15-11-2020

PRÁCTICA 1

Búsqueda no informada y heurística en el espacio de estados

Ángel Ortega Alfaro & Verónica Giraldo Garrido
ANGEL.ORTEGA@ALU.UCLM.ES & VERONICA.GIRALDO1@ALU.UCLM.ES

ÍNDEX

Introducción

En esta práctica nuestro objetivo es abordar el problema de ajedrez usando agentes que planifican (las piezas de tablero), en particular, implementando estrategias de búsqueda no-informada e informada para planificar la secuencia de acciones que posteriormente será ejecutada por el agente para alcanzar el objetivo. Dicho objetivo consiste en, desde la posición de salida, recorrer el tablero mediante los movimientos correspondientes de cada una de las piezas hasta alcanzar el otro lado.

El entorno de nuestro problema consistirá en un tablero de ajedrez, generalmente con un tamaño de 8x8 (las dimensiones son variables), sobre el cual las piezas se generarán de manera aleatoria, y dependiendo de la semilla que especifiquemos como parámetro a la hora de ejecutar el código y crear el tablero.

En la representación, se han numerado las filas y columnas de 0 a 7 (o de 0 a n-1 para cualquier otro caso). Las piezas negras se llaman con letras minúsculas y se mueven de abajo a arriba, y las blancas se llaman con mayúsculas y se mueven de arriba a abajo. En los diferentes tipos de piezas encontramos al peón (P), rey (K), reina (Q), torre (R), alfil (B) y caballo (N), nombradas mediante la notación en inglés.

```
   00 01 02 03 04 05 06 07
-------------------------
00|  |  |  |  | |*Q|  |  |
-------------------------
01|  |  |  |  |  | p|  | p|
-------------------------
02|  | p| P| r|  |  | p|  |
-------------------------
03|  |  |  |  |  |  | q|  |
-------------------------
04|  |  |  |  |  |  |  |  |
-------------------------
05|  |  |  | p| P|  |  | P|
-------------------------
06|  |  | k|  |  | P| P|  |
-------------------------
07|  |  |  | R|  | K|  |  |
-------------------------
```

Imagen 1: Instancia del problema y representación

Jugaremos con las piezas blancas, marcadas por un * para que sean fácil de identificar. Nuestro agente estará inicialmente en la fila 0 del tablero con independencia del valor de la columna. Como hemos mencionado antes, el objetivo del agente es llegar hasta la última fila del tablero, sin importar qué columna le corresponda. Para ello sólo puede usar los movimientos legales del tipo de pieza que represente, incluidos los de capturar oponentes.

Finalmente, durante la ejecución de nuestros algoritmos, asociaremos un coste a cada movimiento realizado. El coste es la máxima distancia (en celdas) recorrida en horizontal o vertical más 1. El 1 que se añade es para que sea siempre mejor hacer un movimiento de distancia 4 que 4 movimientos de distancia 1. La ecuación es la siguiente:

$$\boldsymbol{coste}\ ((\boldsymbol{f_1}, \boldsymbol{c_1}) \rightarrow (\boldsymbol{f_2}, \boldsymbol{c_2})) = \mathbf{max}\ (|\boldsymbol{f_1}, \boldsymbol{f_2}|, |\boldsymbol{c_1}, \boldsymbol{c_2}|) + \mathbf{1}$$

Descripción de la implementación

State, Position y Action

- **State.** State contiene como miembros la matriz bidimensional que representa el tablero y la posición en que se encuentra nuestro agente. Implementa el método *isFinal()*, que devuelve true si el estado actual es final (la posición de nuestro agente está en la última fila del tablero); el método *copy()*, que devuelve una copia idéntica del estado actual sin modificarlo; y el método *applyAction(Action action)*, que devuelve el estado obtenido al aplicar la acción pasada sobre el actual. El estado actual no se modifica puesto que se aplica la acción sobre una copia haciendo uso del método *copy()*. Además, en esta clase también se incluye la función sobreescrita *hashCode()*, proveniente de *Object()*, con la que se sacará la función hash de dos estados iguales.
    - Cabe destacar que deberíamos guardar el estado de todo el tablero para las comparaciones entre piezas; sin embargo, puesto que eso consumiría mucha memoria, usando simplemente las posiciones *x* e *y* de las piezas sería más eficiente porque se podaría gran parte del árbol generado.
- **Position.** La clase Position simplemente representa una celda de nuestro tablero mediante la fila (row) y columna (col) correspondiente.
- **Action.** La clase Action representa la posición inicial y la posición final del agente tras ejecutarse la acción. Además, se implementa el método *getCost()*, que nos devuelve el coste de la acción aplicada según la expresión anterior. También nos encontramos con un *toString()*, que imprime la cadena de posiciones iniciales y finales de las piezas para comprobar el correcto funcionamiento de cada uno de los algoritmos.

Piece y subclases

La clase **Piece** define las variables básicas de *color* y *tipo* para cada pieza, e implementa tantos métodos como movimientos pueden realizar las distintas piezas que representan los elementos del tablero de ajedrez. Cada subclase representa una pieza. A excepción del peón (tiene sus movimientos definidos en su propia clase), todas las demás piezas están formadas por un *ArrayList*, al cual se le añaden todos sus posibles movimientos dependiendo de la posición en la que se encuentre la pieza.

SearchClass y algoritmos

La clase **SearchClass** se encarga de agrupar la llamada a cada algoritmo. Se le pasa como argumento un número del 1 al 5 (representa todos los algoritmos implementados), dos

semillas de tal forma que generen una posición inicial y una situación de piezas aleatoria, y otro número que representa qué tipo de pieza queremos colocar; se llama a un switch en el que, dependiendo del valor del primer parámetro, se genera un algoritmo u otro con los datos pasados como argumentos.

En cuanto a las clases que representan los algoritmos, tenemos una clase que resuelve el tablero mediante el algoritmo por **búsqueda en anchura**, otra clase que lo resuelve mediante el algoritmo por **búsqueda en profundidad**, otra clase que resuelve mediante el algoritmo por **búsqueda en coste uniforme**, otra clase más que resuelve mediante el algoritmo de **búsqueda en primero-mejor**, y por último tenemos una clase que resuelve el algoritmo mediante **búsqueda en A***. Serían cinco clases en total, que corresponden con el primer parámetro numérico que le pasamos a **SearchClass**. Todas estas clases incluyen una función **sucesores** la cual genera los nodos hijos con los posibles movimientos desde la posición donde se llama a dicha función. Se crea un un ArrayList con las posibles acciones, y en un bucle for, se genera un nodo por cada acción posible, estableciendo el nodo padre, la acción a realizar, el coste, la profundidad, etc. Además de actualizar la variable que indica los nodos generados y expandidos.

Búsqueda en anchura

La búsqueda en anchura se caracterizar por utilizar una estructura de datos conocida como *Queue*, que consiste en una lista FIFO la cual coge el nodo más antiguo en esa lista. Además, haciendo uso de hashset, nos aseguramos que el nodo que estamos cogiendo no se vaya a una posición por la que ya hemos pasado.

Búsqueda en profundidad

La búsqueda en profundidad se caracteriza por utilizar una estructura de datos conocida como *Stack,* que consiste en una lista LIFO, la cual coge el nodo más nuevo en esa lista. De forma análoga al algoritmo anterior, hacemos uso del hashset para no repetir estados.

Búsqueda en coste uniforme

La búsqueda en coste uniforme se caracteriza por utilizar una estructura de datos conocida como *PriorityQueue*, la cual ordena la lista según un comparador. En este caso, en la clase nodo hay definido un comparador que ordena la lista de menor a mayor según el coste del nodo. De forma análoga al algoritmo anterior, hacemos uso del hashset para no repetir estados.

Búsqueda en primero-mejor

De forma análoga al algoritmo anterior, la búsqueda en primero-mejor se caracteriza por utilizar una estructura de datos conocida como *PriorityQueue*, la cual ordena la lista según un comparador. En este caso, en la clase nodo hay definido un comparador que ordena la lista de menor a mayor según la heurística del nodo. De forma análoga al algoritmo anterior, hacemos uso del hashset para no repetir estados.

Búsqueda en A*

De forma análoga al algoritmo anterior, la búsqueda en primero-mejor se caracteriza por utilizar una estructura de datos conocida como *PriorityQueue*, la cual ordena la lista según un comparador. En este caso, en la clase nodo hay definido un comparador que ordena la lista de menor a mayor según la función heurística del nodo. De forma análoga al algoritmo anterior, hacemos uso del hashset para no repetir estados.

Clase Nodo

Esta clase genera el tipo de objeto que incluiremos en las variables que representan a los sucesores, la frontera, los nodos ya visitados o cerrados… Consta de un estado, de un coste, una heurística, un nivel de profundidad, y también se indica en esta clase cuál es el nodo padre y la acción que ha generado el nodo que se esté tratando. Aparte de eso, encontramos un par de comparadores, *porCoste* y *porHeurística*, que nos servirán para comparar los datos de la *PriorityQueue* de la frontera y poder aplicar a continuación los algoritmos pertinentes: se tomará siempre el individuo de la cabeza de la lista, y ese individuo será el correcto cuando se deba expandir el nodo.

Comparación del rendimiento

Para poder realizar algunas comparaciones de rendimiento en cada uno de los distintos algoritmos, vamos a ejecutarlos de acuerdo a los escenarios siguientes: longitud, coste, nodos expandidos, nodos generados y tiempo de búsqueda/ejecución del problema.

En general, tenemos un tablero de dimensiones 8x8, con una densidad de piezas de 0.5; torre, alfil y caballo blancos como agentes, y una semilla fija con valor de 1. La otra semilla, que será variable, alterna sus valores entre 1 y 10 paulatinamente, así como se ve en las tablas adjuntas del documento Excel.

Todas las variables que aparecen en dichas tablas serán comparadas entre sí en todos los escenarios establecidos. Así pues, nos encontramos con los siguientes gráficos:

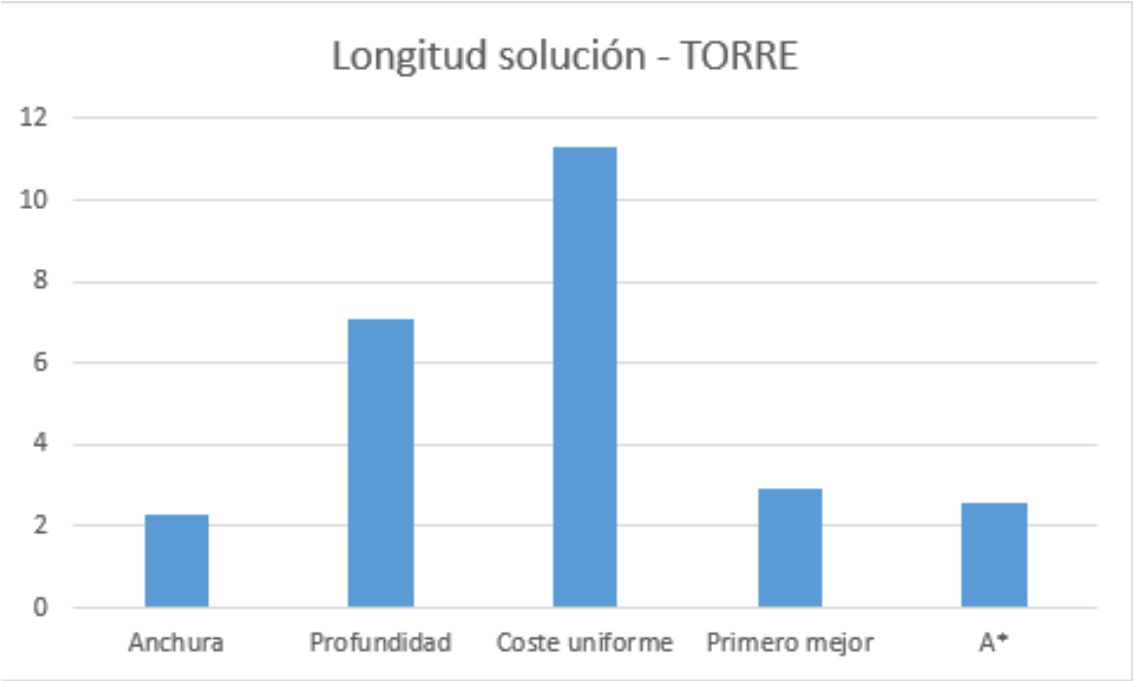

Imagen 2: Gráfico sobre longitud del problema con la torre como agente

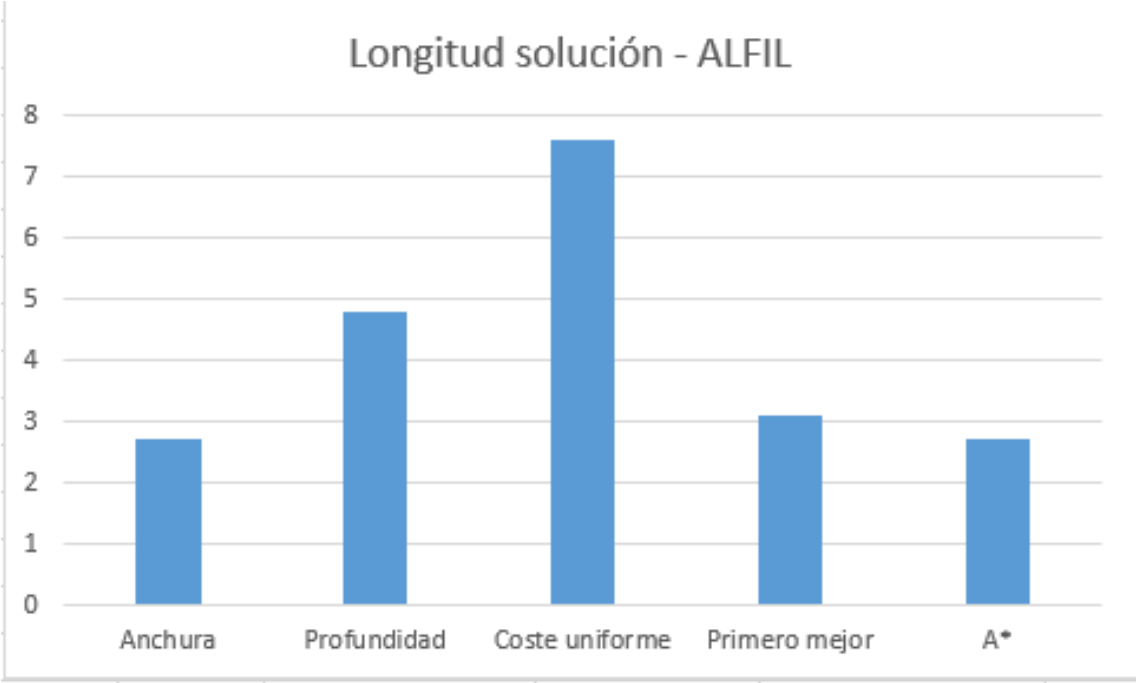

Imagen 3: Gráfico sobre longitud del problema con el alfil como agente

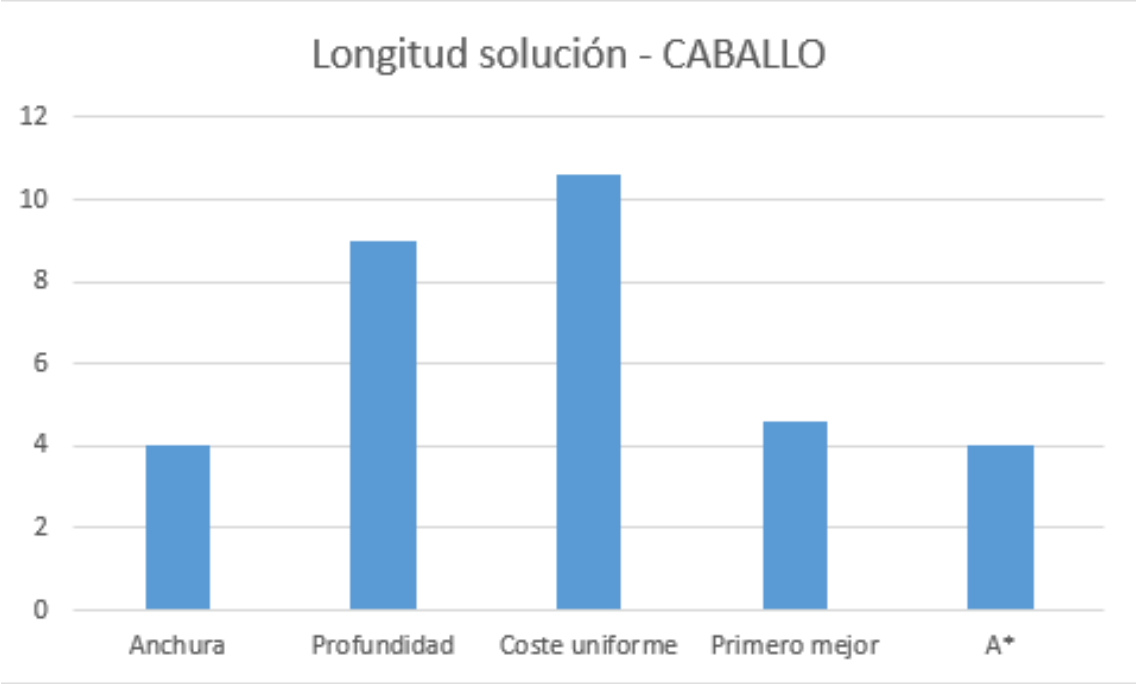

Imagen 4: Gráfico sobre longitud del problema con el caballo como agente

En estos gráficos, se muestra claramente que los resultados obtenidos son muy similares entre las piezas. Sin embargo, a causa del tipo y número de movimientos de cada una de las piezas, podemos observar que algunos algoritmos como **anchura**, **primero mejor** y **A*** tienen una longitud menor.

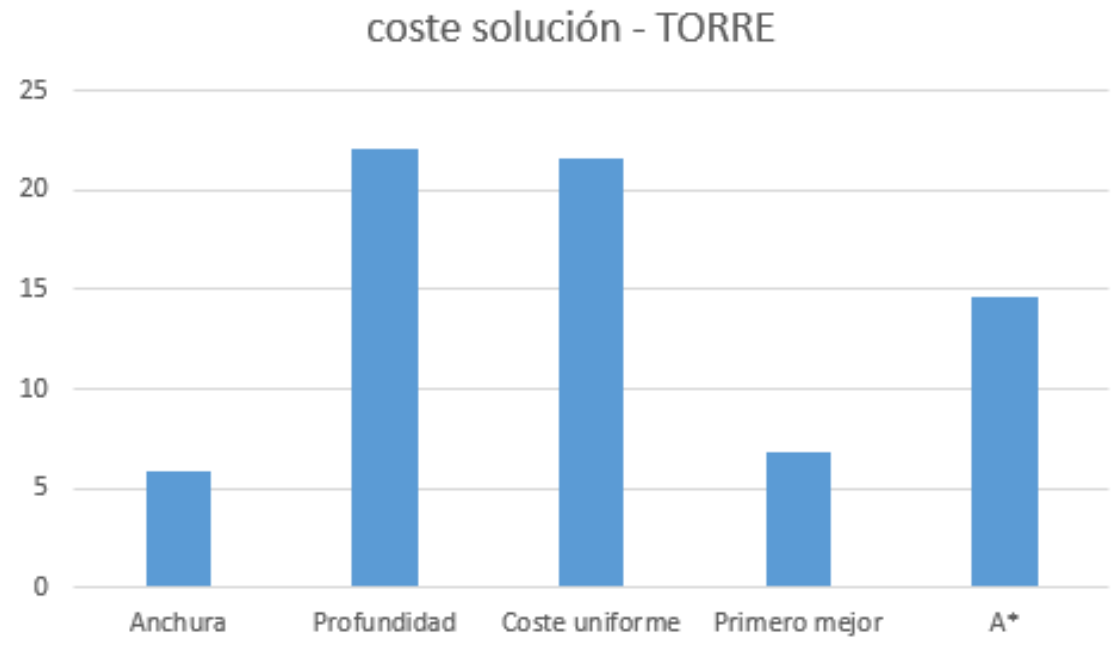

Imagen 5: Gráfico sobre coste del problema con la torre como agente

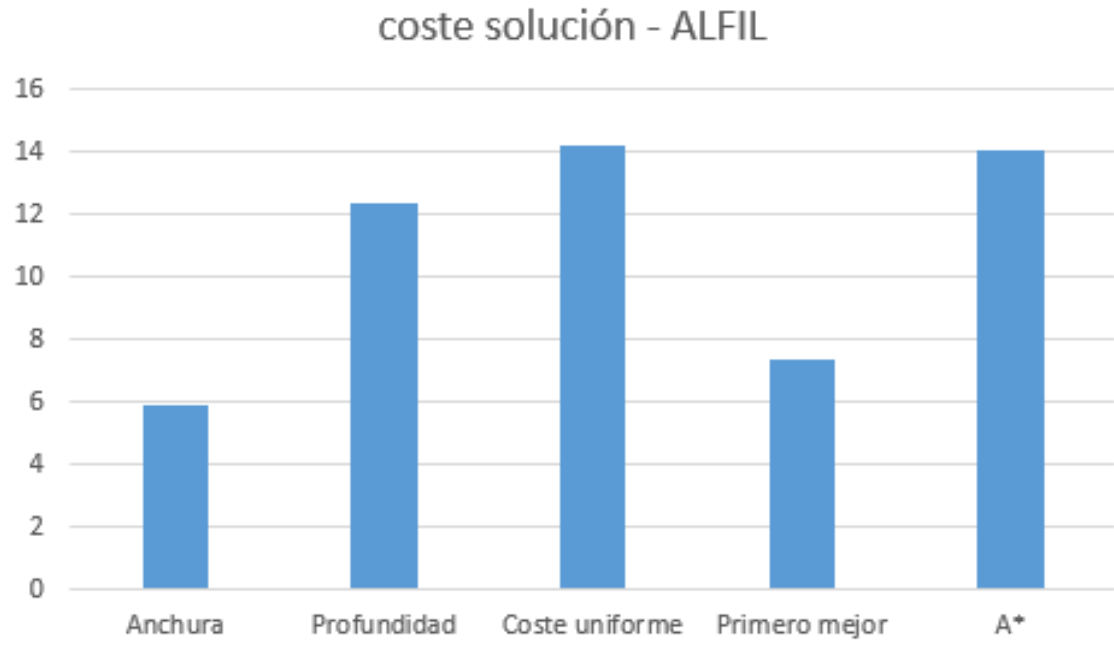

Imagen 6: Gráfico sobre coste del problema con el alfil como agente

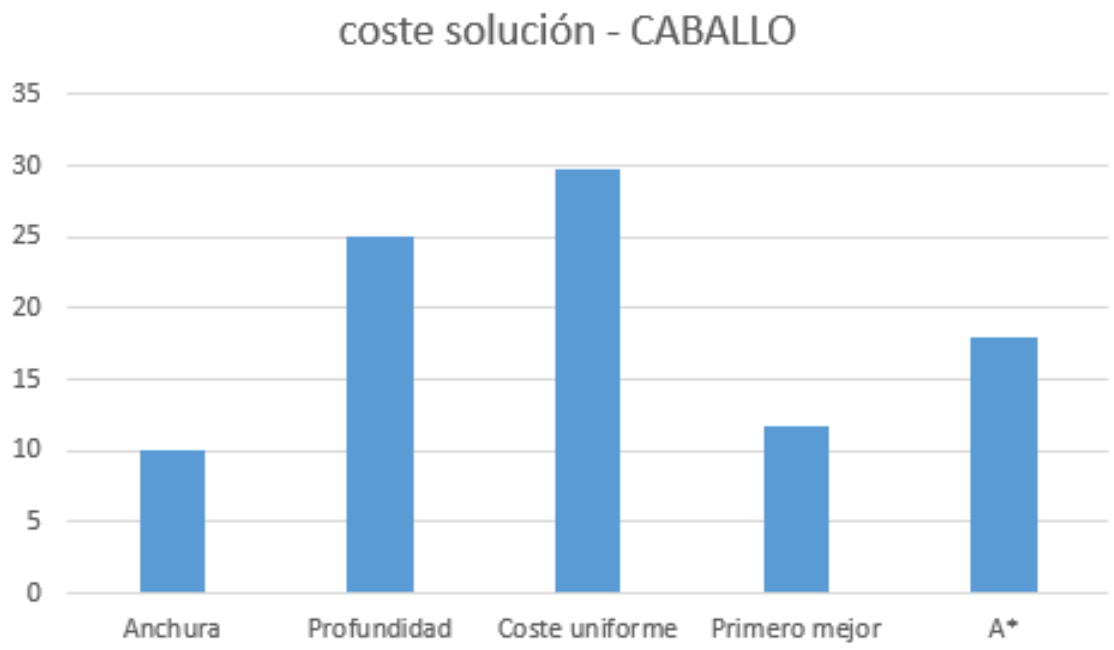

Imagen 7: Gráfico sobre coste del problema con el caballo como agente

Nuevamente, entre estas tres gráficas relativas al coste del problema, no se aprecia una diferencia muy clara. Los resultados son uniformes, aunque para el quinto algoritmo los resultados se han agrandado bastante para el alfil. Esto puede haber sido causado por los movimientos de dicha pieza, condicionados también por la existencia de otras piezas (tanto blancas como negras que se podrá comer) repartidas en el resto del tablero. Pero, tal como se ve, podemos decir que los algoritmos **anchura** y **primero mejor** son los más óptimos en cuanto a coste de la solución.

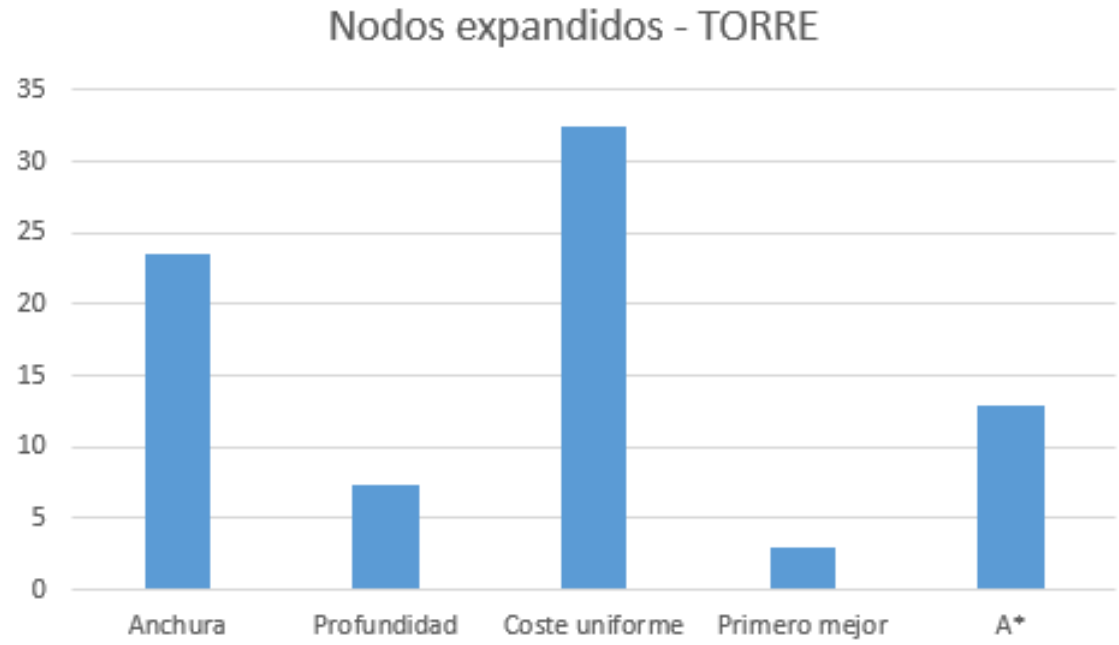

Imagen 8: Gráfico sobre los nodos expandidos con la torre como agente

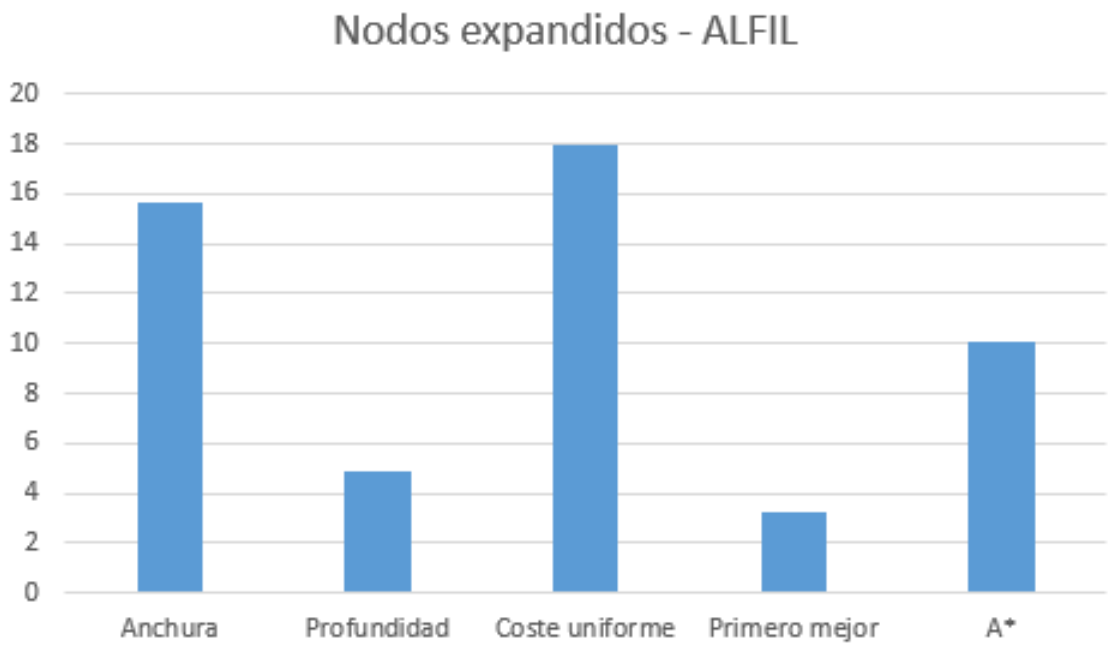

Imagen 9: Gráfico sobre los nodos expandidos con el alfil como agente

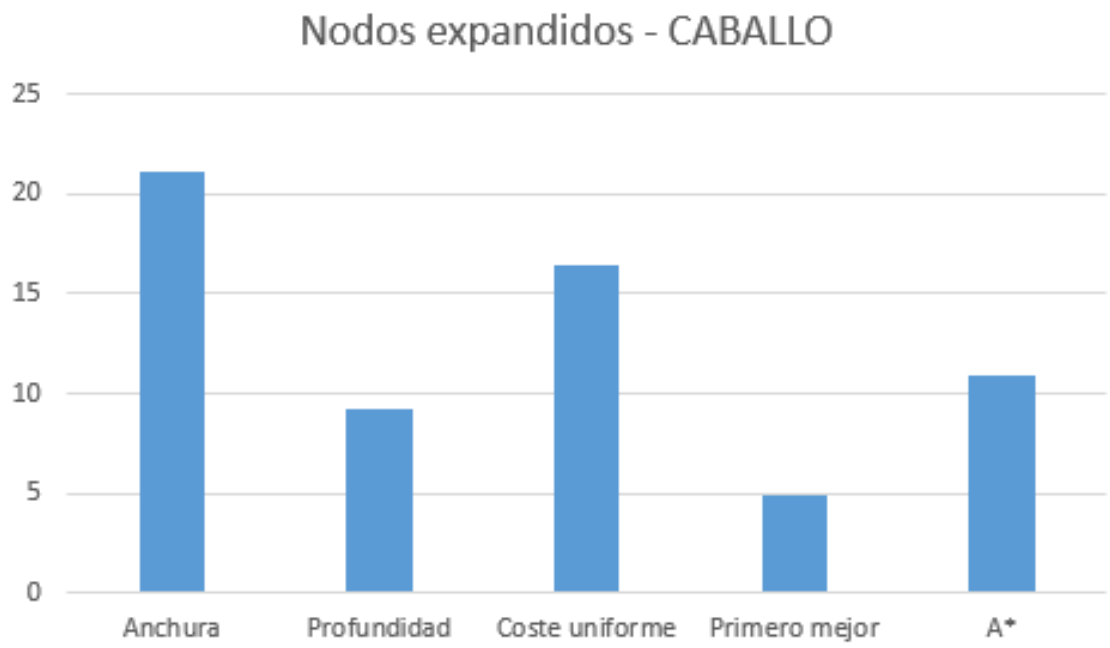

Imagen 10: Gráfico sobre los nodos expandidos con el caballo como agente

En este gráfico, podemos observar que el número de nodos expandidos resulta bastante diferente entre los tres agentes elegidos para el ejemplo. En resumen, los algoritmos de **profundidad** y **primero mejor** son los que menos nodos expanden por funcionamiento. Además, podemos ver que la torre, pese a contar con movimientos simples, es el agente que más nodos expande en cada algoritmo.

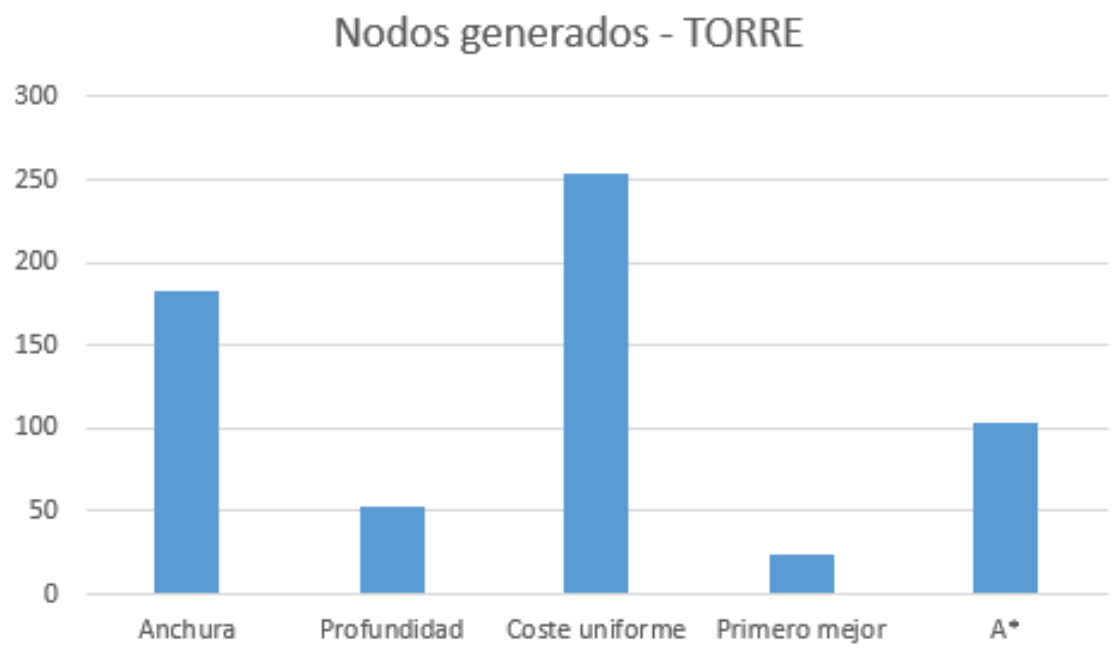

Imagen 11: Gráfico sobre los nodos generados con la torre como agente

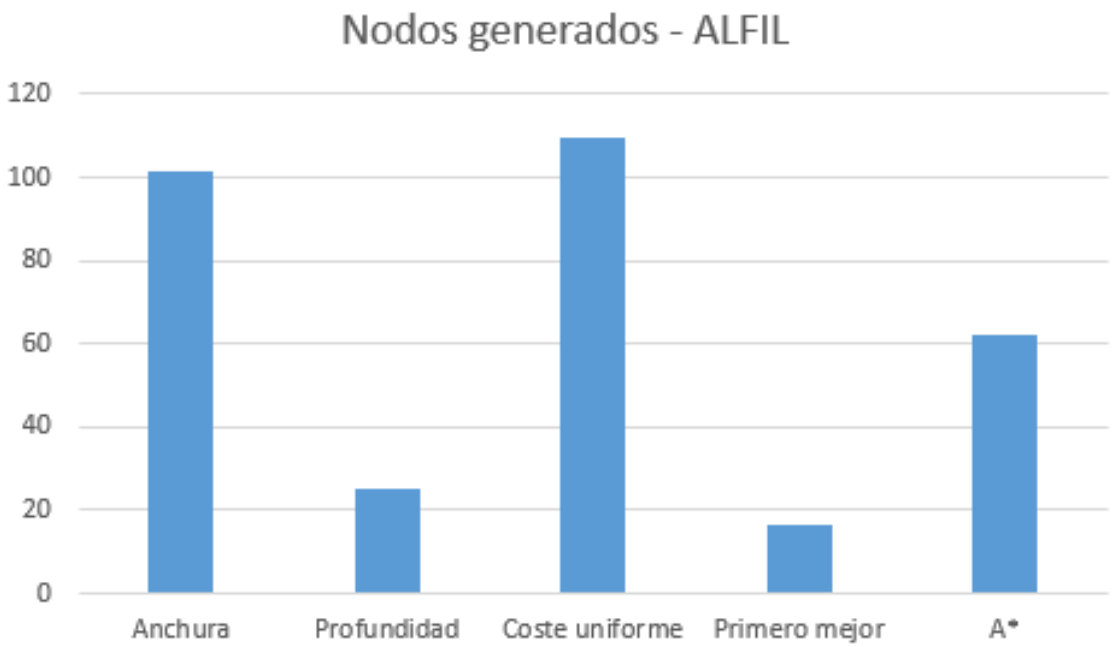

Imagen 12: Gráfico sobre los nodos generados con el alfil como agente

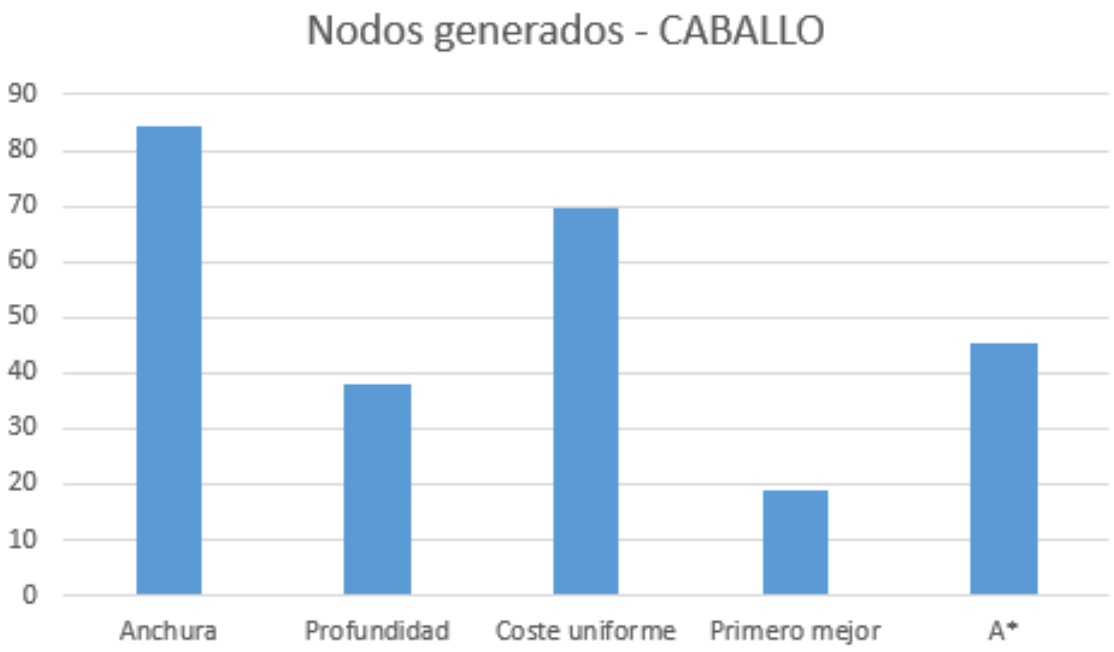

Imagen 13: Gráfico sobre los nodos generados con el caballo como agente

De nuevo, nos encontramos ante una situación similar a la previa en cuanto a generación de nodos se refiere. Podemos observar que **profundidad** y **primero mejor** mantienen un número bajo de nodos generados en comparación a los otros algoritmos; también, la torre sigue generando muchos más nodos que el resto de agentes. Esto puede haber sido causado por los movimientos de la pieza, condicionados también por la existencia de otras piezas (tanto blancas como negras que se podrá comer) repartidas en el resto del tablero, y que impiden a la torre llegar a la casilla final del tablero con poco esfuerzo.

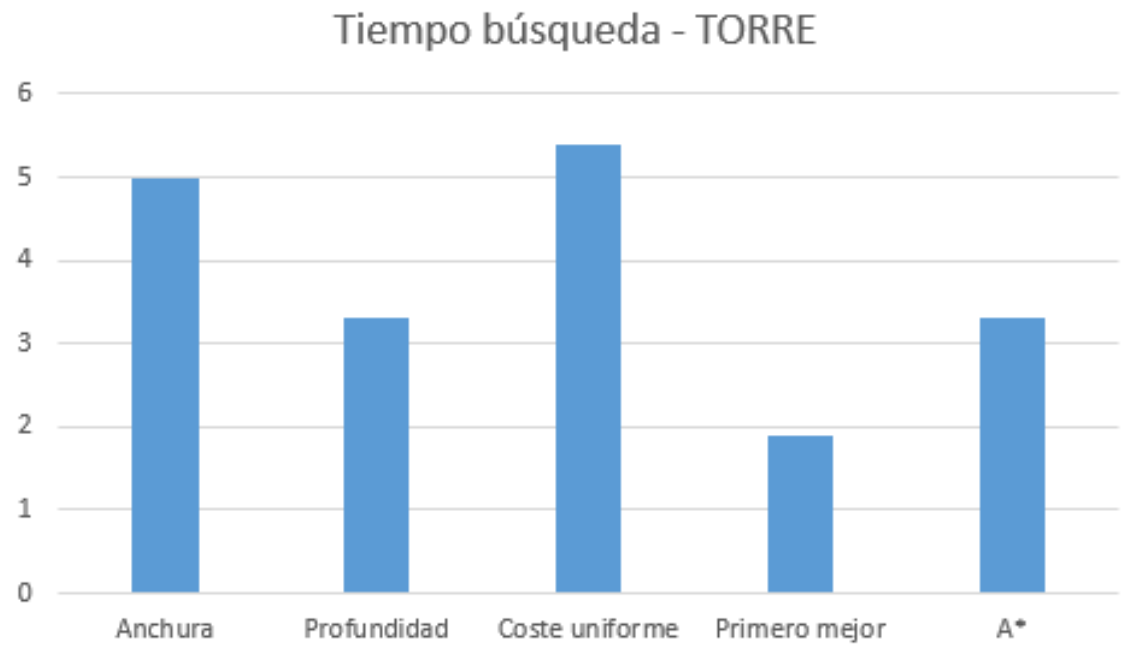

Imagen 14: Gráfico sobre el tiempo de ejecución (ms) con la torre como agente

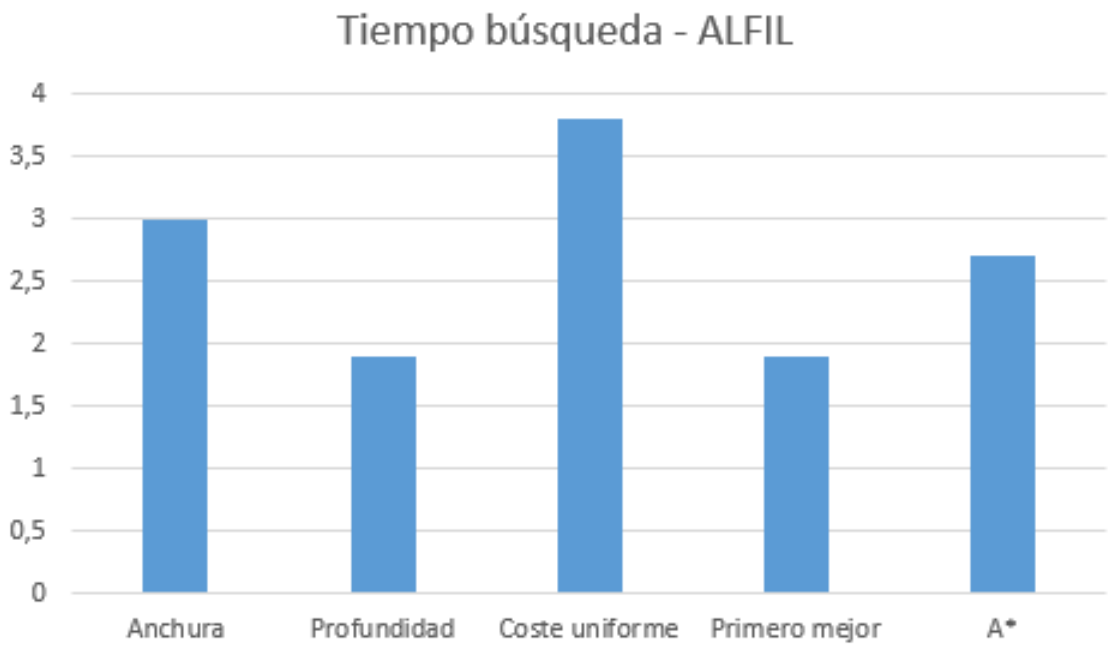

Imagen 15: Gráfico sobre el tiempo de ejecución (ms) con el alfil como agente

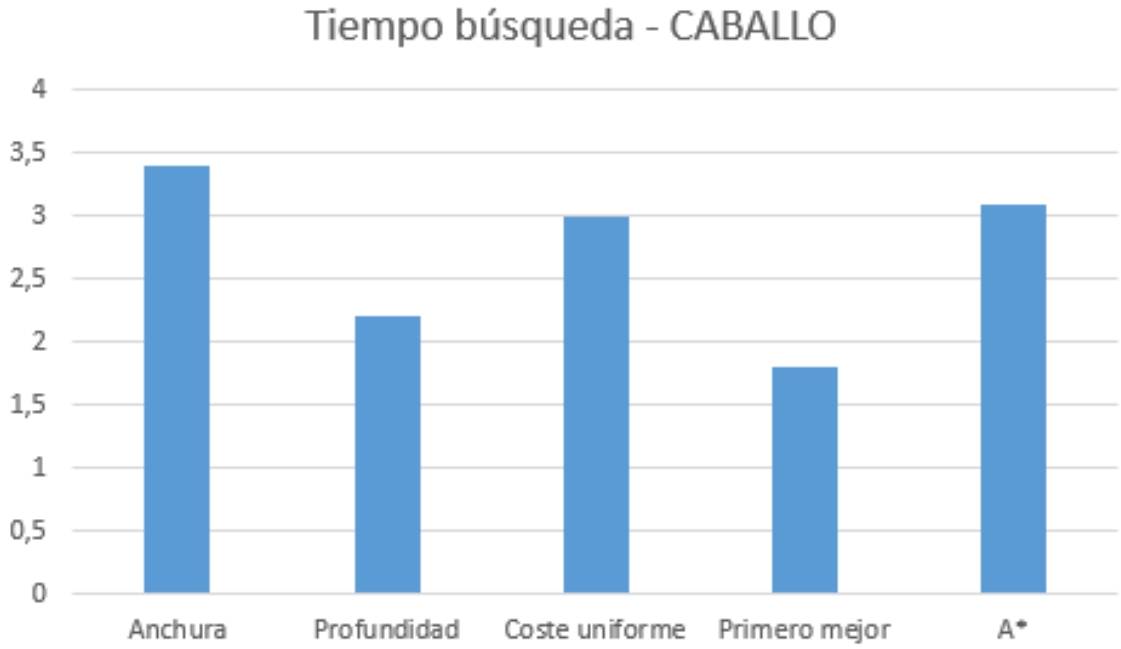

Imagen 16: Gráfico sobre el tiempo de ejecución (ms) con el caballo como agente

En cuanto a los tiempos de ejecución, vemos que son relativamente bajos y uniformes para cualquier agente, aunque van a ser siempre más óptimos dichos resultados en los algoritmos de búsqueda informada porque se presentan valores mejores.

Admisibilidad y consistencia de la heurística

Usaremos el siguiente escenario, en el que el caballo llega al final del tablero mediante el algoritmo de A*.

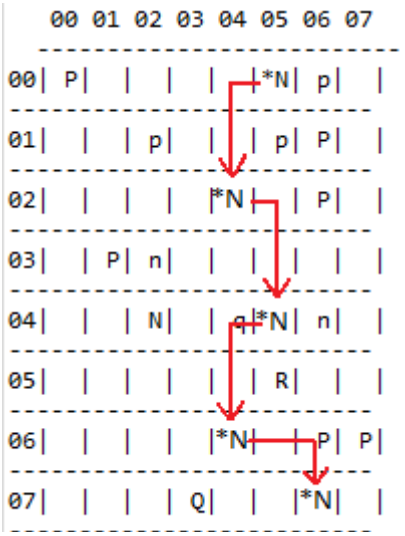

Ilustración 1: Camino del agente

La forma para calcular la heurística en cada nodo es mediante la siguiente fórmula:

Heurística = tamaño tablero – columna actual de la ficha

Respecto a la fórmula dada en el código base para obtener el coste de llegar a cada nodo:

Coste = 1 + max ([tam. Tablero – posición fila], [tam. Tablero – posición columna])

Cogemos las posiciones de los nodos para calcular el coste de cada nodo y su heurística:

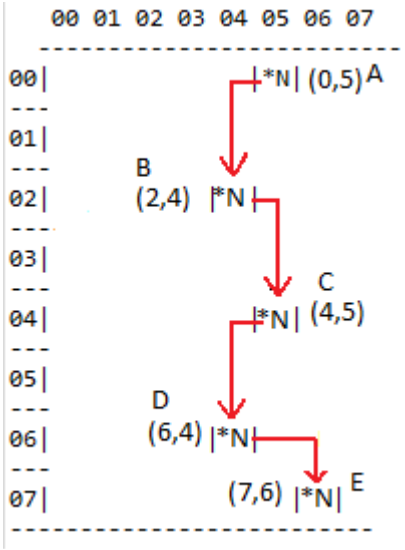

Ilustración 2: Nodos por los que pasa el agente

h(A) = 2 < C*(A) = 8, h(B) = 3 < C*(B) = 6, h(C) = 2 < C*(C) = 4, h(D) = 3 < C*(D) = 4, h(E) = 1 < C*(E) = 2

Por tanto, la heurística es **admisible.**

Conclusiones

La búsqueda informada es mucho más óptima que la de una búsqueda no informada, ya que el coste en general que necesita un ordenador en resolver dicho problema es mucho menor que en algoritmos como en el de anchura, el cual, dependiendo de la situación del tablero, puede generar una cantidad de nodos muy grande.

También debemos tener en cuenta que estas búsquedas sean en grafos para no entrar en un bucle. Dependiendo del algoritmo, se puede generar una cantidad de nodos muy alta, por lo que hay que cuidar el espacio de la memoria y los errores posibles que puede generar que una pieza se nos salga del tablero.

Por último, cabe destacar lo importante que es saber utilizar el *Debugger* en casos como este: al principio del desarrollo de la práctica, en una primera implementación de las piezas, nos encontramos con que, llegados al borde del tablero, las piezas se “salían” de los límites y generaban excepciones. En conclusión, sin la ayuda de esta herramienta, no se habría podido continuar con el desarrollo de los algoritmos.