# SISTEMAS INTELIGENTES

## ESTUDIO Y CONCLUSIONES DEL ALGORITMO A*

Universidad de Alicante
Escuela Politécnica Superior
Ingeniería Informática

Oscar Casado Lorenzo

# Índice

1. Explicación algoritmo A* ..... 2
2. Implementación ..... 3
3. Traza de un caso acotado ..... 7
4. Heurísticas ..... 9
4.1. Heurísticas estudiadas ..... 9
4.1.1. Heurística sin definir ..... 9
4.1.2. Distancia Manhattan ..... 9
4.1.3. Distancia euclídea ..... 9
4.1.4. Distancia en entornos hexagonales ..... 9
4.2. Comparativa ..... 10
4.2.1. Tiempo ..... 10
4.2.2. Nodos expandidos ..... 10
4.2.3. Recorrido ..... 10
5. Cuestiones ..... 11
5.1. ¿Se incluye el nodo inicial y el final en el camino? ..... 11
5.2. ¿El camino explorado empieza por 0 o por 1? ..... 11
5.3. ¿Podemos evitar que A* se recalcule en cada turno? ..... 11
5.4. ¿De qué manera podríamos dotar de inteligencia al dragón? ..... 12
5.5. ¿Cuál es tamaño máximo obteniendo tiempos asequibles? ..... 12
6. Conclusiones ..... 12
7. Bibliografía ..... 13

# 1. Explicación algoritmo A*

El algoritmo A* se clasifica dentro de los algoritmo de búsqueda de grafos. Su función es encontrar siempre y cuando se cumplan determinadas condiciones, el camino de menor coste entre un nodo inicial y uno objetivo, siendo la búsqueda tanto completa (en caso de existir siempre dará con ella) como óptima.

Además, se deber tener muy en cuenta el concepto de heurística apara este algoritmo, esta se trata de una suposición o hipótesis basada tanto en la experiencia como en datos incompletos, todo ello para emitir un juicio de valor con el fin de emitir un juicio de valor. Un ejemplo de esta dinámica sería la siguiente:

**Dato >** Hay nubes, **Heurística >** “Va a llover”

Tras haber presentado este concepto, hay que destacar que la calidad de dichos juicios determinará el coste computacional del algoritmo pudiendo obtener en el peor caso una complejidad exponencial, y en el mejor de ellos una complejidad lineal.

En cuanto a otras de las características de este algoritmo encontramos su mayor problemática, el espacio que requiere dado que tiene que almacenar todos los posibles siguientes nodos de cada estado (como se desarrollará a continuación), requiriendo una cantidad de memoria exponencial con respecto al tamaño del problema. Aunque existen variaciones de este algoritmo que mejoran este aspecto (RTA*, IDA*) no compete a este documentación tratarlas.

Ahora sí entraremos en el eje central del algoritmo A*, su funcionamiento. Este algoritmo se rige según la siguiente función de evaluación, donde:

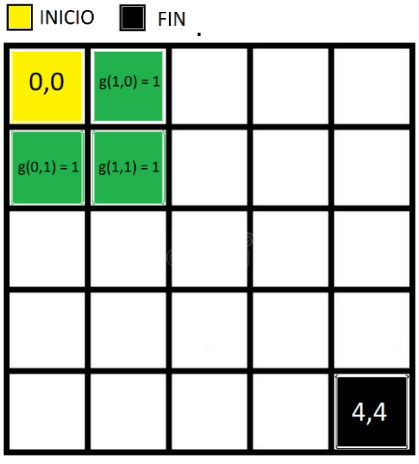
$$f(n) = g(n) + h(n)$$

g(n): Representa el coste real del camino para llegar al nodo actual.
h(n): Representa el valor heurístico del nodo a evaluar desde el actual ´*n*´ hasta el final.

Por otro lado, encontramos que A* posee dos estructuras de datos auxiliares basadas en listas de nodos, que podemos denominar **abiertos (listaFrontera)**, la cual englobará todos aquellos nodos por los que resulta factible seguir expandiendo hasta alcanzar el nodo destino y la lista **cerrados (listaInterior)** que recogerá todos los nodos ya expandidos y analizados. De esta forma el algoritmo es capaz de evaluar si ha de cambiar el camino de búsqueda valorando si existen nodos más prometedores que los anteriores.

Conocidos todos los elementos fundamentales para la implementación del algoritmo pasamos al pseudocódigo de este, aplicable a cualquier lenguaje de programación.

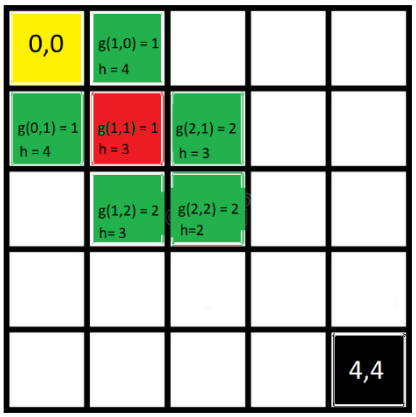
Para explicar más detalladamente la interacción entre todos los elementos anteriores me apoyaré en una representación gráfica de un escenario bidimensional compuesto por nodos cuadrados pues es el caso base de la implementación hexagonal que veremos en apartados posteriores




Como podemos observar, encontramos un escenario con una casilla origen y una destino. En resumidas cuentas, como nuestro objetivo es encontrar un camino entre ellos dos partiendo desde el primero de ellos debemos analizar si es factible pasar por alguno de los vecinos que se encuentran resaltados en verde.

Como todos ellos son factibles los añadimos a la estructura ‘listaFrontera’ y pasamos a decidir cual es el más apropiado para seguir expandiendo; sin embargo, al poseer el mismo valor de coste real, debemos tener en cuenta un segundo factor que nos ayude a determinar el más optimo, la heurística.

Mediante la implementación de una heurística que nos determine, por ejemplo, como de lejano se encuentra un nodo de la coordenada (4,4) ya tendríamos esa 'suposición' mencionada anteriormente que nos ayudará a reducir el número de nodos explorados y decidirnos de entre los 3 posibles por el correspondiente a la coordenada (1,1). Tras esta elección pasaremos el nodo de la primera estructura de datos a la segunda, 'listaInterior' ya que ha sido analizado y empleado para expandir nodos.



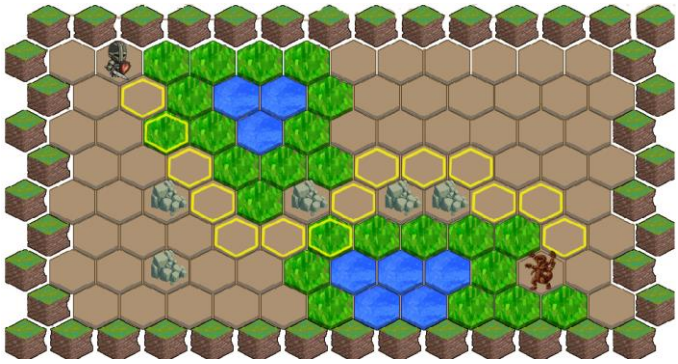

Iterando este procedimiento y teniendo en cuenta otra serie de especificaciones que se verán posteriormente tendríamos implementado el funcionamiento de este conocido algoritmo,

# 2. Implementación

Pasando a la implementación de A* puede ser desarrollada sobre múltiples lenguajes de programación y tecnologías actuales, por lo que es necesario conocer el pseudocódigo común y comprender su significado (representado brevemente en el apartado anterior) para llevar a cabo nuestra propia representación de este.

Sin embargo, y teniendo en cuenta la naturaleza del algoritmo y su propósito se ha decido que sea JAVA el lenguaje matriz debido, en gran medida a estar estrechamente relacionado con la POO (Programación Orientada a Objetos).

Para reflejar las ventajas que esto nos puede ofrecer, dedicaré unas líneas a contextualizar nuestro entorno de trabajo.



```
Alg A*
    listaInterior = vacio
    listaFrontera = inicio

    mientras listaFrontera no esté vacía
        n = obtener nodo de listaFrontera con menor f(n) = g(n) + h(n)
        listaFrontera.del(n)
        listaInterior.add(n)

        si n es meta
            devolver
            reconstruir camino desde la meta al inicio siguiendo los punteros
        fsi

        para cada hijo m de n que no esté en listaInterior
            g'(m) = n.g + c(n, m)  //g del nodo a explorar m

            si m no está en listaFrontera
                almacenar la f, g y h del nodo en (m.f, m.g, m.h)
                m.padre = n
                listaFrontera.add(m)
            sino si g'(m) es mejor que m.g  //Verificamos si el nuevo camino es mejor
                m.padre = n
                recalcular f y g del nodo m
            fsi
        fpara
    fmientras
    devolver no hay solución
falg
```

Nos encontramos con un entorno bidimensional con nodos hexagonales en el que intervienen distintos elementos:

- **El caballero.**
  Representará al nodo inicial, dicho caballero variará su posición a uno de los nodos adyacente en cada jugada de forma aleatoria. Para ganar ha de encontrarse a una lejanía superior a 20 casillas del dragón.

- **El dragón.**
  Representará el nodo destino, este poseerá la misma metodología del caballero en lo que respecta al movimiento. Además, si coincide en localización con este último el juego termina de forma insatisfactoria.

- **Los bloques y las piedras.**
  Representarán elementos estáticos que no podrán ser traspasados por el caballero ni modificados por el jugador. La diferencia entre estos dos elementos se basa en que los bloques únicamente pueden ser generados a partir de la matriz de valores que inicializa el mapa, mientras que las piedras pueden ser colocadas de forma manual por el jugador en cada turno.

- **Los caminos, los lagos y los bosques.**
  Estos nodos constituyen los caminos transitables por el caballero y el dragón, se diferencian entre ellos únicamente los valores que conforman para el caballero acceder a ellos, es decir, en orden creciente el esfuerzo del caballero para pasar un camino (1) es menor que la que requiere para recorrer un bosque (2) y de igual forma para cruzar una laguna (3).

Además de estas características, para cada uno de los nodos debemos de tener en cuenta tanto su posición absoluta en el mapa como los distintos valores de g() y h() que va tomando a lo largo de la ejecución del algoritmo.

Por todo ello y ya que el lenguaje lo permite he decidido crear un objeto llamado Nodo que implemente todo lo mencionado, garantice la persistencia de datos cuando sea necesario y permita añadir otra serie de funcionalidades que nos serán de interés en etapas más avanzadas del desarrollo.

Sobre estas funcionalidades, las que me han parecido más interesantes son los métodos:

```
class Nodo{
    Nodo padre;
    Coordenada localizacion;
    int coste;
    int f;
    int g;
    int h;

    Nodo(Coordenada c){...23 lines }

    Coordenada getCoord() {return this.localizacion;}
    int getX(){return this.localizacion.getX();}
    int getY(){return this.localizacion.getY();}
    int getF(){return this.f;}
    int getG(){return this.g;}
    int getH(){return this.h;}
    Nodo getPadre(){return this.padre;}
    char getMaterial(){...}

    void setF(int f){this.f = f;}
    void setG(int g){this.g = g;}
    void setH(int h){this.h = h;}
    void setPadre(Nodo n){this.padre = n;}

    public ArrayList<Nodo> encontrarVecinas(){...37 lines }

    @Override
    public String toString (){...6 lines }

    public boolean equals(Nodo n) {...3 lines }
}
```

- **<u>toString()</u>**

Para facilitar la depuración y el testeo del código fuente, incluso del seguimiento de la traza.

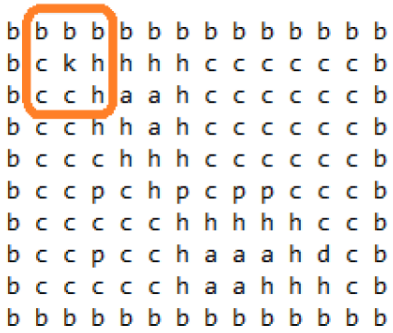
- **<u>equals()</u>**

Comprueba la similitud entre 2 nodos teniendo en cuenta únicamente los valores ‘x’ e ‘y’ de ambos.

- **<u>encontrarVecinos()</u>**

Esta es con diferencia el método auxiliar más relevante dentro de la implementación del propio objeto puesto que no servirá para invocar muy fácilmente a todos los vecinos válidos (considero vecinos válidos a todos aquellos a los que el caballero puede acceder sin impedimentos) adyacentes a uno dado.

Antes de lanzarnos a implementar dicho método debemos de tener en cuenta un aspecto fundamental y es que el entorno se almacena en una matriz bidimensional similar a la siguiente (correspondiente al mundo_defecto proporcionado por el profesorado):




Como se rescata de esta imagen, el caballero representado por el carácter ‘k’ se encuentra circundado por otros 8 nodos adyacentes, esto resulta inmediatamente contradictorio con el vecindario que deberíamos obtener de un nodo hexagonal, el cual debería de estar rodeado únicamente por otros 6.

Entonces, ahora que sabemos que hay que descartar 2 de los 8 supuestos vecinos vamos a determinar cuales de ellos son y en qué circunstancias se eliminan.

Como se puede observar en cualquiera de los mapas que creemos, las filas se encuentran desplazadas unas de otras por una diferencia de ½ casilla hexagonal, esto aun complica más la transcripción de la matriz cuadrada encargada de almacenar los valores de las casillas y los vecinos 'reales' a un nodo en este nuevo sistema de referencia. Por tanto, la distribución de las filas pares e impares posee distintos matices que deberemos controlar.

A modo de observación me gustaría puntualizar que nos es indiferente si consideramos la fila y columna 0 como par o impar ya que en ambas siempre tendremos colocadas bloques intraspasables e invaluables, por lo que este mediático concepto no resultará trascendente.

- **Fila impar**
  Se descartan los nodos que se encuentran (3.0) y (3.2), es decir la de arriba a la derecha y abajo a la derecha. Eso conforma que se descartarán las casillas (k.x+1, k.y-1) y (k.x+1, k.y+1).


- **Fila par**
  Se descartan los nodos (0.1) y (0.3), es decir el de arriba a la izquierda y abajo a la izquierda. Eso conforma que se descartarán las casillas (k.x-1, k.y-1) y (k.x-1, k.y+1).


En definitiva, esta es la implementación que tiene en cuenta todos los conceptos explicados anteriormente para que desde cualquier nodo podamos invocar a sus adyacentes.

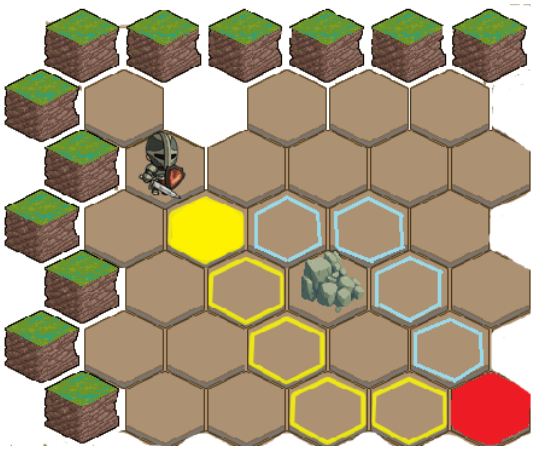
```
public ArrayList<Nodo> encontrarVecinas(){
        ArrayList<Nodo> vecinas = new ArrayList<>();

        //Recorremos las 9 posiciones que hay contiguas en la matriz, incluyendose a si misma
        for(int dx = this.getX() - 1; dx <= this.getX() + 1; dx++){
            for(int dy = this.getY() - 1; dy <= this.getY() + 1; dy++){

                Nodo posibleVecina = new Nodo(new Coordenada(dx, dy));
                //Si es un bloque excluida directamente
                if(posibleVecina.getMaterial() != 'b' && posibleVecina.getMaterial() != 'p'){
                    //Eliminamos la propia casilla y las que no son adyacentes en cada caso...
                    if(!(posibleVecina.getX() == this.getX() && posibleVecina.getY() == this.getY())){
                        //.. si la casilla es fila par las invalidas son (-1,-1) y (+1, -1)
                        if(this.getY() % 2 == 0 ){

                           if(posibleVecina.getX() == this.getX()+1 || posibleVecina.getX() == this.getX() ||
                              ((posibleVecina.getX() == this.getX() - 1) && (posibleVecina).getY() == this.getY()))
                               vecinas.add(posibleVecina);
                        }
                        else{
                            if( (posibleVecina.getX() == this.getX() - 1) || (posibleVecina.getX() == this.getX()) ||
                                ((posibleVecina.getX() == this.getX() + 1) && (posibleVecina).getY() == this.getY()))
                                vecinas.add(posibleVecina);
                                //System.out.println("Coordenada apta");
                            //else System.out.println("Excluida por no adyacente");
                        }
                    }
                }
                //else System.out.println("Excluida por obstaculo");
            }
        }
        return vecinas;
    }
```

Una vez introducida y desmenuzada la estructura Nodo pasare a la parte de creación e inicialización de variables entre las que destacan las estructuras de listas de nodos implementadas con ArrayList's los cuales nos permiten introducir y extraer elementos a partir de índices, lo cual nos será útil a la hora de realizar búsquedas sobre los mismos.

Una posible mejora de mi código sería implementar esto mismo mediante un Set con un patrón de ordenación ascendiente para evitar la complejidad algorítmica de la búsqueda binaria, sin embargo, por cuestiones de práctica y comodidad he decidido implementarlo de una forma más ingenua a la vez que comprensible.

Para finalizar este apartado otra parte de la codificación que me gustaría destacar es la elección del nodo más prometedor de entre todos los presentes en la lista openSet (listaFontera) ya que, como veremos la implementación, esto puede conducir a que para una misma situación se generen recorridos alternativos, lo cual ejemplificaré de manera gráfica.

En este escenario se aprecia de forma directa que existen 2 caminos alternativos para que el caballero llegue hasta el hexágono rojo, el compuesto por el nodo amarillo más los nodos verdes y alternativamente, el que conforma el nodo amarillo junto con los nodos azules, la pregunta es:



¿En qué parte del código se controla esta elección? pues bien, esto se tiene en cuenta en una función auxiliar empleada por el algoritmo A*.

```
public Nodo nodoOptimoOpenSet(ArrayList<Nodo> openSet){
    int indexGanador = 0;
    for(int i = 1; i < openSet.size(); i++){
        if(openSet.get(i).getF() < openSet.get(indexGanador).getF())
            indexGanador = i;
    }
    return openSet.get(indexGanador);
}
```

Vamos a establecer que en el openSet se encuentran únicamente esas 2 casillas, las amarilla y azul que componen el comienzo de cada uno de los caminos (lo que no es correcto ya que también estarían presentes todos los nodos adyacentes a los visitados anteriormente) por simplificar.

De acuerdo con este criterio la lista englobaría la siguiente información (lo representaré en formato JSON para que sea un formato estandarizado):

openSet{ nodoVerde{ F :'6', G: '2', H: '4' } , nodoAzul{ F :'6', G: '2', H: '4' } }

Pues bien, la función auxiliar empezaría recorriendo dicha estructura, escogiendo por defecto el primer elemento de la misma como nodo prometedor, sin embargo, esta seguirá iterando hasta haber analizado por completo si existe otra alternativa mejor. Como la condición para establecer un nuevo nodo se encuentra con esta situación:

¿nodoAzul.f < nodoVerde.f? su respuesta es que no se considera mejor y escoge el camino verde.

De todas formas, ambos caminos están bien y esta decisión recae sobre el desarrollador quien puede variar el operando '<' por '<=' para que el camino escogido sea el azul, no afectando en ningún caso al correcto funcionamiento de este.

Con esto doy por finalizado el análisis de mi implementación y dejo a disposición del lector el código relacionado para facilitar la consulta de otros aspectos como: la implementación recursiva para pintar el camino, el acumulador encargado de contabilizar los nodos expandidos, etc.

# 3. Traza de un caso acotado


Este es el mapa sobre el que trabajaremos y que coincide con el archivo 'pruebas.txt' adjuntado en el directorio /mundos. Es cierto que no cumple con el formato que espera el gestor gráfico, pero esto no influye en la ejecución del algoritmo y me parece muy apropiado al tratarse de un entorno muy reducido.

Me veo obligado a destacar que en este mapa el jugador nunca ganará ya que no posee la suficiente amplitud como para lograr una separación de 20 casillas entre la entidad del caballero y la del dragón, pero esto no afecta realmente al funcionamiento del algoritmo A* sino que se trata de una dinámica secundaria que no tendremos en cuenta para esta traza.

A lo largo de esta traza me referiré en diversas ocasiones a la cardinalidad (x,y) de cada nodo, por tanto para facilitar la comprensión de la misma facilito una plantilla sobre la que indico las coordenadas correspondientes a cada nodo y que lo identifican del resto.

Comenzaremos la ejecución añadiendo el nodo inicial (en el que se encuentra el caballero) a la estructura openSet. Como hemos añadido este elemento y la lista no se encuentra vacía podemos proseguir con el algoritmo eligiendo cual de todos los elementos de dicha lista resulta más prometedor para expandir, sin embargo, al solo contener un elemento en la primera iteración siempre será este el escogido.

Hacemos un comprobación rutinaria para asegurarnos de que dicho nodo no es la meta y proseguimos trasladando este a la segunda estructura 'closeSet' que contendrá todos aquellos nodos que ya han sido tenidos en cuenta, pero no sin antes extraer de este nodo todos los adyacentes a él que son válidos, es decir, los que no constatan ni un bloque, ni una piedra ni están presentes en esta segunda estructura ya mencionada.

Por cada uno de estos nodos debemos de reanalizar si se encuentran dentro de la lista 'openSet', si es el caso debemos acceder mediante un puntero a este elemento y modificar su valor de G (el coste real del camino hasta llegar a él) si el nuevo camino dispone de un valor menor de este mismo atributo. En caso de que no se encontrase en esta lista deberíamos crear un nuevo Nodo e inicializar sus valores de F, G y H pero ¿cómo hacemos esto?

Como F está compuesta por los operandos G y H únicamente debemos evaluar estos 2 y sumándolos inmediatamente obtendremos este valor, en cuanto a la H comentaremos en el apartado siguiente los métodos que podemos emplear para calcular el mismo. El coste real del camino, G se calcula de la siguiente forma:

int gNodoActual = nodoPrometedor.getG() + nodoActual.coste

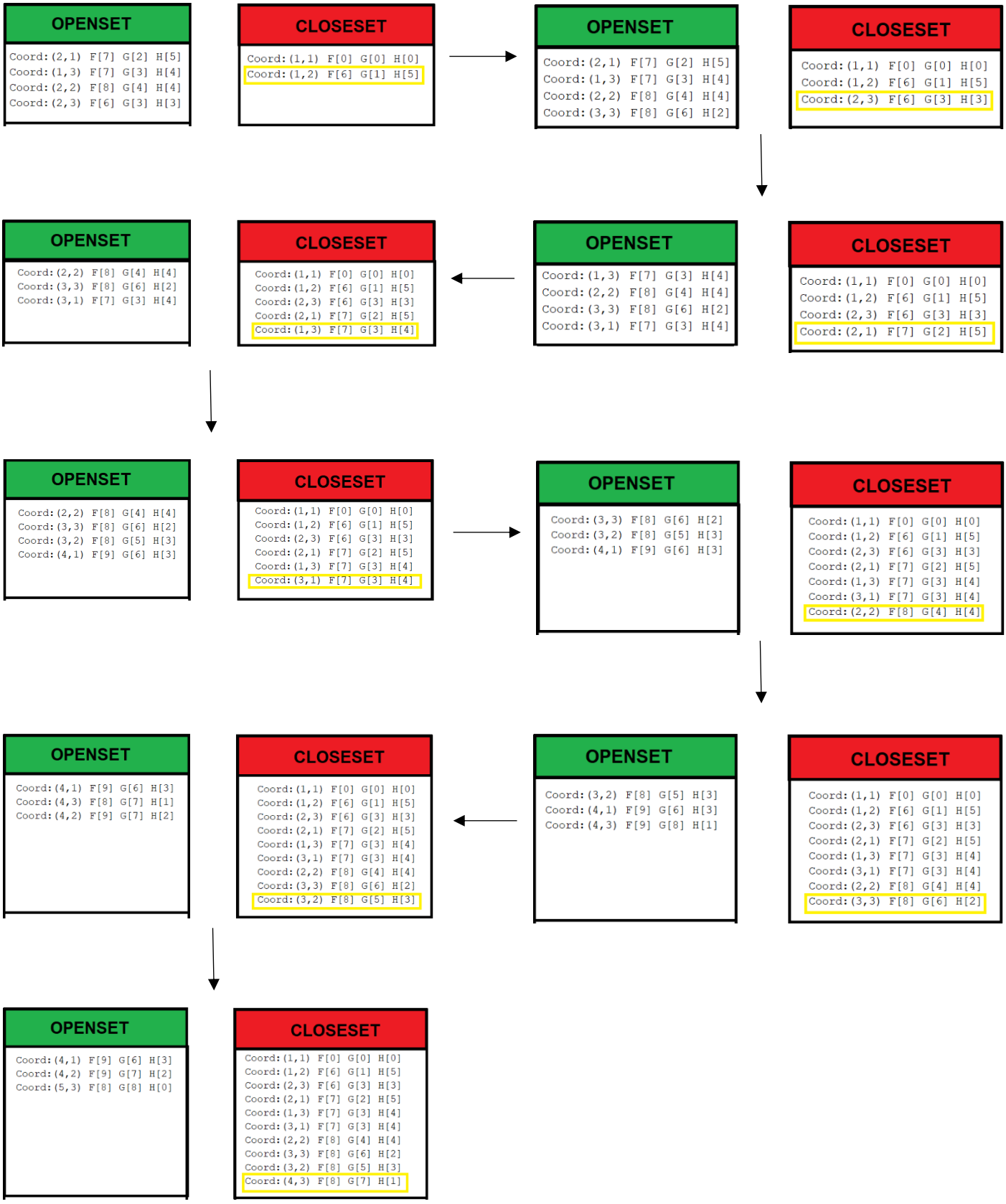
como el nodo anterior resultaba ser el inicial, su coste G era de 0, esto sumado con el coste del nodo correspondiente a la coordenada (2,1) cuyo material es hierba, tiene un coste de 2 en total. Hecho esto en este punto nos encontramos con esta distribución de nodos:

| OPENSET |
| --- |
| (2,1) [F:7, G:2, H:5]<br>(1,2) [F:6, G:1, H:5] |

| CLOSESET |
| --- |
| (1,1) [F:0, G:0, H:0] |

Destacaré en amarillo los nodos que han resultado más prometedores y finalmente traspasados en la misma iteración a la estructura closeSet.

Ahora que ya se ha explicado la dinámica general de ambas estructuras y el funcionamiento en general del algoritmo, expondré en un diagrama los diferentes estados de estas listas en cada iteración hasta que lleguemos a la meta donde volveré a detenerme para explicar lo que sucede en esa casuística.




Tras todas estas iteraciones podemos comprobar que en openSet el nodo que va a ser escogido en la siguiente iteración corresponde con la coordenada (5,3), la cual si empleamos la plantilla que facilité al comienzo del estudio de la traza vemos que coincide con el dragón. Por ello, el procedimiento que va a llevar a cabo el algoritmo cuando lo analice será el siguiente:

- Comunicará que ha encontrado el nodo final y, como consecuencia existe solución al problema propuesto
- Reconstruirá el camino resultante mediante recursividad.
- Finalizará el bucle.

# 4. Heurísticas

## 4.1. Heurísticas estudiadas

### 4.1.1. Heurística sin definir

$$\boldsymbol{h = 0}$$

Esta heurística, si se puede denominar como tal, consiste en omitir dicho juicio de valor que por definición aporta esta parte del algoritmo. Como se puede observar al tratarse de la más trivial también posee la implementación más sencilla.

La principal problemática de esta es que al realizar un análisis por niveles la cantidad de nodos expandidos es mucho mayor, y aunque esto no afecte de forma sistemática a la ejecución de nuestros mapas esta implementación podría resultar inviable dependiendo del tamaño del mapa.

### 4.1.2. Distancia Manhattan

$$\boldsymbol{h = |x_2 - x_1| + |y_2 - y_1|}$$

La distancia Manhattan como representación de una heurística es característica ante todo por aplicarse a entornos cuatro-conectados, es decir, aquellos en los que los nodos poseen otros 4 adyacentes u 8 si se considera posible el movimiento en diagonal. Esto ya podemos prever que va a provocar cálculos erróneos en la distancia entre dos nodos hexagonales; sin embargo, aunque si bien es cierto que no es la más apropiada nos ofrece una métrica incremental y estable que nos servirá para dotar al algoritmo de esa intuición.

Esta heurística no es asumible debido a que en ciertos mapas no ofrece la solución correcta.

### 4.1.3. Distancia euclídea

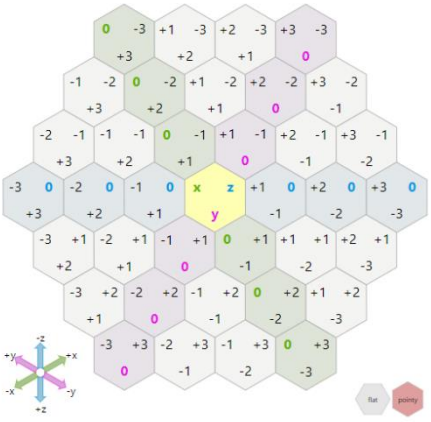
$$\boldsymbol{h = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

La distancia euclídea por su parte determina el espacio existente entre dos puntos, en este caso en un entorno bidimensional. Este cálculo como se puede deducir de la fórmula está estrechamente relacionado con el Teorema de Pitágoras.

### 4.1.4. Distancia en entornos hexagonales

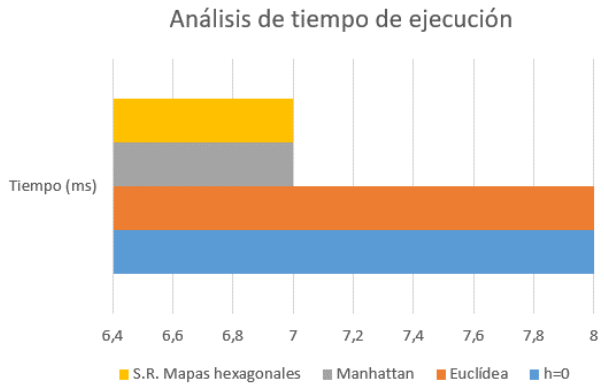
$$\boldsymbol{h = \frac{|x_1 - x_2| + |y_1 - y_2| + |(-x_1 - y_1) - (-x_2 - y_2)|}{2}}$$

Otra forma de ver las cuadrículas hexagonales es ver que hay *tres* ejes primarios, a diferencia de los *dos* que tenemos para las cuadrículas cuadradas. Además, puesto que no tenemos algoritmos obvios para cuadrículas hexadecimales, pero tenemos algoritmos para cuadrículas de cubos nos permite convertirlos para trabajar en cuadrículas hexagonales.



## 4.2. Comparativa

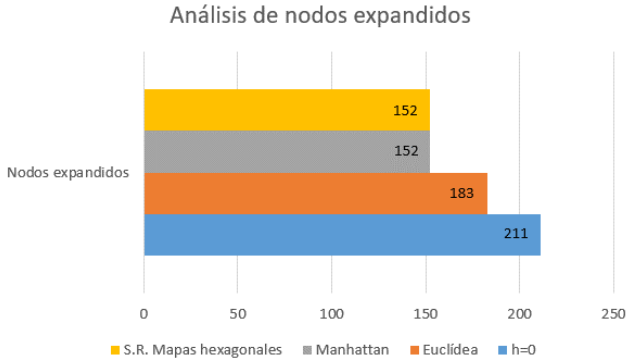
### 4.2.1. Tiempo




De la gráfica resultante al estudio del tiempo de ejecución del algoritmo con cada una de las heurísticas aplicadas se pueden obtener escasas conclusiones, ya que no podemos afirmar una mejora lineal comparando unas sobre otras.
Pero sí queda claro que, por motivos obvios destaca en eficiencia el S.R. de mapas hexagonales y, como se justificará posteriormente la heurística implementada a partir de la distancia Manhattan.

Además, como se ha comentado en apartados anteriores, a la vista de los resultados podríamos considerar tanto a la distancia euclídea como a la h=0 heurísticas inviables para la resolución de problemas exponencialmente mayores.

### 4.2.2. Nodos expandidos




Como vemos este gráfico si ofrece ya una información un poco más detallada, y que coincide en su mayoría con los resultados que esperaba obtener de la ejecución del programa, presentado una mejora lineal de los nodos expandidos y, por tanto, reduciendo el coste computacional con cada una de las heurísticas que hemos ido proponiendo en el anterior apartado.

Sin embargo, hay algo que en primera instancia me llamó la atención pero que identifiqué como un error en la implementación del código, se trata de que a la vista de los resultados la distancia Manhattan resulta tan eficaz como la heurística implementada específicamente para entornos hexagonales. Muy lejos de tratarse de un fallo mío tras investigar concluí lo siguiente.

Aunque la distancia euclídea se matemáticamente más precisa que la Manhattan esta resulta mucho menos eficaz en la práctica a la hora de trazar recorridos entre 2 puntos como se puede apreciar en la imagen (donde se encuentran representadas la ruta euclídea en azul y la manhattan en rojo). Y por ello obtiene menores tiempos y nodos expandidos.

### 4.2.3. Recorrido

En cuanto al recorrido es el último de los aspectos en los que he podido determinar que afecta la implementación de la heurística. Esto es coherente ya que la h( ) es un operando perteneciente a la f( ) y esta afectará de forma colateral a la elección de un nodo prometedor de forma que pueda ser seleccionado antes o después de lo que lo haría con respecto a las demás.

Por todo ello adjunto los caminos devueltos por el algoritmo A* para cada una de estas heurísticas.

h = 0

Distancia Manhattan

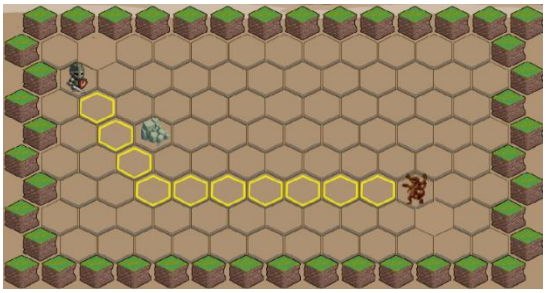
Distancia euclídea

Sistema de referencia hexagonal

# 5. Cuestiones

## 5.1. ¿Se incluye el nodo inicial y el final en el camino?

El nodo inicial no debe de ser tenido en cuenta en el camino ya que el caballero no requiere de ningún esfuerzo para desplazarse hasta su propia posición, pero si para trasladarse al resto de casillas incluyendo la del propio dragón, correspondiente con el nodo final. Esto se puede comprobar en una de las salidas por consola que ofrece nuestro programa.



```
Camino
. . . . . .
. . . . . .
. . . . . .
. . X . . . .
. . X . . . .
. . . X . . .
. . . X X X X X X X . . .
. . . . . .
. . . . . .
. . . . . .
```

## 5.2. ¿El camino explorado empieza por 0 o por 1?

El camino explorado comienza en 0 debido a que al caballero no realiza ningún esfuerzo en trasladarse a su propia localización, sin embargo, el valor final de dicho camino podrá variar dependiendo del entorno sobre el que se ejecute el algoritmo.

## 5.3. ¿Podemos evitar que A* se recalcule en cada turno?

Evidentemente podemos evitar que A* se recalcule cada vez que el jugador introduzca una piedra en el entorno para reducir el coste computacional de la ejecución del juego. La metodología mas sencilla de implementar pero que, sin embargo, resultaría en una optimización notable de este aspecto sería llevar acabo la comprobación de si tras el reposicionamiento aleatorio de las dos entidades ambas se encuentran sobre un nodo que formaba parte, en la iteración anterior, del camino óptimo, ya que simplemente podríamos reducir este en los nodos residuales que se hallen en los extremos obteniendo el nuevo camino con una complejidad constante.

## 5.4. ¿De qué manera podríamos dotar de inteligencia al dragón?

Este aspecto resulta relativamente sencillo razonarlo ya que por la misma regla que el camino toma como nodo inicial al caballero, y al dragón como el final este recorrido es bidireccional y viable en ambos sentidos.

Por lo que, una vez hallado el camino resultante del algoritmo, únicamente deberíamos hallar el nodo anterior del camino antes del dragón, que será a donde debe moverse él mismo para acercarse de manera 'consciente' al caballero.

Como las consecuencias de esta implementación obtendríamos en mi opinión una dinámica mucho más pulida y que incrementaría a dificultad de forma satisfactoria.

## 5.5. ¿Cuál es tamaño máximo obteniendo tiempos asequibles?

En lo que respecta a esta cuestión no me ha sido posible determinar el tamaño del entorno a partir del cual los tiempos empleados para el cálculo de A* sean lo suficientemente elevados como para resultar injugable. Sin embargo, y apoyándome en los tiempos obtenidos en el estudio de las distintas heurísticas este tamaño máximo variará dependiendo de la que empleemos, pero por regla general estimo que con un entorno aproximadamente 7-8 veces mas extenso ya sería costoso.

# 6. Conclusiones

Comenzando por los aspectos positivos me gustaría destacar primeramente el lenguaje de programación que se ha impuesto, es decir JAVA ya que lo encuentro totalmente acorde con los requerimientos de la práctica, además es de agradecer que se trate concretamente del mismo ya que se instruye con anterioridad a los estudiantes del grado respecto a este POO, teniendo una base lo suficientemente estable para encontrarnos cómodos. Relacionado con este punto también me gustaría destacar que, aunque resulte irrelevante, el trabajar con NetBeans también me parece un acierto puesto que es uno de los IDE más utilizados en la actualidad.

Otro de los aspectos que considero acertados es la elección del algoritmo y la presentación de este a modo de reto jugable, lo que seguramente haya animado a muchos estudiantes a prestar atención e interesarse tanto por las clases prácticas como por los conocimientos teóricos.

De forma secundaria me gustaría destacar también mi agrado con respecto a la elección del mapa con nodos hexagonales ya que, creo que es un elemento diferenciador respecto a la abundancia de documentación que existe en cuanto a entornos cuadrados, pero sin ser un aspecto tan difícil como para que dejásemos de lado el verdadero problema a resolver, la implementación del algoritmo A*.

En cuanto a los aspectos negativos, desde la experiencia personal únicamente podría destacar la documentación. Es cierto que la documentación es un mecanismo óptimo para comprobar si un alumno realmente ha comprendido el funcionamiento del algoritmo y que, conlleva al mismo a implicarse activamente a la hora de desarrollar su propio código y evitar copias. Sin embargo, desde la perspectiva de un alumno que lleva implementando y mejorando este desde la primera semana, es cierto que puede resultar muy denso y engorroso plasmar de nuevo todas las reflexiones y valoraciones que ya ha llevado a cabo en el código fuente.

En definitiva, lo que intento expresar es que la elaboración de una documentación resulta mucho más trabajosa para una persona que realmente ha estado implicada con la asignatura que para otra que se halla puesto en la última semana, y estos matices (como saber depurar la información y condensar lo más relevante en vez de redactar 20 páginas como recurso para enmascarar tu falta de conocimiento sobre la materia) pueden pasar desapercibidos a la hora de valorar el trabajo de cada uno de ellos, lo que realmente sería desfavorable.

Para finalizar me gustaría destacar que con todo me parece sobresaliente el planteamiento práctico de esta asignatura y, que nunca he formado parte de un equipo de profesorado y, por tanto, mi opinión puede carecer de valor. Sin embargo, valoro la posibilidad de que el alumnado pueda expresar de esta forma sus diferentes puntos de vista.

# 7. Bibliografía


EcuRed. (s. f.). *Algoritmo de Búsqueda Heurística A* - EcuRed*. https://www.ecured.cu/Algoritmo_de_B%C3%BAsqueda_Heur%C3%ADstica_A*

*Red Blob Games: Hexagonal Grids*. (s. f.). Red Blob Games. https://www.redblobgames.com/grids/hexagons/

*Búsqueda de caminos en los Videojuegos: Algoritmo A* (Estrella)*. (2019, 26 diciembre). [Vídeo]. YouTube. https://www.youtube.com/watch?v=1gszEk8rUS4

*Algoritmo A* (estrella) en HTML5 y JavaScript Tutorial paso a paso*. (2020, 2 enero). [Vídeo]. YouTube. https://www.youtube.com/watch?v=NWS-_VsMab4&ab_channel=Programaresincre%C3%ADble

gammafp. (2017, 27 julio). *YouTube* [Vídeo]. YouTube. https://www.youtube.com/watch?v=X-5JMScsZ14&feature=youtu.be

Grima, C. (2017, 4 septiembre). *Las distancias en Manhattan*. Revista Mètode. https://metode.es/revistas-metode/secciones/cajon-de-ciencia/les-distancies-a-manhattan.html