INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

INGENIERÍA EN SISTEMAS COMPUTACIONALES

Inteligencia Artificial

Laboratorio 4: Búsqueda Informada

Marco Isaí Flores Vicencio

Jesús Pérez González

Sebastián Reyes Núñez

Grupo:

6CV3

Introducción

En esta práctica, se emplea el algoritmo A* para encontrar la ruta más corta en un laberinto representado como una matriz bidimensional, donde el valor de cada celda define si es transitable o no. El objetivo es implementar y analizar el rendimiento del algoritmo A* en dos escenarios: uno con un laberinto básico y otro con un laberinto más complejo. Además, se medirá el tiempo de ejecución del algoritmo en ambos casos para evaluar su eficiencia.

Marco Teórico

El algoritmo A* es una técnica de búsqueda ampliamente utilizada en inteligencia artificial para encontrar caminos óptimos en un grafo o en espacios de búsqueda bidimensionales, como los laberintos. Este algoritmo se basa en una función de evaluación, conocida como la función f(n), que calcula el coste estimado total de llegar al objetivo pasando por un nodo n. Esta función está compuesta por dos componentes principales: g(n), que representa el coste exacto del camino desde el nodo inicial hasta el nodo n, y h(n), que es una estimación heurística del coste restante desde n hasta el objetivo. La clave de la efectividad del algoritmo A* reside en la función heurística, ya que ésta guía al algoritmo hacia la solución, lo que permite reducir el número de nodos explorados.

El algoritmo A* es considerado tanto óptimo como completo, lo que significa que garantiza encontrar la solución más corta, siempre que exista una, y que explora el menor número de nodos posible para lograrlo. Esto se logra cuando la heurística utilizada es admisible, es decir, que nunca sobrestima el coste real para llegar al objetivo. En los problemas de búsqueda, como la navegación en laberintos, una heurística comúnmente empleada es la distancia Manhattan, que es apropiada cuando el movimiento está restringido a las direcciones cardinales (arriba, abajo, izquierda y derecha). Este tipo de heurística es simple y eficiente para los laberintos, ya que mide el número mínimo de pasos requeridos para llegar al objetivo sin tener en cuenta los obstáculos.

En la aplicación del algoritmo A* a los laberintos, la estructura del problema se modela como un grafo, donde cada celda del laberinto se considera un nodo y las celdas adyacentes que son transitables (es decir, no están bloqueadas por paredes) están conectadas por aristas. De este modo, el algoritmo busca el camino más corto desde una celda de inicio hasta una celda de destino, evitando los obstáculos representados por las paredes del laberinto. Los caminos posibles sólo pueden atravesar celdas que estén marcadas como transitables, lo que añade complejidad al problema de búsqueda en comparación con un espacio de búsqueda completamente abierto.

Desarrollo

En esta práctica implementamos el algoritmo A* para resolver el problema de encontrar un camino en un laberinto dado(4-puzzle). El laberinto está representado como una matriz, donde los "1" representan paredes y los "0" representan caminos libres. El objetivo es encontrar un camino desde una posición de inicio hasta una posición de salida. A* utiliza una combinación de búsqueda heurística y costo acumulado para priorizar los caminos más prometedores.

```python
# Representacion del laberinto
# 0 = camino, 1 = pared
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]
]

# Posición de inicio y salida
start = (0, 1)  # Coordenadas (fila, columna) de inicio
end = (3, 4)    # Coordenadas (fila, columna) de salida
```

Para poder realizar la solución propuesta se emplearon algunas estructuras de datos fundamentales como una cola de prioridad para gestionar los nodos abiertos, con la ayuda de '`heapq`'. Esta estructura permite insertar y extraer el nodo con la menor prioridad de manera eficiente, además de un diccionario '**`came_from`**' que almacena el nodo predecesor de cada nodo en el camino hacia el destino, y el diccionario '**`g_score`**' este almacena el costo del camino más corto encontrado hasta el momento desde el inicio hacia cada nodo.

```python
rows, cols = len(maze), len(maze[0])
open_list = []
heapq.heappush(open_list, (0 + heuristic(start, end), 0, start))
came_from = {}
g_score = {start: 0}
```

Para calcular la distancia entre 2 nodos hacemos uso de la heurística 'La distancia de Manhattan" **heuristic(a, b)** por ser adecuada para este tipo de problemas porque sólo permite movimientos ortogonales (arriba, abajo, izquierda, derecha). La distancia de Manhattan es la suma de las diferencias absolutas entre las coordenadas de los puntos. Esta heurística es apropiada para laberintos porque mide el número mínimo de pasos requeridos sin tener en cuenta obstáculos.

```python
# Función para calcular la heurística (distancia de Manhattan)
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

La implementación del algoritmo A* Combina la exploración basada en costos acumulados con una heurística para encontrar el camino más corto. La cual sigue los siguientes pasos:

- Inicializa la cola de prioridad con el nodo de inicio y su heurística.
- En cada iteración:
    1. Extrae el nodo con la menor prioridad.
    2. Si el nodo actual es el objetivo, reconstruye el camino.
    3. Para cada vecino (nodo adyacente):
        - Si el vecino es transitable y su nuevo costo es menor que el registrado, actualiza el puntaje y lo añade a la cola de prioridad.
- Si no hay más nodos por explorar, significa que no existe un camino.

```python
def astar(maze, start, end):
    rows, cols = len(maze), len(maze[0])
    open_list = []
    heapq.heappush(open_list, (0 + heuristic(start, end), 0, start))
    came_from = {}
    g_score = {start: 0}

    while open_list:
        _, current_cost, current = heapq.heappop(open_list)

        if current == end:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]  # Devuelve el camino desde el inicio hasta el final

        for direction in directions:
            neighbor = (current[0] + direction[0], current[1] + direction[1])

            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols and maze[neighbor[0]][neighbor[1]] == 0:
                new_cost = current_cost + 1

                if neighbor not in g_score or new_cost < g_score[neighbor]:
                    g_score[neighbor] = new_cost
                    priority = new_cost + heuristic(neighbor, end)
                    heapq.heappush(open_list, (priority, new_cost, neighbor))
                    came_from[neighbor] = current

    return None  # No hay camino
```

Adicionalmente, para esta práctica medimos el tiempo necesario para poder resolverlo, usando las funciones ‘time’ fue posible hacerlo de la siguiente manera:

```python
# Medir el tiempo de ejecución del primer escenario
start_time = time.time()
path = astar(maze, start, end)
end_time = time.time()
```

Finalmente para el primer ejemplo, imprimimos el camino encontrado, si es que existe un camino y el tiempo de ejecución:

```
# Resultado y tiempo de ejecución
if path:
    print("Camino encontrado:", path)
else:
    print("No se encontró un camino.")
print(f"Tiempo de ejecución: {end_time - start_time:.6f} segundos")
```

RESULTADO:

```
Camino encontrado: [(0, 1), (1, 1), (1, 2), (1, 3), (2, 3), (3, 3), (3, 4)]
Tiempo de ejecución: 0.000119 segundos
```

Para la segunda parte de este laboratorio se nos pidió realizar el mismo algoritmo para un laberinto más complejo y grande, por lo que se propuso un laberinto de tamaño 10x10 por lo que a medida que aumenta el tamaño del laberinto, el número de nodos a explorar crece exponencialmente, lo que incrementa el costo computacional. Esto hace que el tiempo de ejecución de A* depende significativamente de la complejidad del laberinto y la calidad de la heurística utilizada.

Siguiendo los mismos pasos del algoritmo y con el siguiente laberinto, estado inicial y final:

```
# Escenario adicional: laberinto más grande y complejo
maze_large = [
    [1, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 1, 0, 0, 0, 1, 1],
    [1, 1, 1, 0, 1, 0, 1, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 1, 1, 0, 1],
    [1, 1, 1, 1, 1, 0, 0, 1, 0, 1],
    [1, 1, 1, 1, 1, 1, 0, 1, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 1, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 0, 0],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
]
start_large = (0, 1)
end_large = (8, 9)
```

RESULTADO

```
Camino encontrado en el laberinto grande: [(0, 1), (1, 1), (1, 2), (1, 3), (2, 3), (3, 3),
                                           (3, 4), (3, 5), (2, 5), (1, 5), (1, 6), (1, 7),
                                           (2, 7), (2, 8), (3, 8), (4, 8), (5, 8), (6, 8),
                                           (7, 8), (8, 8), (8, 9)]
Tiempo de ejecuciOn: 0.000181 segundos
```

Se muestra todo el camino seguido para encontrar el nodo final desde el nodo inicial.

Conclusiones

Al término de esta práctica podemos realizar diversas conclusiones en referencia a los algoritmos heurísticos y su relación con la resolución de puzzles:

Los algoritmos heurísticos presentan una mejora significativa en tiempo y costos a comparación de otros algoritmos ya utilizados anteriormente como el de *búsqueda en profundidad (DFS)* y *búsqueda en anchura (BFS),* ya que estos se encargan de recorrer el árbol de nodos en un orden específico, lo que resulta en un algoritmo menos eficiente para problemas que presentan una gran cantidad de nodos y cuya solución se encuentra muy lejos del nodo raíz.

Por otro lado, el uso de algoritmos heurísticos para resolver este tipo de problemas ya que buscan la mejor solución mediante funciones heurísticas. En este caso, la función utilizada nos permite conocer el valor del camino más corto mediante la implementación del camino Manhattan ya que se centra en la suma de las diferencias entre cada par de coordenadas.

Tal como suponíamos al inicio de la práctica, la implementación de un laberinto más complejo en cuestión de cantidad de nodos supone un aumento exponencial de recursos y tiempo, aunque menor en comparación cons los algoritmos mencionados anteriormente, por lo que el uso de algoritmos heurísticos, como el de *A**, sigue siendo una opción viable y necesaria para resolver este tipo de problemas.

Referencias

- GeeksforGeeks. (2024, July 30). *A* search algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/a-search-algorithm/
- *Introduction to A**. (n.d.). https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html