INSTITUTO POLITECNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO
NOMBRE: Cerda García Gustavo

Ejercicio de Laboratorio 3. BFS (Breadth First Search)

Ejercicio 1. 4-Puzzle

La solución con BFS fue mejor que la propuesta por DFS

La elección entre BFS (Búsqueda en Amplitud) y DFS (Búsqueda en Profundidad) depende del problema específico y de las características del espacio de búsqueda. Aquí hay algunas consideraciones que pueden ayudarte a decidir cuál es mejor en ciertos casos:

Búsqueda en Amplitud (BFS)

- **Completa**: Encuentra una solución si existe, siempre y cuando los costos sean iguales.
- **Óptima**: Encuentra la solución de menor profundidad, lo que significa que, en un grafo no ponderado, encuentra la solución más corta en términos de número de pasos o niveles.
- **Memoria**: Tiende a requerir más memoria, ya que debe mantener todos los nodos frontera en una cola.
- **Tiempo**: Puede ser más lento en comparación con DFS, especialmente en árboles o grafos con una profundidad considerable.
- **Aplicaciones**: Ideal para encontrar la solución más corta en términos de número de pasos, como en laberintos donde se busca el camino más corto.

Búsqueda en Profundidad (DFS)

- **Completa**: No siempre encuentra una solución si existe, especialmente en grafos con ciclos.
- **No óptima**: No garantiza encontrar la solución de menor profundidad. Encontrará la primera solución que encuentre, que puede no ser la óptima en términos de profundidad.
- **Memoria**: Tiene un uso de memoria más eficiente, ya que solo necesita mantener un camino desde el nodo inicial hasta el nodo actual.
- **Tiempo**: Puede ser más rápido que BFS en grafos con ramificaciones excesivas y profundidad limitada.
- **Aplicaciones**: Útil en problemas donde la profundidad de la solución no es relevante, como en la exploración de un árbol para encontrar una solución, la generación de permutaciones, etc.

Para la elección del algoritmo

- **Laberintos**: Si el objetivo es encontrar la solución más corta en términos de pasos, BFS es mejor.
- **Exploración de Árboles**: DFS puede ser más eficiente para recorrer árboles y grafos grandes.

- **Espacios de Búsqueda con Profundidad Limitada**: DFS puede ser preferible si se conoce que la solución no está demasiado lejos del nodo inicial.
- **Memoria Limitada**: En dispositivos con recursos limitados, DFS puede ser preferible ya que utiliza menos memoria.

a. Analiza la propuesta de backtracking proporcionada en este código ¿es eficiente en memoria?

Vamos a comparar los resultados y el tamaño de los nodos en los dos algoritmos.

Búsqueda en Amplitud (BFS)

```
Ejercicio 1: BFS - 4 Puzzle
[[4, 2, 3, 1], [2, 4, 3, 1], [2, 3, 4, 1], [2, 3, 1, 4], [2, 1, 3, 4], [1, 2, 3, 4]]
Uso de memoria:
Nodos visitados:  248
Nodos frontera:  120
```

Búsqueda en Profundidad (DFS)

```
Ejercicio 1: DFS - 4 Puzzle
[[4, 2, 3, 1], [4, 2, 1, 3], [4, 1, 2, 3], [4, 1, 3, 2], [4, 3, 1, 2], [3, 4, 1, 2], [3, 4, 2, 1], [3,
2, 4, 1], [3, 2, 1, 4], [3, 1, 2, 4], [1, 3, 2, 4], [1, 2, 3, 4]]
Uso de memoria:
Nodos visitados:  184
Nodos frontera:  184
```

Podemos observar que en DFS se visitan mas hijos para llegar a la solución, pero al no guardarse, el uso de la memoria es menor que en la BFS, ahí se visitan menos nodos para llegar a la solución, pero al guardar los nodos, la memoria aumenta, pero se llega a la solución en menos pasos. Como ya habíamos visto, el uso de la memoria es una limitante que se tiene al momento de seleccionar el algoritmo para resolver una problemática.

Ejercicio 2. Laberinto

¿Qué diferencias se tiene con respecto al método DFS?

Primer Código (Usando deque para BFS):

- Utiliza la estructura de datos deque de la biblioteca collections.

- Define la función solve_maze que toma el laberinto, la posición inicial y la posición final como entrada.
- Inicia una cola (queue) con la tupla que contiene la posición inicial y una lista que contiene solo la posición inicial como camino inicial.
- Luego, en un bucle while, extrae el primer elemento de la cola y verifica si es el destino final.
- Si no es el destino final, marca la posición actual como visitada (maze[x][y] = '2') y expande los nodos vecinos válidos.
- Para cada vecino válido, crea una nueva tupla con la posición del vecino y el nuevo camino (agregando el vecino al camino actual).
- Agrega esta nueva tupla a la cola.
- Si encuentra la solución, devuelve True y el camino hasta ese punto.
- Al final, si no se encuentra una solución, devuelve False y una lista vacía.

Segundo Código (Utilizando una Función Externa solve_maze):

- Este código importa una función solve_maze desde un módulo externo llamado Laberinto.
- En el módulo Laberinto, está la definición de la función solve_maze, que es similar a la del primer código.
- Sin embargo, en este código, no se ve la implementación de la función solve_maze directamente en el archivo principal.
- En lugar de eso, se llama a la función solve_maze pasándole el laberinto, la posición inicial y la posición final.
- Luego, se maneja el resultado de la función solve_maze de manera similar al primer código: si se encuentra una solución, marca el camino y lo muestra; de lo contrario, muestra un mensaje de "No solution found".

Comparación:

- El primer código tiene todo el código dentro del mismo archivo y utiliza una cola deque para implementar BFS de manera explícita.
- El segundo código separa la lógica de resolución del laberinto en un módulo externo Laberinto y lo importa para su uso.
- En términos de funcionalidad, ambos códigos hacen lo mismo: resuelven el laberinto y muestran el camino si se encuentra.
- El segundo código puede ser más modular y organizado, ya que separa la lógica de la resolución del laberinto en un archivo separado, lo que puede ser útil para proyectos más grandes o con múltiples funciones.
- El primer código es más autónomo y autocontenido en el archivo principal, lo que puede ser más sencillo para problemas más simples o como ejemplo de implementación directa de BFS en un laberinto.

Programa en funcionamiento

- Laberinto Normal con solución

```
Laberinto con soluci♦n (5x5)
Maze Solved!
11111
**101
1*121
1***E
11111
Uso de memoria:
Solved:  28
Path:  112

[Done] exited with code=0 in 0.103 seconds
```

- Laberinto más grande con solución

```
Laberinto m♦s grande con soluci♦n (13x13)
Maze Solved!
1111111111111
****101010001
111*101010101
1***101010101
1*11101010101
1*10001010101
1*10100010101
1*10111110101
1*10000000101
1*11111110101
1*******10101
1111111*10101
1E******10101
1111111111111
Uso de memoria:
Solved:  28
Path:  288

[Done] exited with code=0 in 0.095 seconds
```

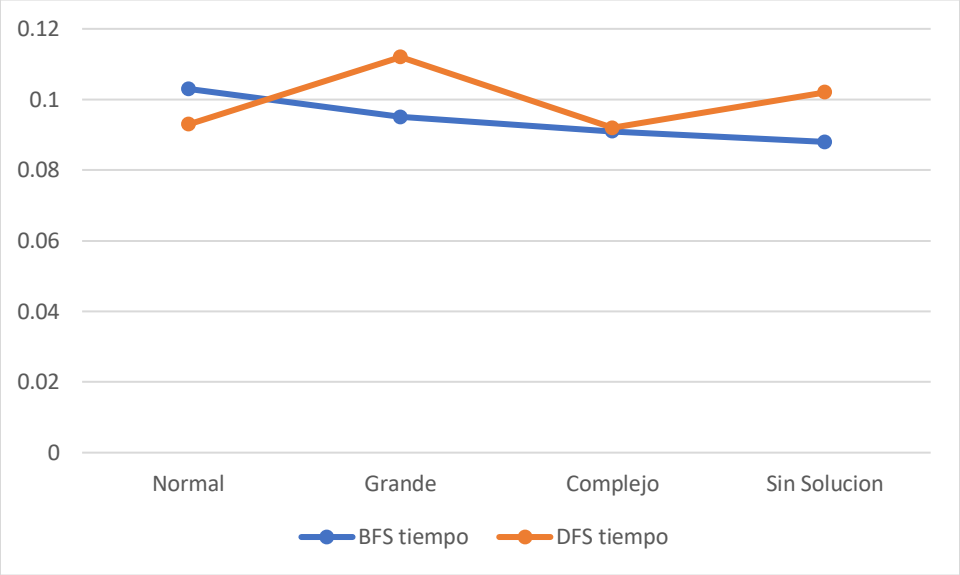
- Laberinto complejo

```
Laberinto m♦s grande y complejo con soluci♦n (10x10)
Maze Solved!
1111111111
**12121011
1*12121011
1*****1011
11121*1011
12121*1011
12121*1011
12222*1011
11111*1011
11111E1001
Uso de memoria:
Solved:  28
Path:  168

[Done] exited with code=0 in 0.091 seconds
```

- Laberinto sin solución

```
Laberinto m♦s grande y complejo sin soluci♦n (25x16)
No solution found.
Uso de memoria:
Solved:  28
Path:  56

[Done] exited with code=0 in 0.088 seconds
```

Comparación de resultados



	BFS tiempo	DFS tiempo
Normal	0.103	0.093
Grande	0.095	0.112
Complejo	0.091	0.092
Sin Solucion	0.088	0.102