10-4-2021

Ajedrez: Búsqueda no informada y heurística en el espacio de estados.

Alexandru Andrei Abrudan
IES AZARQUIEL

1. Introducción

Un problema típico de la Inteligencia Artificial consiste en buscar un estado concreto entre un conjunto determinado, al que se le llama espacio de estados. El objetivo de esta práctica es abordar este problema usando agentes que planifican, 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:

Los algoritmos de **búsqueda no informada**, también denominados de búsqueda ciega, son algoritmos de búsqueda que no dependen de información propia del problema a la hora de resolverlo, sino que proporcionan métodos generales para recorrer los árboles de búsqueda asociados a la representación del problema, por lo que se pueden aplicar en cualquier circunstancia. Se basan en la estructura del espacio de estados y determinan estrategias sistemáticas para su exploración, es decir, que siguen una estrategia fija a la hora de visitar los nodos que representan los estados del problema. Se trata también de algoritmos exhaustivos, de manera que, en el peor de los casos, pueden acabar recorriendo todos los nodos del problema para hallar la solución.

Por otro lado, la técnica de **búsqueda informada**, utiliza el conocimiento específico del problema para dar una pista sobre la solución del problema. La búsqueda informada puede ser ventajosa en términos del costo donde la optimización se logra con costos de búsqueda más bajos. Este algoritmo busca la ruto con coste más óptimo mediante la implementación de una **función heurística h(n)**, en la cual se insertan los estados más prometedores y esta devuelve un costo de ruta aproximado calculado desde el estado hasta el estado objetivo. Es por esto que aquí, la parte más importante de la técnica informada es la función heurística que facilita la transmisión del conocimiento adicional del problema al algoritmo.

2. Descripción del problema

El entorno de nuestro problema consiste en un tablero de ajedrez en el cual nuestro agente de búsqueda será una pieza blanca situada en la fila 0 y su objetivo o estado final será alcanzar la última fila del tablero. Pero no es tan fácil como parece, el tablero también estará poblado por otras piezas estáticas, blancas o negras, que harán de obstáculo. Por otro lado, nuestro agente podrá ser cualquier pieza del ajedrez, a nuestra elección, y solo podrá sus movimientos o descartar las piezas enemigas según las normas del Ajedrez. El tamaño del tablero, la densidad y la colocación de los obstáculos son variables que se pueden modificar con el fin de realizar unas pruebas más exhaustivas.

Las instancias del problema que hemos decidido usar para comprobar el desempeño de los algoritmos son:

-Tamaño del tablero: 8x8

-Densidad de obstáculos: 50%

-Colocación obstáculos: semillas (2020 a 2029)

-Fichas escogidas: Reina y Caballo

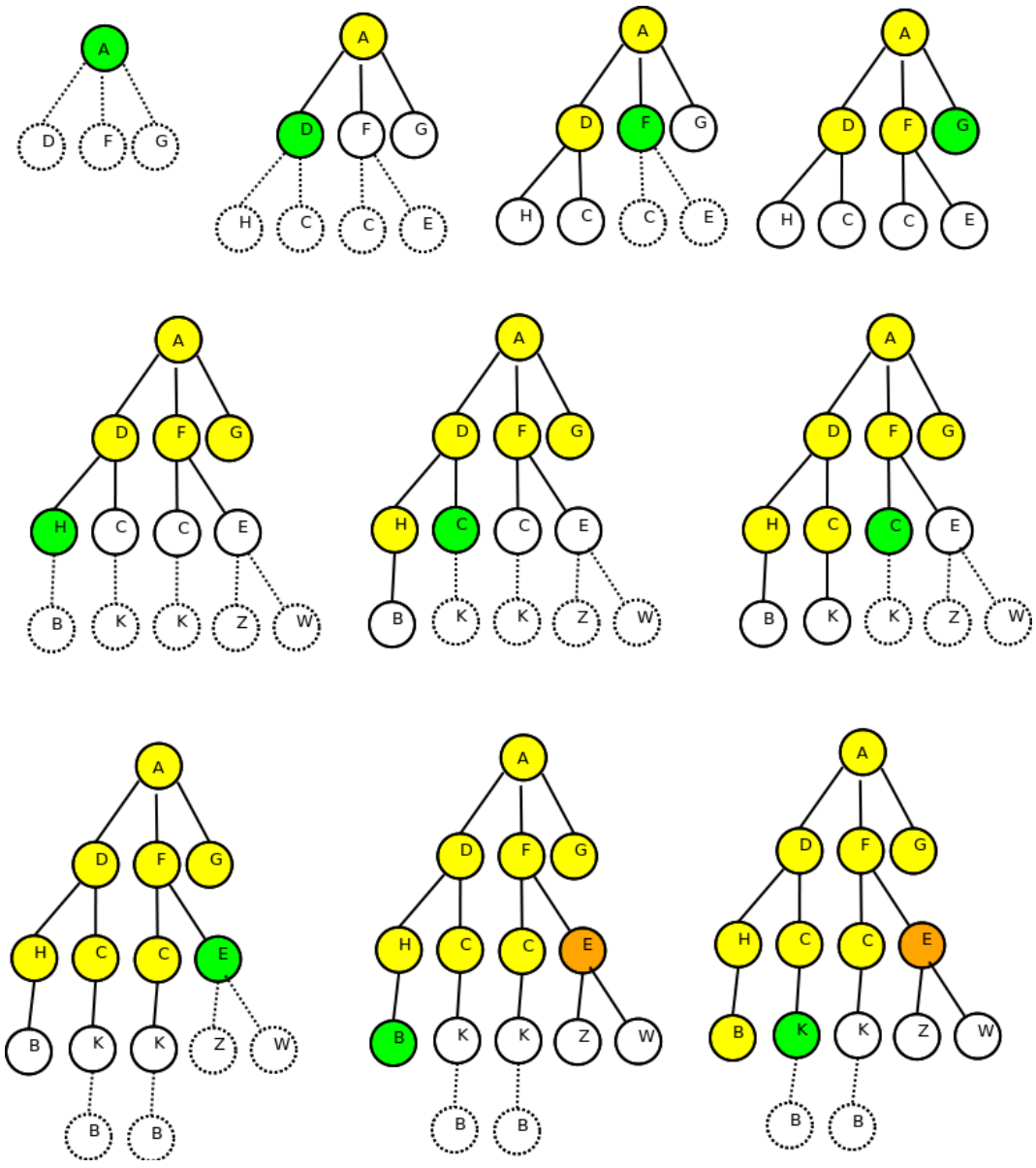
Se ha optado por estas instancias por ser las recomendadas por el autor de la práctica y por parecer lo suficientemente complejas como para mostrar el desempeño real de cada algoritmo en nuestro problema.

3. Búsquedas no informadas

Los algoritmos de búsqueda no informada que se han implementado para la resolución de nuestro problema son:

3.1. Búsqueda en Anchura

La idea principal de la Búsqueda en Anchura (BFS) consiste en visitar todos los estados que hay a profundidad “i” antes de pasar a visitar aquellos que hay a profundidad “i+1”. Es decir, tras visitar un estado, pasamos a visitar a sus hermanos antes que a sus hijos. Este algoritmo es completo, es decir, si existe solución, este algoritmo la encuentra. Más aún, es óptimo, en el sentido de que, si hay solución, encuentra una de las soluciones a distancia mínima de la raíz.



El algoritmo que se ha diseñado para aplicar la búsqueda en anchura es la siguiente:

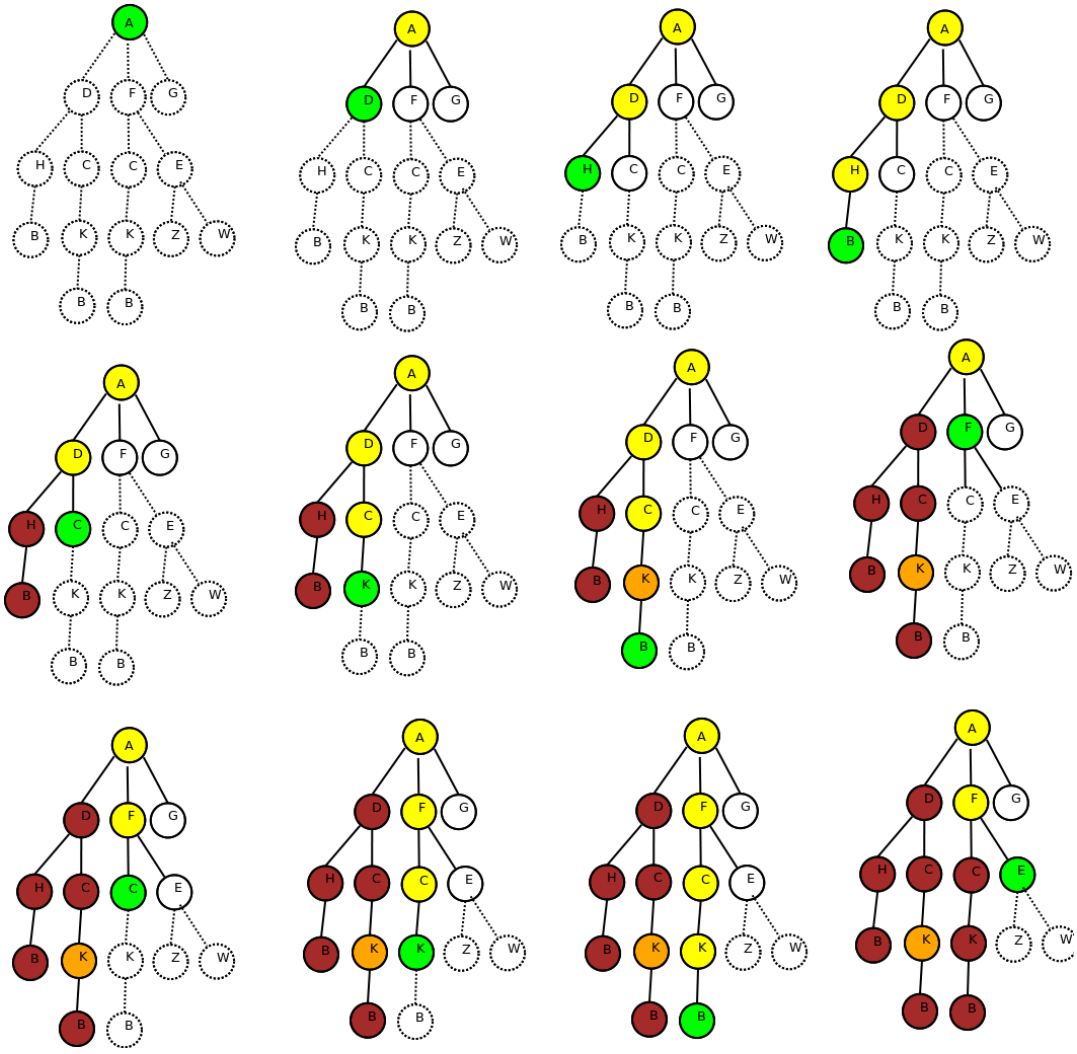
```
FUNCIÓN breadthFirst()

-DECLARAR y OBTENER primerEstado;

-DECLARAR lista de estados abiertos; (Lista FIFO)

-DECLARAR lista de estados cerrados; (Lista HashMap)

-AÑADIR nuestro primer estado a nuestra lista de estados abiertos;

-Repetir hasta que la lista de estados abierta este vacía:

        -estado: primer estado de la lista abierta;

        -idEstado: coordenadas en la tabla del estado;

        -AÑADIR a la lista cerrada el estado y su id como identificado;

        -DECLARAR lista de caminos hijos;

        -AÑADIR a la lista de caminos hijos los posibles caminos hijos de Estado;

        -REPETIR cada camino hijo de la lista caminos hijos

                -nuevoEstado: aplicar camino hijo a estado;

                -idNuevoEstado: coordenadas en la tabla del nuevoEstado;

                -nuevoEstado.listaCaminosPadre: añadir camino utilizado para llegar a
                este estado;

                -SI nuevoEstado esta en lista cerrada

                        -SI nuevoEstado es Final RETORNAR nuevoEstado como resultado;

                        -SI NO AÑADIR nuevoEstado a lista abierta;

        -FIN REPETIR
```

La principal característica de este algoritmo es que se utilizar una lista FIFO (First Input First Output) para almacenar la lista de estados abiertos, es decir, los estados que no se han examinado aún. Que esta lista sea de tipo FIFO es lo que permite que la búsqueda del estado objetivo sea en anchura puesto que al ser añadidos antes los estados padres serán examinados estos antes que sus hijos.

3.2. Búsqueda en Profundidad

Al igual que en el caso de la búsqueda en anchura, la búsqueda en profundidad (DFS) también puede ser vista como un proceso por niveles, pero con la diferencia de que, tras visitar un nodo, se visitan sus hijos antes que sus hermanos, por lo que el algoritmo tiende a bajar por las ramas del árbol hacia las hojas antes de visitar cada una de las ramas posibles.

DFS no es ni óptimo ni completo. No es óptimo porque si existe más de una solución, podría encontrar la primera que estuviese a un nivel de profundidad mayor, y puede no ser completo porque tomar un camino infinito sin solución.



El algoritmo que se ha diseñado para aplicar la búsqueda en profundidad es la siguiente:

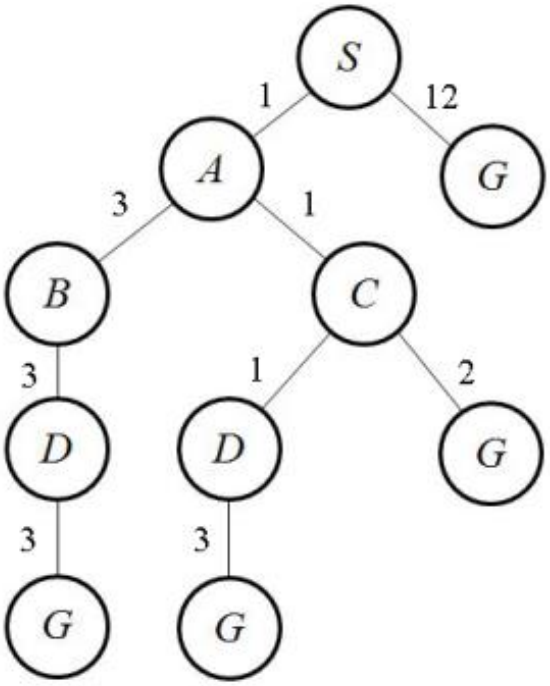
```
FUNCIÓN depthFirst()

-DECLARAR y OBTENER primerEstado;

-DECLARAR lista de estados abiertos; (Lista LIFO)

-DECLARAR lista de estados cerrados; (Lista HashMap)

-AÑADIR nuestro primer estado a nuestra lista de estados abiertos;

-Repetir hasta que la lista de estados abierta este vacía:

        -estado: primer estado de la lista abierta;

        -idEstado: coordenadas en la tabla del estado;

        -AÑADIR a la lista cerrada el estado y su id como identificado;

        -DECLARAR lista de caminos hijos;

        -AÑADIR a la lista de caminos hijos los posibles caminos hijos de Estado;

        -REPETIR cada camino hijo de la lista caminos hijos

                -nuevoEstado: aplicar camino hijo a estado;

                -idNuevoEstado: coordenadas en la tabla del nuevoEstado;

                -nuevoEstado.listaCaminosPadre: añadir camino utilizado para llegar a
                este estado;

                -SI nuevoEstado esta en lista cerrada

                        -SI nuevoEstado es Final RETORNAR nuevoEstado como resultado;

                        -SI NO AÑADIR nuevoEstado a lista abierta;

        -FIN REPETIR
```

La principal característica de este algoritmo, al contrario que el anterior, es que utilizar una lista LIFO (Last Input First Output) para almacenar la lista de estados abiertos, es decir, los estados que no se han examinado aún. La lista LIFO es la que permite que la búsqueda del estado objetivo sea en profundidad ya que siempre será examinado ultimo estado añadido a la lista.

3.3. Búsqueda Coste Uniforme

La búsqueda de coste uniforme (BCU) es un algoritmo de búsqueda no informada que se basa en recorrer el árbol de estado siguiendo el camino de menor coste entre el estado raíz y el estado destino, es decir, la búsqueda comienza por el nodo raíz y continúa visitando el siguiente estado que tiene menor coste y así los estados son visitados de esta manera hasta que el estado objetivo es alcanzado. Para esto debemos aplicar un coste a cada camino o acción que tomamos para cambiar de estado. En este caso hemos usado el siguiente coste:

```
"coste((f1, c1) -→ (f2, c2)) = max(|f1 - f2|, |c1 - c2|) + 1"
```



Usando este ejemplo nuestro algoritmo escogerá el camino (`S->A->C->G`) para llegar al objetivo G puesto que es el camino con menor coste, pero esto no implica que siempre escoja el camino más corto puesto que puede optar por un camino que a priori puede parecer el más óptimo, pero en realidad no lo sea para llegar a nuestro estado objetivo. Por último, la búsqueda BCU es completa y óptima.

El algoritmo que se ha diseñado para aplicar la búsqueda con coste uniforme es la siguiente:

```
FUNCIÓN uniformCost()
-DECLARAR y OBTENER primerEstado;
-DECLARAR lista de estados abiertos; (Lista de prioridad por coste de camino)
-DECLARAR lista de estados cerrados; (Lista HashMap)
-AÑADIR nuestro primer estado a nuestra lista de estados abiertos;
-Repetir hasta que la lista de estados abierta este vacía:
        -estado: primer estado de la lista abierta;
        -idEstado: coordenadas en la tabla del estado;
        -AÑADIR a la lista cerrada el estado y su id como identificador;
        -SI nuevoEstado es Final RETORNAR nuevoEstado como resultado;
        -DECLARAR lista de caminos hijos;
        -AÑADIR a la lista de caminos hijos los posibles caminos hijos de Estado;
        -REPETIR cada camino hijo de la lista caminos hijos
                -nuevoEstado: aplicar camino hijo a estado;
                -idNuevoEstado: coordenadas en la tabla del nuevoEstado;
                -nuevoEstado.coste: conseguir coste del camino;
                -nuevoEstado.listaCaminosPadre: añadir camino utilizado para llegar a
                este estado;
                -SI nuevoEstado esta en lista cerrada
                        - AÑADIR nuevoEstado a lista abierta;
                -SI NO
                        -estadoCerrado: conseguir nuevoEstado de lista cerrada;
                        -SI nuevoEstado.coste < estadoCerrado.coste
                                -ELIMINAR estadoCerrado de lista abiertos;
                                -AÑADIR nuevoEstado a lista abiertos;
        -FIN REPETIR
```

Como en el resto de algoritmos la principal característica para el correcto funcionamiento de este es la lista de estados abiertos que, en este caso, es una lista de prioridad que cada vez que recibe un estado nuevo, reordena la lista según el coste para llegar cada estado. Otro gran cambio es que al sacar un estado de la lista de abiertos comprobamos directamente si es el estado objetivo y por ultimo al final del algoritmo también hay que comprobar que, en el caso de que tengamos un estado por el que ya hemos pasado, el camino que hemos usado para llegar de nuevo a dicho estado no tenga un menor coste.

4. Búsqueda Informada

A continuación, vamos a ver la heurística y los algoritmos de búsqueda informada que hemos implementado para la resolución de nuestro problema.

4.1. Heurística

La función heurística h(n) que se ha diseñado para afrontar nuestro problema, indicando al agente que estado es el más prometedor, es a la vez que simple, eficaz. Esto demuestra que no necesariamente se necesita un algoritmo complejo para afrontar un problema complejo.

Nuestra h(n) recibe como parámetro el estado, y calcula la longitud que hay desde dicho estado hasta el estado objetivo. Por ultimo retorna dicha longitud como coste del estado.

El algoritmo que se ha diseñado para implementar la heurística es la siguiente:

```
FUNCIÓN Double getObjetiveLength (State estado)
-posicionActual: estado.posicion;
-posicionObjetivo: estado.tabla.posicionObjetivo;
-RETORNAR posicionObjetivo-posicionActual;
```

4.2. Búsqueda Primero Mejor

La búsqueda informada Primero Mejor (BPM) o también denominada Voraz es un algoritmo de búsqueda que se basa en el algoritmo no informado coste uniforme que ya hemos visto.

Es por esto que no voy a entrar mucho en detalle ni en este algoritmo ni en el siguiente para evitar repetirme con las explicaciones.

La única diferencia que existe entre ellos es que, el algoritmo BPM utilizar la función heurística para calcular el coste que tiene un camino en relación con el estado objetivo por lo tanto siempre va a optar por los caminos más cortos respecto al objetivo sin tener en cuenta el coste de los propios caminos.

El algoritmo que se ha diseñado para aplicar la búsqueda primero a mejor es la siguiente:

```
FUNCIÓN bestFirst()
-DECLARAR y OBTENER primerEstado;
-DECLARAR lista de estados abiertos; (Lista de prioridad por coste de heurística)
-DECLARAR lista de estados cerrados; (Lista HashMap)
-AÑADIR nuestro primer estado a nuestra lista de estados abiertos;
-Repetir hasta que la lista de estados abierta este vacía:
        -estado: primer estado de la lista abierta;
        -idEstado: coordenadas en la tabla del estado;
        -AÑADIR a la lista cerrada el estado y su id como identificador;
        -SI nuevoEstado es Final RETORNAR nuevoEstado como resultado;
        -DECLARAR lista de caminos hijos;
        -AÑADIR a la lista de caminos hijos los posibles caminos hijos de Estado;
        -REPETIR cada camino hijo de la lista caminos hijos
                -nuevoEstado: aplicar camino hijo a estado;
                -idNuevoEstado: coordenadas en la tabla del nuevoEstado;
                -nuevoEstado.coste: conseguir coste del camino;
                -nuevoEstado.listaCaminosPadre: añadir camino utilizado para llegar a
                este estado;
                -SI nuevoEstado esta en lista cerrada
                        - AÑADIR nuevoEstado a lista abierta;
                -SI NO
                        -estadoCerrado: conseguir nuevoEstado de lista cerrada;
                        -SI nuevoEstado.coste < estadoCerrado.coste
                                -ELIMINAR estadoCerrado de lista abiertos;
                                -AÑADIR nuevoEstado a lista abiertos;
        -FIN REPETIR
```

Al igual que el algoritmo coste uniforme utilizar una lista de prioridad, pero esta está ordenada en función del coste que devuelve la función heurística.

4.3. A*

La búsqueda informada A* es un algoritmo de búsqueda que intenta resolver el inconveniente que tiene el algoritmo Primero Mejor, que es el no tener en cuenta el coste de los caminos que toma. Para la aplicación de este algoritmo tan solo hay que sumar el costo del propio camino en sí y el costo del camino respecto al estado objetivo y asignarlo a cada estado destino. Con esta simple adición este algoritmo va a optar por el camino hacia el estado más óptimo respecto a coste normal y coste destino. Este algoritmo es completo, admisible y óptimamente eficiente.

El algoritmo que se ha diseñado para aplicar la búsqueda A* es la siguiente:

```
FUNCIÓN aStar()

-DECLARAR y OBTENER primerEstado;

-DECLARAR lista de estados abiertos; (Lista de prioridad con coste del camino y coste de heurística)

-DECLARAR lista de estados cerrados; (Lista HashMap)

-AÑADIR nuestro primer estado a nuestra lista de estados abiertos;

-Repetir hasta que la lista de estados abierta este vacía:

        -estado: primer estado de la lista abierta;

        -idEstado: coordenadas en la tabla del estado;

        -AÑADIR a la lista cerrada el estado y su id como identificador;

        -SI nuevoEstado es Final RETORNAR nuevoEstado como resultado;

        -DECLARAR lista de caminos hijos;

        -AÑADIR a la lista de caminos hijos los posibles caminos hijos de Estado;

        -REPETIR cada camino hijo de la lista caminos hijos

                -nuevoEstado: aplicar camino hijo a estado;

                -idNuevoEstado: coordenadas en la tabla del nuevoEstado;

                -nuevoEstado.coste: conseguir coste del camino;

                -nuevoEstado.listaCaminosPadre: añadir camino utilizado para llegar a
                este estado;

                -SI nuevoEstado esta en lista cerrada

                        - AÑADIR nuevoEstado a lista abierta;

                -SI NO

                        -estadoCerrado: conseguir nuevoEstado de lista cerrada;

                        -SI nuevoEstado.coste < estadoCerrado.coste

                                -ELIMINAR estadoCerrado de lista abiertos;

                                -AÑADIR nuevoEstado a lista abiertos;

        -FIN REPETIR
```

5. Comparaciones

A continuación, vamos a comparar los resultados de todos los algoritmos que hemos diseñado usando la instancia de nuestro problema. Además, incluiremos las pruebas realizadas con un agente que no planifica, es decir, los resultados obtenidos al aplicar un algoritmo que buscaba el estado objetivo aleatoriamente sin seguir ninguna norma. Esto nos permitirá destacar la eficacia en tiempo y caminos que se obtienen al aplicar agentes de búsqueda que planifican, ya sea con búsqueda no informada o informada.

	Búsqueda Aleatoria					
	Media					
Pieza	Longitud	Coste	Nodos generados	Nodos explorados	Nodos expandidos	Desviación estándar
Reina	16.34	52.14	256.79	17.34	16.34	2020 a 2029
Caballo	24.5	73.5	136.18	25.5	24.5	

	Búsqueda Anchura					
	Media					
Pieza	Longitud	Coste	Nodos generados	Nodos explorados	Nodos expandidos	Desviación estándar
Reina	2.0	9.2	561.7	38.5	37.5	2020 a 2029
Caballo	4.0	12.0	544.6	99.6	98.6	

	Búsqueda Profundidad					
	Media					
Pieza	Longitud	Coste	Nodos generados	Nodos explorados	Nodos expandidos	Desviación estándar
Reina	5.3	19.9	81.2	6.3	5.3	2020 a 2029
Caballo	5.8	17.4	25.0	6.9	5.9	

	Búsqueda Coste Uniforme					
	Media					
Pieza	Longitud	Coste	Nodos generados	Nodos explorados	Nodos expandidos	Desviación estándar
Reina	2.1	9.1	8641.0	531.3	530.3	2020 a 2029
Caballo	4.0	12.0	565.2	103.1	102.1	

	Búsqueda Primero Mejor					
	Media					
Pieza	Longitud	Coste	Nodos generados	Nodos explorados	Nodos expandidos	Desviación estándar
Reina	2.5	9.5	36.0	3.5	2.5	2020 a 2029
Caballo	4.0	12.0	12.0	5.0	4.0	

	Búsqueda A*					
	Media					
Pieza	Longitud	Coste	Nodos generados	Nodos explorados	Nodos expandidos	Desviación estándar
Reina	2.1	9.1	300.4	21.1	20.1	2020 a 2029
Caballo	4.0	12.0	235.0	42.9	41.9	

6. Conclusión

Como se puede observar cada algoritmo va dando mejores resultados, aunque a coste de procesamiento y de memoria en nodos expandidos.

Personalmente pienso que el algoritmo más óptimo para resolver nuestro problema planteado seria la búsqueda **Primero Mejor** porque considero que a pesar de que de un camino un poco más largo que el algoritmo búsqueda A*, el tiempo de procesamiento y la memoria ahorrada es clave para escogerlo por encima de A*.