Instituto Politecnico Nacional

Escuela Superior de Computo

Ingeniería en Sistemas Computacionales

Sexto semestre

Grupo: 6CV3

Inteligencia Artificial

Practica 2: Búsqueda no informada DFS

Profesor: García Floriano Andrés

Presentado por:

Reyes Núñez Sebastián

Flores Vicencio Marco Isaí

Pérez Gonzales Jesús

Introducción

Las listas, colas y pilas son estructuras de datos lineales que poseen restricciones para agregar y eliminar elementos de estas. En el caso de las colas y pilas, se consideran como casos especiales de listas, que tienen la particularidad de formarse mediante vectores o listas enlazadas.

Las pilas (stacks) son variaciones de listas donde agregar o eliminar elementos se realiza por un extremo conocido como cima (top). También conocidas como listas LIFO por el principio "Last In – First Out", que expresa que el ultimo elemento que haya sido agregado a la pila será el primero en ser extraído.

Por otro lado, una cola (Queue) es una variación de lista donde agregar o eliminar elementos se realizan en dos extremos diferentes de la lista. Añadir un elemento a la lista se realiza en el extremo final (rear), mientras que eliminar un elemento se realiza en el extremo contrario conocido como frente (front). Del mismo modo, son conocidas como listas FIFO debido a su principio de "First In – First Out" que, como su nombre indica, expresa que el primer elemento en ser agregado a la lista será el primer elemento en ser extraído.

El algoritmo de búsqueda en profundidad (Depth First Search por sus siglas en ingles) es un algoritmo utilizado para recorrer los nodos de un grado de manera secuencial y ordenada. Consiste en ir expandiendo los nodos localizados de forma recurrente, desde el nodo padre hasta los nodos hijos. Una vez visitados cada no de los nodos del camino, regresa al nodo predecesor. Los algoritmos de búsqueda de este tipo suelen ser utilizados para resolver puzles que cuentan con una única solución, como los ya vistos 8 – Puzzle y los laberintos.

Desarrollo

Para el desarrollo de esta práctica, se llevaron a cabo tres ejercicios:

1er ejercicio: El primer ejercicio consistió en elaborar una biblioteca que contenga las estructuras de datos principales, capaz de almacenar datos primitivos y objetos.

```python
class Pila:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.empty():
            return self.items.pop()
        return None

    def empty(self):
        return len(self.items) == 0

    def top(self):
        if not self.empty():
            return self.items[-1]
        return None

    def size(self):
        return len(self.items)
```

Se define una clase *Pila*, donde se declaran las operaciones de *push* (agregar), *pop* (sacar), *empty* (verificar si esta vacia), *top* (retornar la cima) y *size* (verificar el tamaño).

```python
class Cola:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.empty():
            return self.items.pop(0)
        return None

    def empty(self):
        return len(self.items) == 0

    def travel(self):
        return self.items

    def search(self, item):
        return item in self.items
```

Se define una clase *Cola*, que permite realizar las operaciones de *push (*agregar), *pop (*sacar), *empty*(verificar si esta vacia), *travel* (recorre la cola) y *search (*para buscar un elemento).

```python
class Lista:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def search(self, item):
        return item in self.items

    def travel(self):
        return self.items

```

De declara una clase *Lista*, que permite las operaciones de *push* (agregar), *search* (búsqueda) y travel (verificar elementos)

2do ejercicio: Se utiliza la biblioteca desarrollada para emplear el algoritmo DFS con el objetivo de resolver el problema de *4-puzzle*.

```python
class Puzzle4:
    def __init__(self, initial_state, final_state):
        self.initial_state = initial_state
        self.final_state = final_state
        self.moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # derecha, abajo, izquierda, arriba
        self.name_moves = ["Derecha", "Abajo", "Izquierda", "Arriba"]  # Para visualizar los movimientos
```

Se declara la clase *Puzzle4* con un constructor que recibe el estado inicial y estado final que se guardan en variables de instancia. Además se definen los posibles movimientos que se pueden realizar mediante coordenadas con sus respectivos nombres.

```python
def find_void(self, estado): #Encontrar el espacio vacio (0)
    for i in range(2):
        for j in range(2):
            if estado[i][j] == 0:
                return (i, j)
    return None
```

El método *find_void* tiene como objetivo encontrar el espacio de la matriz que está vacío. Esto lo realiza realizando ciclos *for* a través de la matriz hasta en encontrar el valor 0.

```python
def move(self, estado, pos_void, direction): #Mover el espacio vacio
    x, y = pos_void
    dx, dy = direction
    new_x, new_y = x + dx, y + dy

    if 0 <= new_x < 2 and 0 <= new_y < 2:
        new_state = [list(fila) for fila in estado]  # Copia el estado
        new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
        return new_state
    return None
```

El método *move* es el que se encarga de mover el espacio vacío mediante actualizaciones en la matriz. Toma como parámetros la matriz que representa el tablero, una tupla que representa la posición actual del valor 0 y una tupla que representa el movimiento a realizar. Para realizar el movimiento, se suman los valores de la posición del elemento vacío con las nuevas coordenadas y se verifica que no pasen el limite del tablero. Se realiza una copia de la lista y se realiza el intercambio de posiciones entre el elemento 0 y el número que estaba en la posición.

```python
def its_final_state(self, estado): #Encontrar estado final
    return estado == self.final_state

def print_state(self, estado): #Imprimir el estado del tablero
    for fila in estado:
        print(fila)
    print()
```

El primer método *its_final_state* se encarga de verificar si se ha llegado al estado objetivo, mientras que el método *print_state* se encarga de imprimir el estado del tablero.

```python
def dfs(self):
    stack = Pila()
    stack.push((self.initial_state, []))  # Guardar el estado y el camino
    visited = set()

    while not stack.empty():
        actual_state, path = stack.pop()

        if self.its_final_state(actual_state):
            return path  # Se encontró el estado objetivo

        estado_tuple = tuple(map(tuple, actual_state))  # Convertimos a tupla para poder usar en conjuntos
        if estado_tuple in visited:
            continue

        visited.add(estado_tuple)

        pos_void = self.find_void(actual_state)

        for i, direction in enumerate(self.moves):
            new_state = self.move(actual_state, pos_void, direction)
            if new_state and tuple(map(tuple, new_state)) not in visited:
                stack.push((new_state, path + [(new_state, self.name_moves[i])]))

    return None  # No se encontró solución
```

El método *dfs* es el relacionado al algoritmo de búsqueda profunda, donde se utiliza una lista, una pila que se encarga de guardar el estado inicial y el camino vacío y una lista para almacenar los estados visitados.

Se llama al método *its_final_state* en caso de encontrar el estado objetivo. En otro caso, convertimos el estado actual en una tupla, para posteriormente ser almacenada en la lista de estados visitados.

Se llaman a los métodos *find_void* y *move* para encontrar el espacio vacío y moverlo en las cuatro direcciones posibles, si el estado no ha sido visitado entonces se agrega a la pila.

```
def show_solution(self, path):
    print("Estado inicial:")
    self.print_state(self.initial_state)

    for step, (estado, movimiento) in enumerate(path, 1):
        print(f"Paso {step}: Mover {movimiento}")
        self.print_state(estado)

    print("Estado objetivo alcanzado")
```

El método *show_solution* es el encargado de llamar al método *print_state*, con el objetivo de imprimir cada uno de los movimientos realizados en el tablero.

```
initial_state = [
    [1, 2],
    [0, 3]
]

final_state = [
    [1, 2],
    [3, 0]
]

puzzle = Puzzle4(initial_state, final_state)
path = puzzle.dfs()

if path:
    puzzle.show_solution(path)
else:
    print("No se encontró solución")
```

Finalmente, se declara el estado inicial y el estado al que se quiere llegar. Estas variables sirven como parámetro de la clase *Puzzle4*. Se llama al método *dfs* para buscar la solución y se imprime.

Resultados:

```
Estado inicial:
[1, 2]
[0, 3]

Paso 1: Mover Arriba
[0, 2]
[1, 3]

Paso 2: Mover Derecha
[2, 0]
[1, 3]

Paso 3: Mover Abajo
[2, 3]
[1, 0]

Paso 4: Mover Izquierda
[2, 3]
[0, 1]

Paso 5: Mover Arriba
[0, 3]
[2, 1]

Paso 6: Mover Derecha
[3, 0]
[2, 1]

Paso 7: Mover Abajo
[3, 1]
[2, 0]

Paso 8: Mover Izquierda
[3, 1]
[0, 2]

Paso 9: Mover Arriba
[0, 1]
[3, 2]

Paso 10: Mover Derecha
[1, 0]
[3, 2]

Paso 11: Mover Abajo
[1, 2]
[3, 0]

Estado objetivo alcanzado
```

Como se puede observar, se imprimen todos los movimientos realizados hasta que el algoritmo de búsqueda en profundidad encuentra la solución.

3er ejercicio: Utilizar el algoritmo DFS para hallar la solución a un laberinto propuesto.

```
class Laberinto: #Declaración de método con constructor
    def __init__(self, maze, start, exit):
        self.maze = maze
        self.n = len(maze)
        self.start = start
        self.exit = exit
        self.moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # derecha, abajo, izquierda, arriba
        self.path = []
```

Declaramos la clase Laberinto con su constructor que toma como parámetros el un arreglo que servirá como tablero, una tupla que servirá como punto de entrada y una tupla que servirá como punto de salida. Además, se establecen los movimientos posibles y una lista que almacenara el camino.

```
def valid_position(self, x, y):
    return 0 <= x < self.n and 0 <= y < self.n and self.maze[x][y] == 0
```

El método *valid_position* se encarga de verificar que los movimientos no se salgan del limite del tablero.

```
def dfs(self, x, y):
    if (x, y) == self.exit:
        self.path.append((x, y))
        return True #Encuentra salida

    # Marcar la posición actual como parte del camino.
    self.maze[x][y] = 2  # Marcamos con el número para evitar volver a pasar.
    self.path.append((x, y))
    self.show_maze()

    for move in self.moves:
        new_x, new_y = x + move[0], y + move[1]
        if self.valid_position(new_x, new_y):
            if self.dfs(new_x, new_y):
                return True  # Movimientos hasta encontrar salida

    # Backtracking
    self.path.pop()
    self.maze[x][y] = 0  # Desmarcamos el camino
    self.show_maze()
    return False
```

El método *dsf* funciona de la misma forma que el ejercicio anterior, con la diferencia de que se agrega un proceso de backtracking: Al pasar por un camino posible este se marca con el número dos, pero si se llega a una posición donde no se pueden realizar mas movimiento y no corresponde a la tupla que representa el punto de salida entonces se empiezan a sacar elementos de la fila y el elemento eliminado se marco con un 0.

```python
def resolve(self): #Resolver desde posicion de inicio
    x_initial, y_initial = self.start
    if not self.dfs(x_initial, y_initial):
        print("No se encontró un camino a la salida")
    else:
        print("Se encontró un camino a la salida")
        print("Camino recorrido:", self.path)
```

El método *resolve* se encarga de tomar la posición de inicio y llama al método *dfs* para hallar una solución posible. En cada caso de encontrar o no una salida se muestra un mensaje. Además, se muestra el camino recorrido.

```python
def show_maze(self):
    print("Laberinto:")
    for fila in self.maze:
        print(fila)
    print()
```

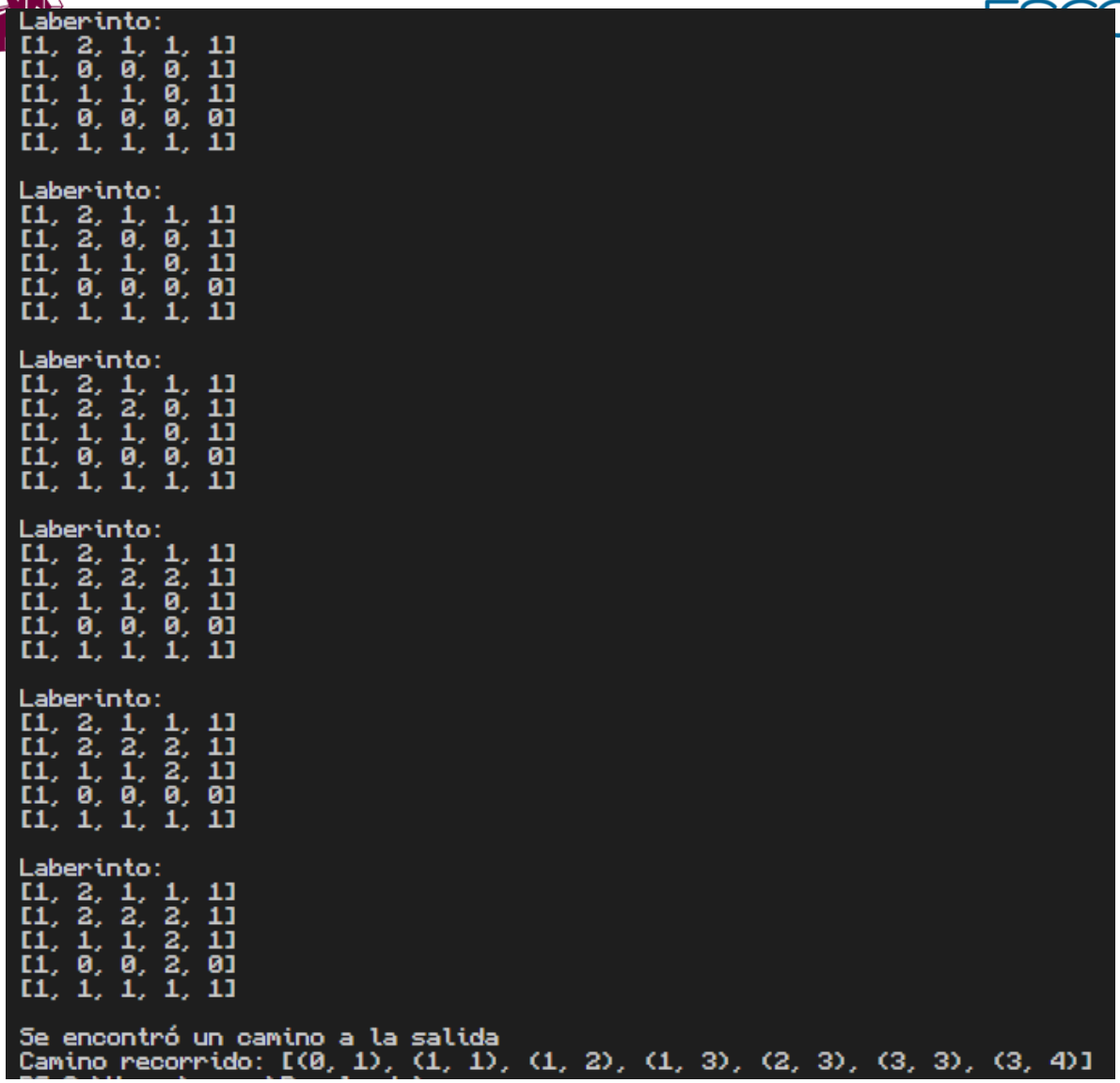
El método *show_maze* se encarga de imprimir cada cambio en el laberinto.

```python
# Definición del laberinto (1 representa pared, 0 representa camino)
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)
exit = (3, 4)

Maze_example = Laberinto(maze, start, exit)
Maze_example.resolve()
```

Finalmente, definimos como será el tablero, donde el 1 representa una pared y el 0 representa el camino posible. Se establecen los puntos de entrada y salida y se llama a la clase *Laberinto* y al método *resolve* para mostrar la solución.

Resultados:



Como se puede observar, al llamar al método *resolve* con la posición inicial y la posición final se obtiene cada instancia del laberinto al realizar el recorrido, donde el algoritmo dfs se encarga de mostrar recorrer cada camino posible hasta encontrar la salida. En caso de encontrarse en un punto muerto se realiza el proceso de backtracking para volver a una posición anterior e ir por otro camino. También se muestran los movimientos realizados mediante una lista de tuplas.

Conclusiones

Una vez finalizada la primera parte de la práctica de búsqueda no informada podemos concluir lo siguiente:

El uso del algoritmo Depth First Search para resolver este tipo de problemas resulta de una gran eficacia y eficiencia, pues es útil para resolver problemas que solo cuentan con una única solución. El tiempo que le toma al algoritmo resolver los problemas resulta corto pues los estados que componen cada ejercicio son pocos en comparaciones a problemas de mayor dificultad como puede ser el Puzzle – 8, donde el tiempo para resolverlo usando el algoritmo crece exponencialmente.

El uso de listas, colas y filas en Python resulta de mucha ayuda pues es de fácil implementación y comprensión, con ligeras modificaciones en sus operaciones. El uso de estas estructuras de datos resulta obligatorio para poder desarrollar el algoritmo de búsqueda profunda, pues la parte primordial del algoritmo es la de implementar una fila para poder agregar y sacar según se requiera, tal es el caso del backtracking en el tercer ejercicio, ya que sin el uso de filas no seria posible salir de estados donde ya no es posible realizar mas movimientos y nos encontramos lejos de la solución.

Además, el uso de este algoritmo resulta útil cuando necesitamos obtener todos los caminos posibles antes de tener que retroceder y probar más alternativas de solución. También resulta interesante su fácil implementación a nivel de lógica computacional. Sin embargo, también cuenta con algunas desventajas, ya que no siempre se mostrará el resultado más optimo globalmente, lo que no garantiza encontrar la mejor solución posible. Del mismo modo, se presentan problemas de memoria al generar recursión debido al tamaño de la pila, este problema es apreciable en problemas de escala mayor.

Referencias

- Zalimben, S. (2022, 16 marzo). *Pilas y colas*. DEV Community. https://dev.to/szalimben/pilas-y-colas-b3

- Murillo, J. (2022, 18 julio). Difference between Breadth Search (BFS) and Deep Search (DFS). *Encora*. https://www.encora.com/es/blog/dfs-vs-bfs

-