INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

INGENIERIA EN SISTEMAS COMPUTACIONALES

Ejercicio de laboratorio 2

“Depth-Firts Search”

Presentan

GOMEZ HERNANDEZ ALAN JAVIER
HERNÁNDEZ PÉREZ JUAN MANUEL
JIMÉNEZ CRUZ DANIEL
RIVAS CARRERA DIANA LAURA

Profesor

ANDRÉS GARCÍA FLORIANO

Ciudad de México, a 20 de septiembre 2023

INTRODUCCION

El Depth First Search (DFS), que significa "Búsqueda en Profundidad" en español, es un algoritmo utilizado en informática y matemáticas para recorrer o buscar en estructuras de datos como árboles y grafos. DFS se utiliza comúnmente para explorar y encontrar soluciones en problemas relacionados con la búsqueda y la conectividad en estas estructuras. El algoritmo se puede implementar de manera recursiva o mediante el uso de una pila (enfoque iterativo).

El principio básico del algoritmo DFS es explorar tan profundamente como sea posible a lo largo de un camino antes de retroceder y explorar otros caminos. Esto significa que el algoritmo seguirá profundizando en un nodo hijo antes de explorar otros nodos hermanos. En otras palabras, DFS se asemeja a explorar un laberinto siguiendo un camino hasta llegar a un punto muerto antes de retroceder y probar otro camino.

El DFS es utilizado en una variedad de aplicaciones, incluyendo:

1. Búsqueda de rutas: Puede utilizarse para encontrar un camino desde un nodo de inicio hasta un nodo objetivo en un grafo o un mapa.
2. Análisis de conectividad: Se utiliza para determinar si un grafo es conexo o para encontrar componentes conectados en un grafo.
3. Resolución de problemas en inteligencia artificial: Se utiliza en algoritmos como la búsqueda en profundidad limitada (LDFS) y la búsqueda en profundidad iterativa (IDDFS) para encontrar soluciones en árboles de juego, planificación de rutas y otros problemas.
4. Algoritmos de búsqueda y backtracking: El DFS es un componente fundamental en algoritmos de búsqueda en árboles y grafos, así como en algoritmos de backtracking, como la resolución de problemas de búsqueda de soluciones.
5. Topología de grafos: Se utiliza para encontrar el orden topológico en un grafo dirigido, lo que es importante en la planificación de proyectos y en el procesamiento de dependencias.

El DFS no garantiza encontrar la solución óptima en todos los casos, ya que puede quedar atrapado en un camino que no lleva a la solución óptima si no se establece una condición de parada adecuada. Sin embargo, en muchos casos, el DFS es una herramienta eficaz para explorar y buscar en estructuras de datos, especialmente en problemas donde la exploración profunda es útil.

DESARROLLO

1. 4-Puzzle

Se adjunta se encuentra el código para el modelado y solución del problema del 4-puzzle; sin embargo, hay un problema con el código de la estructura auxiliar (el árbol).

Analiza el código proporcionado y determina cuál es el problema que impide que funcione correctamente y corrígelo.

En caso de que lo consideres necesario puedes reescribir partes o todo el código desarrollado.

El código implementado hace una búsqueda en profundidad para encontrar una solución al Puzle Lineal, donde los estados se representan como listas y los movimientos válidos son intercambiar dos números adyacentes en la lista. El resultado muestra la secuencia de movimientos necesarios para llegar desde el estado inicial al estado objetivo.

Líneas del código árbol.py corregidas para su correcto funcionamiento:

```
def en_lista(self, lista_nodos):
    en_la_lista = False
    for n in lista_nodos:
        if self.igual(n):
            en_la_lista = True
    return en_la_lista
```

→ return en_la_lista; Se encontraba dentro de un ciclo for por lo que se quedaba en el ciclo no regresaba nada

```
def igual(self, nodo):
    if self.get_datos() == nodo.get_datos():
        return True
    else:
        return False
```

→ Se simplifica a;

```
def igual(self, nodo):
    return self.get_datos() == nodo.get_datos()
```

Para devolver el dato evaluado directamente

Árbol.py

```
class Nodo:
    def __init__(self, datos, hijos=None):
        self.datos = datos
        self.hijos = None
        self.padre = None
```

```
        self.coste = None
        self.set_hijos(hijos)

    def set_hijos(self, hijos):
        self.hijos = hijos
        if self.hijos != None:
            for h in self.hijos:
                h.padre = self

    def get_hijos(self):
        return self.hijos

    def get_padre(self):
        return self.padre

    def set_padre(self, padre):
        self.padre = padre

    def set_datos(self, datos):
        self.datos = datos

    def get_datos(self):
        return self.datos

    def set_coste(self, coste):
        self.coste = coste

    def get_coste(self):
        return self.coste

    def igual(self, nodo):
        return self.get_datos() == nodo.get_datos()

    def en_lista(self, lista_nodos):
        en_la_lista = False
        for n in lista_nodos:
            if self.igual(n):
                en_la_lista = True
        return en_la_lista

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

Puzzle.py

```
# Puzle Lineal con búsqueda en profundidad
from arbol import Nodo

def buscar_solucion_DFS(estado_inicial, solucion):
    solucionado = False
    nodos_visitados = []
    nodos_frontera = []
    nodoInicial = Nodo(estado_inicial)
    nodos_frontera.append(nodoInicial)
```

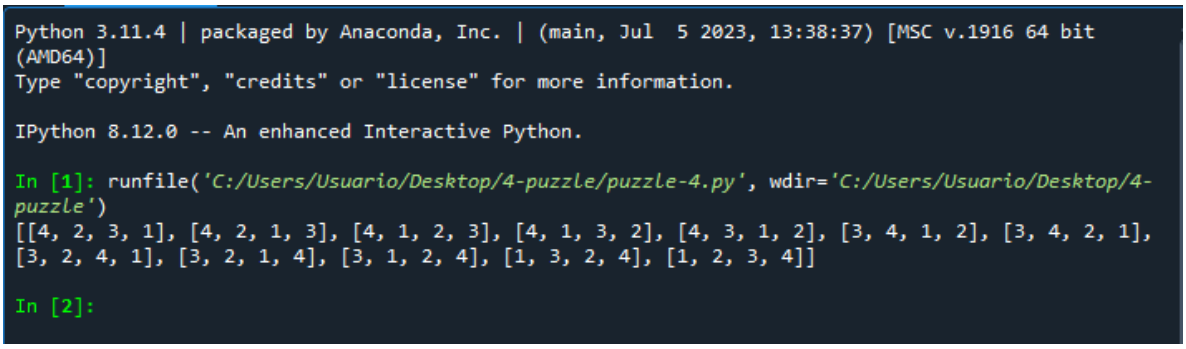
```
    while (not solucionado) and len(nodos_frontera) != 0:
        nodo = nodos_frontera.pop()
        # extraer nodo y añadirlo a visitados
        nodos_visitados.append(nodo)

        if nodo.get_datos() == solucion:
            # solución encontrada
            solucionado = True
            return nodo
        else:
            # expandir nodos hijo
            dato_nodo = nodo.get_datos()
            # operador izquierdo
            hijo = [dato_nodo[1], dato_nodo[0], dato_nodo[2],
dato_nodo[3]]
            hijo_izquierdo = Nodo(hijo)
            if not hijo_izquierdo.en_lista(nodos_visitados) \
                    and not hijo_izquierdo.en_lista(nodos_frontera):
                nodos_frontera.append(hijo_izquierdo)
            # operador central
            hijo = [dato_nodo[0], dato_nodo[2], dato_nodo[1],
dato_nodo[3]]
            hijo_central = Nodo(hijo)
            if not hijo_central.en_lista(nodos_visitados) \
                    and not hijo_central.en_lista(nodos_frontera):
                nodos_frontera.append(hijo_central)
            # operador derecho
            hijo = [dato_nodo[0], dato_nodo[1], dato_nodo[3],
dato_nodo[2]]
            hijo_derecho = Nodo(hijo)
            if not hijo_derecho.en_lista(nodos_visitados) \
                    and not hijo_derecho.en_lista(nodos_frontera):
                nodos_frontera.append(hijo_derecho)


            nodo.set_hijos([hijo_izquierdo, hijo_central, hijo_derecho])



if __name__ == "__main__":
    estado_inicial=[4,2,3,1]
    solucion=[1,2,3,4]
    nodo_solucion = buscar_solucion_DFS(estado_inicial, solucion)
    # mostrar resultado
    resultado=[]
    nodo=nodo_solucion
    while nodo.get_padre() != None:
        resultado.append(nodo.get_datos())
        nodo = nodo.get_padre()

    resultado.append(estado_inicial)
    resultado.reverse()
    print(resultado)
```

maze-solver.py

```
import random
def find_doors(maze):
    start, end = None, None

    for y in range(len(maze)):
        for x in range(len(maze[0])):
            if maze[y][x] == 'S':
                start = (y, x)
            elif maze[y][x] == 'E':
                end = (y, x)

    return start, end
def solve_maze(maze, start, end):

    stack = [start]
    while stack:
        x, y = stack[-1]  # Cima de la pila

        # If reached the end point
        if (x, y) == end:
            return True, stack

        # Mark as visited
        if maze[x][y] != 'S':
            maze[x][y] = '-'

        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            nx, ny = x + dx, y + dy

            if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]):
                if maze[nx][ny] == ' ' or maze[nx][ny] == 'E':
                    stack.append((nx, ny))
                    break
        else:
            stack.pop()

    return False, []


def print_maze(maze):
    for row in maze:
        print(' '.join(row))


def create_maze(rows, cols):
    # Crear un laberinto vacío
    maze = [[' ' for _ in range(cols)] for _ in range(rows)]

    # Establecer las paredes fronterizas
    for row in range(rows):
        for col in range(cols):
            if row == 0 or row == rows - 1 or col == 0 or col == cols - 1:  # Si
es una pared fronteriza
```

```
                maze[row][col] = '#'
            else:
                if random.random() < 0.40:  # Probabilidad del 40% de ser una
pared
                    maze[row][col] = '#'  # Pared

    return maze


def set_random_door(maze, type_of_door):
    rows = len(maze)
    cols = len(maze[0])

    border = random.choice(['top', 'bottom', 'left', 'right'])

    match border:
        case 'top':
            row, col = 0, random.randint(1, cols - 2)
            maze[row + 1][col] = ' '
        case 'bottom':
            row, col = rows - 1, random.randint(1, cols - 2)
            maze[row - 1][col] = ' '
        case 'left':
            row, col = random.randint(1, rows - 2), 0
            maze[row][col + 1] = ' '
        case 'right':
            row, col = random.randint(1, rows - 2), cols - 1
            maze[row][col - 1] = ' '
        case _:
            raise ValueError('Invalid border')

    if maze[row][col] == 'S' or maze[row][col] == 'E':
        return set_random_door(maze, type_of_door)
    else:
        maze[row][col] = type_of_door

    return maze


if __name__ == '__main__':
    maze = set_random_door(set_random_door(create_maze(10, 12), 'S'), 'E')
    maze_copy = [row[:] for row in maze]
    solved, path = solve_maze(maze_copy, *find_doors(maze))

    if solved:
        print("[+] Maze Solved!\n")
        for x, y in path:
            if maze[x][y] != 'S' and maze[x][y] != 'E':  # Si no es S ni E, pone
un
                maze[x][y] = '.'
        print_maze(maze)
    else:
        print("[x] No solution found.\n")
        print_maze(maze_copy)
```

PRUEBAS DE FUNCIONAMIENTO



2. Laberinto

Se proporciona un código para resolver un laberinto. Verifica que funcione correctamente y de acuerdo con la teoría estudiada del algoritmo DFS.

Propón diversas modificaciones al laberinto:

- Tamaño.
- En complejidad.
- Incluye casos en los que no tenga solución.

Mide el tiempo de ejecución del programa para cada una de las modificaciones relacionadas.

El programa desarrollado genera y resuelve laberintos aleatorios. Inicialmente, crea un laberinto de tamaño especificado, poblándolo con paredes basándose en una probabilidad dada. Luego, establece puntos de entrada y salida ('S' y 'E') en ubicaciones aleatorias en los bordes del laberinto. Una vez configurado, el programa intenta resolver el laberinto utilizando un algoritmo basado en la técnica DFS (Depth First Search). Si se encuentra una solución, se muestra el laberinto con el camino resuelto marcado; de lo contrario, se presenta el laberinto sin cambios, indicando que no se encontró una solución.

A continuación, se proporcionan dos ejemplos de laberintos creados aleatoriamente de acuerdo con las medidas dadas por el usuario (n, m):

- Ejemplo de laberinto **con solución** hallada

- Ejemplo de laberinto **sin solución** hallada

CONCLUSIÓN

La resolución de problemas utilizando algoritmos es fundamental en la ciencia de la computación, y particularmente en el ámbito de la inteligencia artificial. El puzzle-4, al igual que el laberinto, representa un desafío que pone a prueba nuestras habilidades de programación y comprensión de algoritmos. Enfrentarse a este tipo de problemas nos obliga a analizar en profundidad cómo funcionan ciertas técnicas, cuáles son sus fortalezas y dónde pueden encontrarse sus limitaciones

En la primera practica el error que se observo fue en el árbol dado que se llamaba la función en_lista esta era un ciclo infinito dado que no regresa la lista ya que está en un ciclo infinito sin embargo lo que se mueve en sacarlo de esta sentencia a fuera para que la función retorne una vez acabe el ciclo. Por otro lado, se hace una optimización de código en "igual" donde las validaciones puede arrojar directamente un true en caso de que su cumpla en vez de hace la iteración junto con el false, solo se acorto la línea de código.

La implementación del laberinto proporciona una visión práctica y clara de cómo funciona el algoritmo de Búsqueda en Profundidad (DFS, por sus siglas en inglés). A través del laberinto, se puede apreciar cómo DFS explora exhaustivamente una ruta antes de retroceder, una característica inherente de este método. Sin embargo, es importante destacar que, aunque DFS puede encontrar una solución, no garantiza que sea la más óptima en términos de longitud o eficiencia. Para problemas donde se requiere la ruta más corta, como un laberinto, otras técnicas como la Búsqueda en Amplitud (BFS) podrían ser más adecuadas. Sin embargo, el estudio de DFS a través de este ejercicio demuestra su utilidad y versatilidad en el ámbito de la inteligencia artificial y la resolución de problemas.