Introducción a la Inteligencia Artificial - Trabajo Práctico 1

Diego Braga

Para responder las preguntas iniciales se toma en cuenta el problema en el mundo real de las torres de Hanoi en el cual un agente robótico intentará resolverlo.

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

Performance: En este problema se asume que todos los movimientos tienen el mismo costo, por lo que una medida simple de performance es contar la cantidad de movimientos necesarios para llegar a la solución partiendo desde el estado inicial.

Environment: El entorno se compone por el conjunto de discos y las tres varillas.

Actuators: Brazo capaz llevar un disco de una varilla a otra.

Sensors: Un sensor que determina el estado inicial de los discos en las varillas. Aquí se asume que el entorno es estático en todo momento y que ninguna acción externa puede cambiar la posición de los discos y las varillas.

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

Totalmente observable: El agente tiene conocimiento del estado de todos los discos en las varillas.

Determinista: Dada una disposición de los discos en las varillas y un movimiento (acción), se conoce claramente el estado siguiente.

Secuencial: Al cambiar realizar un movimiento se obtendrá una nueva disposición de los discos que afectará a los movimientos siguientes.

Estático: El estado de los discos y las varillas no cambia mientras el agente está determinando el próximo movimiento.

Discreto: Es posible enumerar todas las posibles posiciones de los discos en las varillas. Hay 3^n estados posibles.

Agente individual: En el entorno planteado no hay otros objetos que puedan ser tratados como agentes, es decir, que intenten maximizar su propia medida de performance.

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: Es una disposición particular válida de los discos en las varillas. Se entiende como válido que cada disco debe tener asignada una varilla y ordenados por tamaño de forma descendente (desde abajo hacia arriba).

Espacio de estados: Es el conjunto de todos los posibles estados mencionados en el punto anterior. Está compuesto por 3^n estados posibles (cada disco tiene tres varillas posibles para elegir).

Árbol de búsqueda: En el árbol de búsqueda se representan todos los posibles caminos desde el estado inicial realizando acciones válidas. En este caso el árbol es infinito por lo que es necesario tenerlo en cuenta para que el algoritmo a implementar no entre en ciclos innecesarios. El árbol es infinito porque dada una acción o una serie de acciones realizada sobre los discos, es posible revertirlas hasta volver al estado inicial y así comenzar de nuevo.

Nodo de búsqueda: Independientemente de la implementación particular, cada nodo en el árbol debería contener información sobre su estado, un puntero al nodo padre y el costo para llegar al mismo (total de movimientos realizados).

Objetivo: El objetivo es llevar todos los discos desde una varilla a otra respetando las reglas definidas.

Acción: Es un movimiento válido de un disco desde una varilla a otra. Ej. si partimos del estado inicial con 5 discos, una acción válida sería mover el disco más chico a una de las varillas vacías.

Frontera: Son todos los estados que han sido generados pero aún no explorados. Determina una separación entre los nodos explorados y los que aún no se conocen. En este caso dependerá de la implementación del algoritmo. Por ejemplo, para la búsqueda en anchura se mantiene la frontera en una cola FIFO, de forma de ir recorriendo los nodos del nivel actual antes de pasar a los hijos.

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.

A los efectos de resolver el ejercicio se implementó el algoritmo de búsqueda en profundidad primero (*depth first graph search*). Se utilizaron las estructuras dadas en clase y se creó una nueva función en el archivo *search.py* de nombre *depth_first_graph_search*.

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

El algoritmo tiene una complejidad en tiempo de $O(b^d)$ donde *b* es la cantidad de acciones disponibles dado un estado y *d* es la profundidad máxima a la que se puede llegar.

La profundidad máxima *d* está en el rango $\left[2^n - 1,\ 3^n\right]$, siendo *n* la cantidad de discos. El extremo inferior del intervalo corresponde a la cantidad óptima de movimientos mientras que el superior la cantidad máxima de estados. Este máximo es muy poco probable que se alcance a medida que *n* crece, debido a las limitaciones planteadas en las reglas del juego y los propios controles del algoritmo para los nodos explorados. Por este mismo motivo, también pueden existir caminos con largo menor a $2^n - 1$.

Por su parte, *b* está acotada en el rango [2, 3] que son las acciones permitidas dado un estado cualquiera del juego. Este rango es posible determinarlo analizando los diferentes casos:

- Cuando hay dos varillas vacías, hay sólo 2 movimientos posibles: mover el disco más chico a cualquiera de estas dos.
- Cuando hay una única varilla vacía, hay 3 movimientos posibles: mover el disco más chico a la varilla vacía o sobre el disco más grande, o mover el disco más grande sobre la varilla vacía.
- Cuando están las tres varillas con discos, también hay 3 movimientos posibles: mover el disco más chico sobre cualquiera de los dos discos más grandes, o mover el disco de tamaño medio sobre el disco más grande

Para analizar la complejidad de la memoria se deben analizar por separado las dos estructuras principales: la cola *frontier* (LIFO) y la lista *explored*. La complejidad de la cola *frontier* es $O(bd)$, ya que para cada nodo del camino en profundidad se van guardando sus hijos en la cola. Por su parte la lista *explored* tiene una complejidad $O(b^d)$, ya que en el peor caso se almacenan todos los posibles estados antes de encontrar la solución. Nuevamente, debido a los controles realizados sobre los estados explorados no se va a

llegar a esa cantidad de nodos. Por lo tanto, dadas estas aclaraciones y tomando el término con mayor orden, la complejidad en memoria queda $O(b^d)$.

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).

En la siguiente tabla se muestran los tiempos de ejecución y memoria utilizada para el problema planteado de 5 discos en 10 ejecuciones.

Ejecución	Tiempo (ms)	Memoria (MB)
1	40,71	0,16
2	39,16	0,16
3	57,73	0,16
4	42,46	0,16
5	44,82	0,16
6	51,95	0,16
7	39,15	0,16
8	45,58	0,16
9	39,69	0,16
10	49,98	0,16

En cuanto al tiempo de ejecución se obtuvo una media de 45,12 ms con una desviación estándar de 6,30 ms. En el caso de la memoria se obtuvo siempre el mismo valor, por lo que la media es 0,16 MB y su desviación estándar es 0,00 MB.

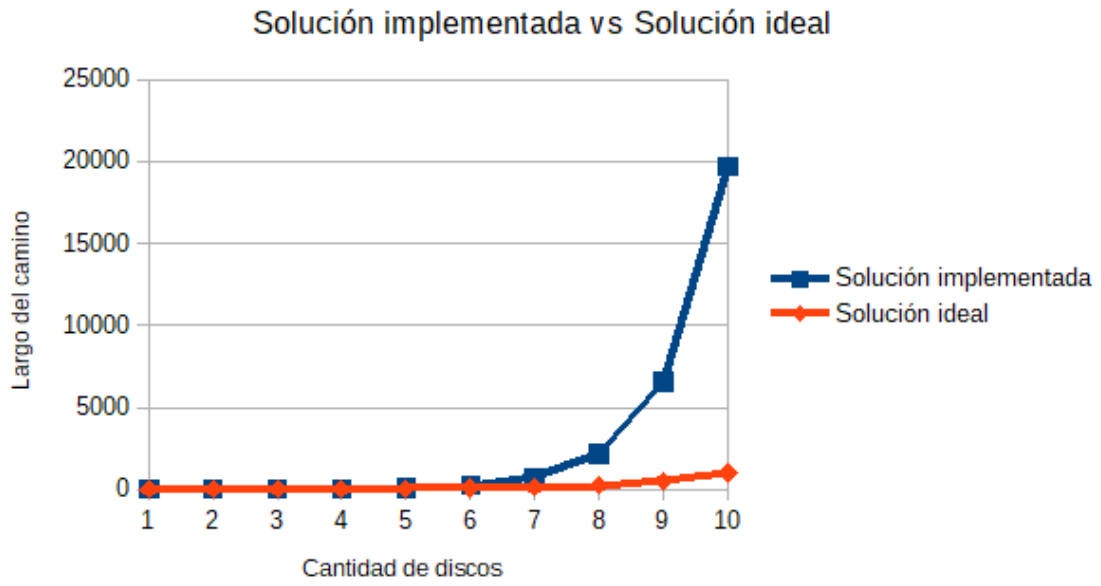
A modo de comparación, se probó el algoritmo *breadth first graph search* que estaba implementado en el archivo *search.py* en 10 corridas. Se obtuvo un tiempo de ejecución de 136,7 ms con una desviación estándar de 9,9 ms y un consumo de memoria de 0,21 MB con desviación estándar de 0,00 MB.

Esto confirma lo visto teóricamente en clase acerca de la performance en tiempo y memoria de ambos algoritmos. *Depth first graph search* es muy rápido y hace un uso bastante eficiente de la memoria en comparación con *breadth first graph search*, lo cual es una muy buena opción si sólo buscamos solucionar el problema de forma rápida.

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).

El algoritmo implementado es determinista ya que devuelve siempre el mismo resultado dado un estado inicial. De todas formas, a continuación se presenta una tabla que muestra la cantidad de movimientos hasta 10 discos y la diferencia comparada a la solución ideal.

Discos	Solución ideal (2^k - 1 movimientos)	Solución implementada	Diferencia
1	1	1	0
2	3	5	2
3	7	9	2
4	15	29	14
5	31	81	50
6	63	245	182
7	127	729	602
8	255	2189	1934
9	511	6561	6050
10	1023	19685	18662



Es posible observar que aunque ambas soluciones presentan una cantidad de movimientos que crece exponencialmente a medida que se aumenta la cantidad de

discos, la del algoritmo implementado lo hace en un orden mayor. A pesar de esto, es muy rápido en comparación con otros algoritmos como por ejemplo búsqueda en anchura o Dijkstra, e implica un consumo muy bajo de memoria.

Breve análisis de otro algoritmo

A modo experimental se implementó una variación del algoritmo *deep-limited deepening search* el cual recibe por parámetro la profundidad máxima en la que se buscará la solución. Se tomó como base el presentado en clase pero se cambió la estructura de la variable *reached* por un stack para que se pudiera ir almacenando el camino desde el nodo inicial hasta el nodo actual para el único fin de evitar ciclos (como se muestra en el pseudocódigo del libro[1]).

Esta implementación evalúa la función objetivo sólo en la profundidad recibida y no en estados intermedios, ya que para el problema de las Torres de Hanoi se conoce de antemano el costo de la solución óptima.

Tiene una complejidad en tiempo $O(b^d)$ al igual que *depth first graph search* pero con la diferencia de que aquí no se realiza control de nodos ya explorados, lo cual degrada mucho la performance.

En cuanto a memoria, debemos considerar la complejidad de la estructura reached como también que el algoritmo corre en forma recursiva, lo cual tiene un impacto en la memoria stack. Por el lado de de la variable *reached* tenemos una complejidad de $O(d)$ ya que como máximo se almacenan los nodos hasta la profundidad del árbol d. Sin embargo, debido a la recursividad hay que considerar que para cada nodo intermedio (no hoja) estamos obteniendo sus hijos, que quedarán en la memoria stack para cada invocación a la función *recursive_dls*. Esto implica una complejidad $O(bd)$, donde *b* es la cantidad media de acciones disponibles para un estado y *d* la profundidad del árbol.

Esta implementación se encuentra también en el archivo *search.py* con el nombre *depth_limited_search*. En base a ésta se probó también el algoritmo *iterative deepening search*, el cual no tiene mucho sentido en este caso ya que conocemos a priori el largo del camino a la solución.

Por fuera de este trabajo se probarán otros algoritmos mostrados más recientemente en clase, que tal vez hubiesen arrojado resultados aún más interesantes.

[1] *Peter Norvig y Stuart J. Russel, Artificial Intelligence - A Modern Approach Fourth Edition*