INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO
INTELIGENCIA ARTIFICIAL
ING. EN SISTEMAS COMPUTACIONALES

Laboratorio 2: Búsqueda no informada P1

Integrantes:

Hurtado Morales Emiliano - 2021630390

Maestro: Andrés García Floriano

Grupo: 6CV3

Fecha de entrega: 20/09/2024

Ciclo Escolar: 2025 – 1

Introducción

La búsqueda de soluciones a problemas que implican múltiples configuraciones y caminos posibles es un área importante en la informática y las ciencias computacionales. Dos problemas clásicos que ejemplifican este tipo de desafío son el 4-Puzzle en una línea y el problema del laberinto. Ambos requieren técnicas de búsqueda para explorar y encontrar una solución, ya que, en ambos casos, existen muchas posibilidades que deben ser analizadas exhaustivamente.

El 4-Puzzle en una línea es un problema sencillo de permutación donde cuatro números desordenados deben ser ordenados mediante intercambios consecutivos de elementos adyacentes. Aunque el problema parece simple debido al pequeño número de elementos, la cantidad de configuraciones posibles aumenta rápidamente, lo que requiere una estrategia sistemática para explorar las distintas permutaciones y encontrar el orden correcto.

Por otro lado, el problema del laberinto consiste en encontrar un camino válido desde un punto de inicio hasta una salida en una matriz que representa el laberinto. Algunas celdas en el laberinto son inaccesibles (paredes), mientras que otras son caminos transitables. Este problema es muy similar a muchos desafíos de navegación que aparecen en el campo de la robótica, los videojuegos y las simulaciones, donde es necesario explorar todas las rutas posibles hasta encontrar una solución óptima.

Una de las técnicas más utilizadas para este tipo de problemas es el algoritmo de Búsqueda en Profundidad (DFS, Depth-First Search), el cual explora exhaustivamente todas las posibles configuraciones de un problema antes de retroceder y probar alternativas. DFS es particularmente útil en problemas que requieren la búsqueda de soluciones en grandes espacios de estado, ya que permite realizar búsquedas sistemáticas con un consumo moderado de memoria, en comparación con otros algoritmos de búsqueda.

Este reporte aborda la implementación de DFS para resolver tanto el 4-Puzzle en una línea como el problema del laberinto. Para gestionar eficientemente los estados explorados y evitar ciclos, se ha desarrollado una biblioteca personalizada que incluye estructuras de datos como pila y lista para manejar los caminos y estados visitados. Estas herramientas permiten una implementación clara y eficiente de los algoritmos de búsqueda, evitando problemas de sobrecarga computacional que podrían surgir al manejar grandes cantidades de estados o caminos.

A lo largo de este informe, se explicará en detalle la implementación de las estructuras de datos, los algoritmos para ambos problemas, así como los resultados esperados y las conclusiones derivadas de las pruebas realizadas.

Explicación de la Biblioteca

Enlace de GitHub: https://github.com/EmilianoHM/Bender-IA/tree/main/Lab_2

Antes de entrar en los algoritmos específicos, es importante entender la **biblioteca** utilizada. Esta proporciona las estructuras de datos necesarias para implementar DFS.

1. Clase Pila (Stack)

La **pila** es fundamental para la implementación de DFS, ya que nos permite gestionar los caminos que se exploran en el algoritmo. La pila sigue una estructura **LIFO** (Last In, First Out), lo que significa que el último estado insertado es el primero en salir.

- **apilar(objeto)**: Agrega un objeto al final de la pila (lo que representa un nuevo estado que queremos explorar).
- **desapilar()**: Retira y devuelve el último elemento de la pila.
- **esta_vacia()**: Devuelve True si la pila no contiene ningún elemento.
- **cima()**: Devuelve el elemento en la parte superior de la pila sin retirarlo.

Esta clase se utiliza para manejar la lista de caminos que estamos explorando en DFS.

2. Clase Lista

La clase **Lista** se usa para almacenar los estados que ya han sido visitados, evitando así revisitar los mismos nodos o configuraciones. Esto es crucial para evitar ciclos en los problemas de búsqueda.

- **insertar(objeto)**: Inserta un objeto al final de la lista.
- **buscar(objeto)**: Busca si un objeto ya existe en la lista. Es fundamental para saber si un estado ya ha sido explorado.

3. Clase Cola (Queue)

La cola es una estructura de datos de tipo **FIFO** (First In, First Out), lo que significa que el primer elemento en entrar es el primero en salir. Aunque no se usa directamente en DFS, es útil para implementar algoritmos como BFS (Búsqueda en Amplitud).

- **insertar(objeto)**: Agrega un objeto al final de la cola.
- **quitar()**: Retira y devuelve el objeto al frente de la cola.
- **esta_vacia()**: Devuelve True si la cola está vacía.
- **recorrer()**: Recorre e imprime todos los elementos de la cola.

Explicación de los Algoritmos

1. 4-Puzzle en una línea

Este puzzle consiste en ordenar cuatro números desordenados, como [4, 2, 1, 3], en el orden correcto [1, 2, 3, 4], intercambiando solo elementos adyacentes. Aquí, DFS nos ayuda a explorar todas las posibles combinaciones de intercambios hasta llegar a la solución.

Descripción del Código:

- **Estado Inicial**: Se define una lista de números desordenados. En este caso, el estado inicial es [4, 2, 1, 3].
- **Generación de Nuevos Estados**: La función **generar_nuevos_estados(estado_actual)** es responsable de generar todas las permutaciones posibles de la lista al intercambiar elementos adyacentes. Recorre la lista y genera una nueva lista con dos elementos intercambiados, creando un nuevo estado.

```
# Función que genera nuevos estados intercambiando pares de elementos adyacentes
def generar_nuevos_estados(estado_actual):
    nuevos_estados = []
    for i in range(len(estado_actual) - 1):
        # Intercambiar elementos adyacentes
        nuevo_estado = estado_actual[:]
        nuevo_estado[i], nuevo_estado[i + 1] = nuevo_estado[i + 1],
nuevo_estado[i]
        nuevos_estados.append(nuevo_estado)
    return nuevos_estados
```

- **DFS para resolver el puzzle**: El DFS se implementa en la función **dfs_4puzzle_linea**. Utiliza una pila para explorar los caminos y la clase Lista para almacenar los estados visitados, garantizando que no se revisite un estado ya explorado.
    - Se inicializa la pila con el estado inicial.
    - Mientras la pila no esté vacía, se toma el último estado añadido (último camino explorado).
    - Si el estado actual es el objetivo (lista ordenada), el algoritmo retorna la secuencia de pasos que llevó a esa solución.
    - Si el estado actual no es el objetivo y no ha sido visitado, se generan nuevos estados intercambiando elementos adyacentes, y esos nuevos caminos se agregan a la pila.

```
    while not pila.esta_vacia():
        camino = pila.desapilar()
        estado_actual = camino[-1]

        # Verificamos si hemos llegado al estado objetivo
        if estado_actual == estado_objetivo:
            return camino
```

```
        estado_tupla = tuple(estado_actual)  # Convertimos el estado en tupla
para buscarla en la lista de visitados

        if not visitado.buscar(estado_tupla):  # Usamos la función buscar de la
clase Lista
            visitado.insertar(estado_tupla)  # Insertamos el estado actual en la
lista de visitados

            # Generamos los nuevos estados y los agregamos a la pila
            for nuevo_estado in generar_nuevos_estados(estado_actual):
                nuevo_camino = camino[:]
                nuevo_camino.append(nuevo_estado)
                pila.apilar(nuevo_camino)
```

2. Laberinto

El segundo problema implica encontrar un camino desde un punto de inicio hasta un punto de destino en un laberinto representado por una matriz de celdas. En este caso, las celdas con 1 son paredes (bloqueadas), mientras que las celdas con 0 son caminos transitables.

Descripción del Código:

- **Movimientos Posibles**: En el laberinto, podemos movernos en cuatro direcciones: arriba, abajo, izquierda y derecha. Estos movimientos se representan mediante las coordenadas (fila, columna).

```
# Movimientos posibles: Arriba, Abajo, Izquierda, Derecha
movimientos = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Representados como (fila,
columna)
```

- **Generación de Nuevos Estados**: La función **generar_nuevos_estados(laberinto, posicion_actual)** genera los nuevos estados (posiciones) posibles desde la posición actual. Verifica que el nuevo movimiento esté dentro de los límites del laberinto y que no sea una pared.

```
# Función para generar nuevos estados a partir de la posición actual
def generar_nuevos_estados(laberinto, posicion_actual):
    nuevos_estados = []
    fila_actual, columna_actual = posicion_actual

    for movimiento in movimientos:
        nueva_fila = fila_actual + movimiento[0]
        nueva_columna = columna_actual + movimiento[1]
        nueva_posicion = (nueva_fila, nueva_columna)

        if es_posicion_valida(laberinto, nueva_posicion):
```

```
            nuevos_estados.append(nueva_posicion)

    return nuevos_estados
```

- **DFS para resolver el laberinto**: Similar al puzzle, se usa DFS para explorar caminos desde la posición inicial hasta la posición objetivo. En cada paso, se generan nuevas posiciones a las que se puede mover, y se exploran hasta encontrar la salida.

```
    while not pila.esta_vacia():
        camino = pila.desapilar()
        posicion_actual = camino[-1]

        # Verificamos si hemos llegado al objetivo
        if posicion_actual == objetivo:
            return camino

        # Verificamos si ya hemos visitado la posición actual
        if not visitado.buscar(posicion_actual):  # Usamos la función buscar de
la clase Lista
            visitado.insertar(posicion_actual)  # Insertamos la posición actual
en la lista de visitados

            # Generamos los nuevos estados y los agregamos a la pila
            for nueva_posicion in generar_nuevos_estados(laberinto,
posicion_actual):
                nuevo_camino = list(camino)
                nuevo_camino.append(nueva_posicion)
                pila.apilar(nuevo_camino)
```

Resultados Esperados

1. 4-Puzzle

El DFS debe encontrar una secuencia de intercambios de elementos adyacentes que lleve desde el estado inicial hasta el estado objetivo. Un ejemplo de la secuencia esperada es:

```
4-puzzle:

Solución encontrada:
[4, 2, 1, 3]
[4, 2, 3, 1]
[4, 3, 2, 1]
[4, 3, 1, 2]
[4, 1, 3, 2]
[4, 1, 2, 3]
[1, 4, 2, 3]
[1, 4, 3, 2]
[1, 3, 4, 2]
[1, 3, 2, 4]
[1, 2, 3, 4]
```

2. Laberinto

El algoritmo debe encontrar un camino válido desde la posición inicial hasta la posición objetivo dentro de los límites del laberinto. Un ejemplo de resultado esperado es:

```python
# Representación del laberinto
laberinto = [
    [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]
]
```

```
Laberinto:

Solución encontrada:
(0, 1)
(1, 1)
(1, 2)
(1, 3)
(2, 3)
(3, 3)
(3, 4)
```

Conclusión

El uso de algoritmos de búsqueda como el Búsqueda en Profundidad (DFS) ha demostrado ser una técnica efectiva para resolver problemas que implican la exploración exhaustiva de posibles configuraciones, como el 4-Puzzle en una línea y el problema del laberinto. Ambos problemas, aunque diferentes en su naturaleza, presentan desafíos comunes en términos de la exploración de espacios de estado y la identificación de caminos óptimos.

En el caso del 4-Puzzle en una línea, DFS permite explorar todas las permutaciones posibles de los números hasta encontrar la secuencia correcta. A pesar de que este problema tiene un espacio de soluciones reducido, la implementación del algoritmo con estructuras de datos eficientes, como la pila y la lista implementadas en la biblioteca, garantiza que se pueda realizar la búsqueda sin repetir estados ya visitados. Esto minimiza el consumo de recursos y asegura que el algoritmo funcione de manera eficiente.

El problema del laberinto, por otro lado, representa un reto más complejo debido a la naturaleza bidimensional del espacio de búsqueda. Aun así, DFS logra navegar a través de los distintos caminos posibles, identificando la ruta correcta desde el punto de inicio hasta la salida, siempre que esta exista. La implementación de las estructuras de datos adecuadas para manejar los caminos y los estados visitados es crucial en este caso, ya que evita ciclos y asegura que el algoritmo no se quede atrapado en rutas sin salida.

Los resultados obtenidos muestran que la correcta implementación de DFS, combinada con una buena gestión de los estados visitados, permite resolver ambos problemas de manera eficiente. En situaciones más complejas, donde el espacio de búsqueda es más grande o los obstáculos son más numerosos, podría considerarse el uso de otras técnicas de búsqueda.

En resumen, los experimentos realizados muestran que DFS, apoyado por estructuras de datos adecuadas, es una solución robusta para problemas de búsqueda de este tipo. Sin embargo, es importante tener en cuenta las limitaciones de DFS en problemas de gran escala o cuando se requiere una solución óptima en términos de distancia o recursos, ya que puede no ser la técnica más adecuada en todos los escenarios.