves:
            new_x, new_y = x + move[0], y + move[1]

            # Verificar si no choca con pared y no se sale del laberinto
            if 0 <= new_x < len(maze) and 0 <= new_y < len(maze[0]) and maze[new_x][new_y] == 0:
                queue.enqueue(((new_x, new_y), path + [(new_x, new_y)]))

    return None  # Si no se encuentra solución
```

Se definen todos los movimientos posibles, representados en tuplas (Arriba, abajo, izquierda, derecha) y se itera sobre la lista para obtener las nuevas al realizar el movimiento, realizando una suma de posiciones.

Se verifica que la nueva posición calculada se encuentre dentro de los limites del tablero y que la siguiente posición no sea una pared. En caso de cumplir las condiciones, se añade a la cola posición actual, así como el camino recorrido.

```
# Representación del laberinto
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]
]

start = (0, 1)
end = (3, 4)

solution_path = bfs_maze(maze, start, end)

if solution_path:
    print("Camino encontrado:", solution_path)
else:
    print("No hay solución para el laberinto.")
```

Finalmente, realizamos la representación del laberinto, así como las coordenadas de entrada y de salida y se llama al método *bfs_maze* para resolver el laberinto usando un algoritmo BFS. Se muestra si se encuentra el camino o no existe solución.

Resultado:

```
Camino encontrado: [(0, 1), (1, 1), (1, 2), (1, 3), (2, 3), (3, 3), (3, 4)]
PS C:\Users\marco\Downloads\Practica 3>
```

Este es el resultado obtenido, donde se muestra la lista que contiene las tuplas que representan las posiciones recorridas hasta encontrar la solución.

Conclusiones

En este laboratorio se implementaron y analizaron diversas estructuras de datos, como listas genéricas, colas FIFO y pilas LIFO, que son fundamentales para resolver problemas

computacionales de manera eficiente. Utilizando estas estructuras, se aplicaron algoritmos de búsqueda no informada, específicamente el algoritmo de Búsqueda en Anchura (BFS), para resolver dos problemas: un puzzle de desplazamiento y un laberinto.

El uso del BFS demostró ser una solución efectiva para encontrar caminos óptimos en estos tipos de problemas. La implementación de una cola FIFO como estructura principal del algoritmo permitió una exploración sistemática y exhaustiva de los nodos, asegurando que se encontrara la solución más corta en términos de número de pasos. Sin embargo, se evidenció también la limitación inherente del BFS en cuanto al consumo de memoria, ya que, al explorar todos los nodos de un mismo nivel antes de avanzar, el tamaño de la cola puede crecer rápidamente en problemas más complejos o con grandes espacios de búsqueda.

En conclusión, este laboratorio permitió no solo reforzar la comprensión de las estructuras de datos básicas y su implementación, sino también visualizar su aplicación práctica en la resolución de problemas de búsqueda, destacando tanto sus ventajas como limitaciones en escenarios reales.

Referencias

Breadth First Search Tutorials & Notes | Algorithms | HackerEarth. (2016, 27 abril). HackerEarth. https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

GeeksforGeeks. (2024, 24 septiembre). *Breadth First Search or BFS for a Graph*. GeeksforGeeks. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

takeuforward - Best Coding Tutorials for Free. (s. f.). takeUforward - ~ Strive For Excellence. https://takeuforward.org/graph/breadth-first-search-bfs-level-order-traversal/