Inteligencia Artificial

Nombre: Eduardo Echeverria (a1516)

1. ¿Cuáles son los PEAS de este problema? (Performance, Environment, Actuators, Sensors).

Performance: Para el problema de las Torres de Hanoi, el performance estaría expresado como la cantidad de pasos para mover los discos de la varilla origen a la varilla destino, donde se busca siempre que la cantidad de pasos sea la minima posible.

Environment: Para este problema el environment estaría conformado por las tres varillas más la cantidad de discos, que para este caso serian los 5 discos.

Actuators: En este caso los actuadores serían las funciones del Agente racional encargadas de mover los discos. Para el caso del código sobre el que se trabajó podemos analizar puntualmente las funciones de búsqueda (breadth_first_tree_search , breadth_first_graph_search, search_fifo, search_lifo) dado que estas funciones no solo buscan la solución sino que también ejecutan el movimiento de los discos a través de expresiones como pop(), insert(), append(), etc.

Sensors: En un caso mas general, los sensores del problema vendrían a ser las varillas. Sin embargo para el caso del código donde venimos trabajando los sensores vienen dados por la clase de StatesHanoi que dependiendo de si se instancian con los datos de initial_state o goal_state sirven para saber de donde se esta partiendo y a que estado se desea llegar.

2. ¿Cuáles son las propiedades del entorno de trabajo?

El entorno de trabajo correspondiente al problema de las torres de Hanoi tiene las siguientes propiedades:

Es un entorno **Totalmente Observable**, dado que los diferentes estados o pasos hacia la resolución del problema pueden ser observados en todo momento. Nosotros en todo momento tenemos visibilidad completa de que discos están en cual varilla.

Es también un entorno **Determinista** porque el siguiente estado o el siguiente paso hacia solución, depende enteramente del estado actual y de la acción del agente. Es decir que el siguiente paso con la nueva posición de los discos en las varillas depende únicamente de la posición actual y de la acción del agente acerca de que disco mover y hacia cual varilla.

También podemos mencionar que es un entorno de trabajo **Secuencial**, debido a que una decisión actual con respecto a mover un determinado disco hacia una varilla en particular, puede llegar a afectar el resto de las decisiones a futuro. Esto debido a que los nuevos movimientos van a estar condicionados a la posición actual de los discos.

Se trata también de un entorno **Estático**, debido a que mientras se esta tomando la decisión acerca del siguiente paso, el entorno permanece inalterado. Mientras el agente decide que disco mover, la posición actual de los discos no cambia.

Es también un entorno **Discreto** debido a que en cualquier caso, el numero de pasos hacia la solución es finito. Independientemente de la eficiencia del procedimiento que siga el agente, siempre hay numero finito de pasos hasta encontrar la solución.

Finalmente podemos decir que es un entorno de **Agente Individual**, debido a que es un solo agente el que se requiere para mover los discos entre las varillas.

3. En el contexto de este problema, establezca cuáles son los: estado, espacio de estados, árbol de búsqueda, nodo de búsqueda, objetivo, acción y frontera.

Estado: Cada una de las posibles posiciones de los discos en las varillas, correspondiente con cada uno de los posibles pasos camino a la solución.

Espacio de estados: El espacio de estados viene determinado por la relación 3^n, donde “n” corresponde a la cantidad de discos. Por tanto para 5 discos tenemos 243 posibles estados, para 64 discos hay 3,43x10^30 posibles estados.

Arbol de búsqueda: En el contexto del problema, el árbol de búsqueda correspondería al diagrama de todos los posibles estados hasta encontrar la solución. Partiendo del estado inicial, es decir con los 5 discos en una varilla origen (supongamos la varilla de la izquierda), se empezarían a dividir en dos posible nodos, dependiendo si el primer disco se movió a la varilla central o a la de la derecha. De ahi, cada movimiento de un disco producirá un nuevo nodo hasta llegar a la solución, es decir con los 5 discos migrados a la varilla destino.

Nodo de búsqueda: Corresponde a cada posible estado originado con el movimiento de un disco visto como parte del árbol de búsqueda.

Objetivos: Para el caso del problema de las torres de Hanoi, el objetivo consiste en mover los discos, tomemos como ejemplo 5 discos, de una varilla origen a una varilla destino, de la forma mas eficiente posible, es decir en la menor cantidad de pasos posible.

Acción: La acción a ejecutarse en el contexto del problema corresponde al movimiento de un disco que realiza el agente racional.

Frontera: La frontera en un árbol de búsqueda corresponde a un nodo que delimita dos regiones, aquella que fue explorada y la que no fue explorada aún. En el contexto del problema, el nodo frontera correspondería a la última posición determinada de los discos camino hacia la solución, pero que no corresponde a la solución del problema, sino a un estado intermedio. Antes de este nodo se encuentra el área explorada y los siguientes estados de posiciones de los discos corresponden al área no explorada.

4. Implemente algún método de búsqueda. Puedes elegir cualquiera menos búsqueda en anchura primero (el desarrollado en clase). Sos libre de elegir cualquiera de los vistos en clases, o inclusive buscar nuevos.

En mi caso, escogí el método de búsqueda en anchura pero tomando en cuenta colas LIFO, implementado en la siguiente función:

```python
def search_lifo(problem: hanoi_states.ProblemHanoi):
    node_root = NodeHanoi(problem.initial)
    lifo = [node_root]
    explored = []

    while len(lifo) > 0:
        node = lifo.pop()
        explored.append(node.state)

        if problem.goal_test(node.state):
            return node

        for child in node.expand(problem):
            if child.state not in explored:
                lifo.append(child)

    return None
```

5. ¿Qué complejidad en tiempo y memoria tiene el algoritmo elegido?

El algoritmo de búsqueda en anchura por colas LIFO en general tiene el siguiente comportamiento:

En términos de tiempo de ejecución, este varía entre 0,017s y 0,021s.

En cuestión de consumo de memoria, el algoritmo por colas LIFO varía entre 0,21MB y 0,22MB

6. A nivel implementación, ¿qué tiempo y memoria ocupa el algoritmo? (Se recomienda correr 10 veces y calcular promedio y desvío estándar de las métricas).

Luego de 10 ejecuciones obtenemos los siguientes resultados:

Promedio del tiempo: 0,02017s
Para la desviación estándar seguimos la siguiente ecuación:

$$\mathrm{DE} = \sqrt{\frac{\sum |x - \mu|^2}{N}}$$

La cual nos da una desviación estándar de 4,98 x10^-4

7. Si la solución óptima es $2^k - 1$ movimientos con *k* igual al número de discos. Qué tan lejos está la solución del algoritmo implementado de esta solución óptima (se recomienda correr al menos 10 veces y usar el promedio de trayecto usado).

Partiendo de un valor de k=5 tenemos que: (2^5) - 1 = 31 movimientos.

Ejecutando el código de búsqueda por anchura por colas LIFO, obtenemos que en promedio (10 ejecuciones) el numero de pasos es de 122:

<Node HanoiState: 5 4 3 2 1 | | >
<Node HanoiState: 5 4 3 2 | | 1>
<Node HanoiState: 5 4 3 2 | 1 | >
<Node HanoiState: 5 4 3 | 1 | 2>
<Node HanoiState: 5 4 3 | | 2 1>
<Node HanoiState: 5 4 3 1 | | 2>
<Node HanoiState: 5 4 3 1 | 2 | >
<Node HanoiState: 5 4 3 | 2 | 1>
<Node HanoiState: 5 4 3 | 2 1 | >
<Node HanoiState: 5 4 | 2 1 | 3>
<Node HanoiState: 5 4 | 2 | 3 1>
<Node HanoiState: 5 4 1 | 2 | 3>
<Node HanoiState: 5 4 1 | | 3 2>
<Node HanoiState: 5 4 | | 3 2 1>
<Node HanoiState: 5 4 | 1 | 3 2>
<Node HanoiState: 5 4 2 | 1 | 3>
<Node HanoiState: 5 4 2 | | 3 1>
<Node HanoiState: 5 4 2 1 | | 3>

<Node HanoiState: 5 4 2 1 | 3 | >
<Node HanoiState: 5 4 2 | 3 | 1>
<Node HanoiState: 5 4 2 | 3 1 | >
<Node HanoiState: 5 4 | 3 1 | 2>
<Node HanoiState: 5 4 | 3 | 2 1>
<Node HanoiState: 5 4 1 | 3 | 2>
<Node HanoiState: 5 4 1 | 3 2 | >
<Node HanoiState: 5 4 | 3 2 | 1>
<Node HanoiState: 5 4 | 3 2 1 | >
<Node HanoiState: 5 | 3 2 1 | 4>
<Node HanoiState: 5 | 3 2 | 4 1>
<Node HanoiState: 5 1 | 3 2 | 4>
<Node HanoiState: 5 1 | 3 | 4 2>
<Node HanoiState: 5 | 3 | 4 2 1>
<Node HanoiState: 5 | 3 1 | 4 2>
<Node HanoiState: 5 2 | 3 1 | 4>
<Node HanoiState: 5 2 | 3 | 4 1>
<Node HanoiState: 5 2 1 | 3 | 4>
<Node HanoiState: 5 2 1 | | 4 3>
<Node HanoiState: 5 2 | | 4 3 1>
<Node HanoiState: 5 2 | 1 | 4 3>
<Node HanoiState: 5 | 1 | 4 3 2>
<Node HanoiState: 5 | | 4 3 2 1>
<Node HanoiState: 5 1 | | 4 3 2>
<Node HanoiState: 5 1 | 2 | 4 3>
<Node HanoiState: 5 | 2 | 4 3 1>
<Node HanoiState: 5 | 2 1 | 4 3>
<Node HanoiState: 5 3 | 2 1 | 4>
<Node HanoiState: 5 3 | 2 | 4 1>
<Node HanoiState: 5 3 1 | 2 | 4>
<Node HanoiState: 5 3 1 | | 4 2>
<Node HanoiState: 5 3 | | 4 2 1>
<Node HanoiState: 5 3 | 1 | 4 2>
<Node HanoiState: 5 3 2 | 1 | 4>
<Node HanoiState: 5 3 2 | | 4 1>
<Node HanoiState: 5 3 2 1 | | 4>
<Node HanoiState: 5 3 2 1 | 4 | >
<Node HanoiState: 5 3 2 | 4 | 1>
<Node HanoiState: 5 3 2 | 4 1 | >
<Node HanoiState: 5 3 | 4 1 | 2>
<Node HanoiState: 5 3 | 4 | 2 1>
<Node HanoiState: 5 3 1 | 4 | 2>
<Node HanoiState: 5 3 1 | 4 2 | >
<Node HanoiState: 5 3 | 4 2 | 1>
<Node HanoiState: 5 3 | 4 2 1 | >
<Node HanoiState: 5 | 4 2 1 | 3>

<Node HanoiState: 5 | 4 2 | 3 1>
<Node HanoiState: 5 1 | 4 2 | 3>
<Node HanoiState: 5 1 | 4 | 3 2>
<Node HanoiState: 5 | 4 | 3 2 1>
<Node HanoiState: 5 | 4 1 | 3 2>
<Node HanoiState: 5 2 | 4 1 | 3>
<Node HanoiState: 5 2 | 4 | 3 1>
<Node HanoiState: 5 2 1 | 4 | 3>
<Node HanoiState: 5 2 1 | 4 3 | >
<Node HanoiState: 5 2 | 4 3 | 1>
<Node HanoiState: 5 2 | 4 3 1 | >
<Node HanoiState: 5 | 4 3 1 | 2>
<Node HanoiState: 5 | 4 3 | 2 1>
<Node HanoiState: 5 1 | 4 3 | 2>
<Node HanoiState: 5 1 | 4 3 2 | >
<Node HanoiState: 5 | 4 3 2 | 1>
<Node HanoiState: 5 | 4 3 2 1 | >
<Node HanoiState: | 4 3 2 1 | 5>
<Node HanoiState: | 4 3 2 | 5 1>
<Node HanoiState: 1 | 4 3 2 | 5>
<Node HanoiState: 1 | 4 3 | 5 2>
<Node HanoiState: | 4 3 | 5 2 1>
<Node HanoiState: | 4 3 1 | 5 2>
<Node HanoiState: 2 | 4 3 1 | 5>
<Node HanoiState: 2 | 4 3 | 5 1>
<Node HanoiState: 2 1 | 4 3 | 5>
<Node HanoiState: 2 1 | 4 | 5 3>
<Node HanoiState: 2 | 4 | 5 3 1>
<Node HanoiState: 2 | 4 1 | 5 3>
<Node HanoiState: | 4 1 | 5 3 2>
<Node HanoiState: | 4 | 5 3 2 1>
<Node HanoiState: 1 | 4 | 5 3 2>
<Node HanoiState: 1 | 4 2 | 5 3>
<Node HanoiState: | 4 2 | 5 3 1>
<Node HanoiState: | 4 2 1 | 5 3>
<Node HanoiState: 3 | 4 2 1 | 5>
<Node HanoiState: 3 | 4 2 | 5 1>
<Node HanoiState: 3 1 | 4 2 | 5>
<Node HanoiState: 3 1 | 4 | 5 2>
<Node HanoiState: 3 | 4 | 5 2 1>
<Node HanoiState: 3 | 4 1 | 5 2>
<Node HanoiState: 3 2 | 4 1 | 5>
<Node HanoiState: 3 2 | 4 | 5 1>
<Node HanoiState: 3 2 1 | 4 | 5>
<Node HanoiState: 3 2 1 | | 5 4>
<Node HanoiState: 3 2 | | 5 4 1>

<Node HanoiState: 3 2 | 1 | 5 4>
<Node HanoiState: 3 | 1 | 5 4 2>
<Node HanoiState: 3 | | 5 4 2 1>
<Node HanoiState: 3 1 | | 5 4 2>
<Node HanoiState: 3 1 | 2 | 5 4>
<Node HanoiState: 3 | 2 | 5 4 1>
<Node HanoiState: 3 | 2 1 | 5 4>
<Node HanoiState: | 2 1 | 5 4 3>
<Node HanoiState: | 2 | 5 4 3 1>
<Node HanoiState: 1 | 2 | 5 4 3>
<Node HanoiState: 1 | | 5 4 3 2>
<Node HanoiState: | | 5 4 3 2 1>

Por tanto con el algoritmo de búsqueda por anchura por colas LIFO estamos a 91 pasos de la solución optima. Podemos concluir que si bien esta función de búsqueda es eficiente en términos de tiempo de ejecución y manejo de memoria, es altamente ineficiente al momento de buscar la solución en la menor cantidad de pasos.

El código con la solución implementada se encuentra en el repositorio:

https://github.com/eduardo-echeverria/entregas-ai/tree/main/hanoi_tower