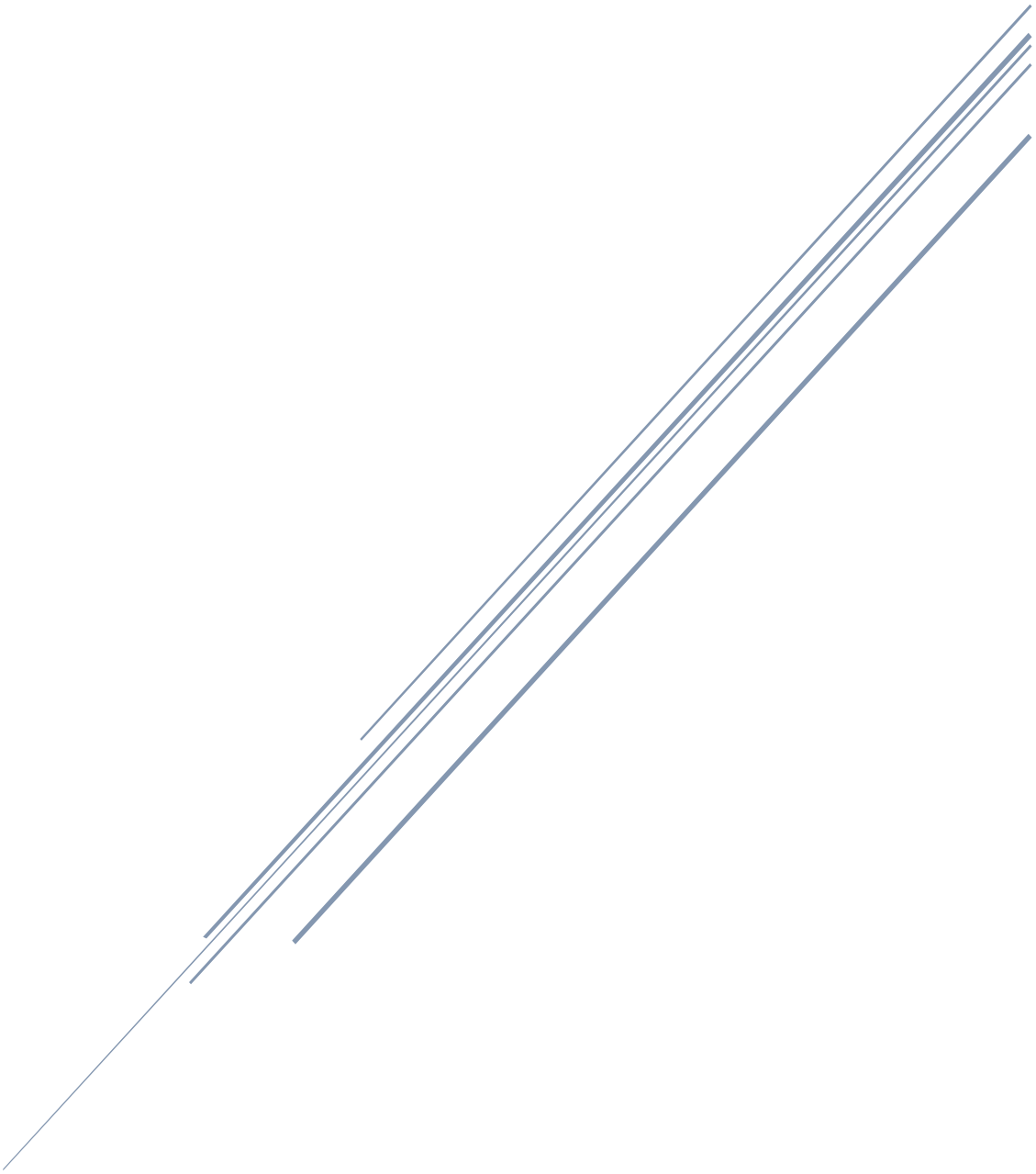
SISTEMAS INTELIGENTES



Raúl Beltrán Marco
23900664 F

ÍNDICE

1. INTRODUCCIÓN

En la siguiente memoria se explicará el funcionamiento de algoritmo A* y su aplicación en el juego del Dragón y el Caballero, un tablero hexagonal que está compuesto por 10 filas y 14 columnas llenas de casillas hexagonales.

El objetivo del juego es conseguir que el Dragón escape del Caballero, para ello, hemos implementado el algoritmo A* el cual creará un camino para el Caballero con el cual poder llegar al Dragón. El juego acaba cuando no hay camino posible o cuando el coste de este camino es igual a 20.

Además, realizaremos una traza en la que se podrá comprender a la perfección cómo funciona el algoritmo A*.

También probaremos distintas heurísticas, Manhattan, Euclídea y Mapas Hexagonales, para averiguar cuál es la mejor para este tipo de mapa y el como han sido implementadas en el código, es necesario mencionar que esta práctica ha sido realizada en Java, haciendo uso de la aplicación NetBeans.

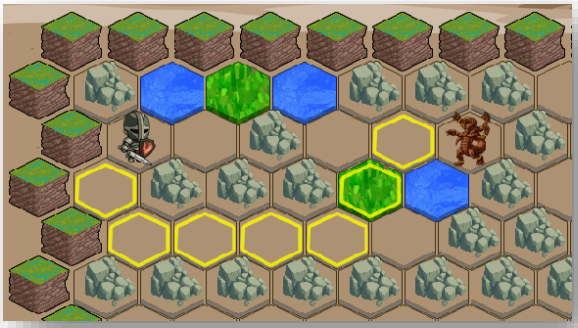
2. EXPLICACIÓN Y TRAZA

El Algoritmo A* se basa en llegar desde una estado o nodo **origen** a una estado o nodo **meta,** mediante un **camino óptimo** como solución. Para llegar a esta meta, el algoritmo analiza los nodos en base a las siguientes variables que tiene cada nodo:

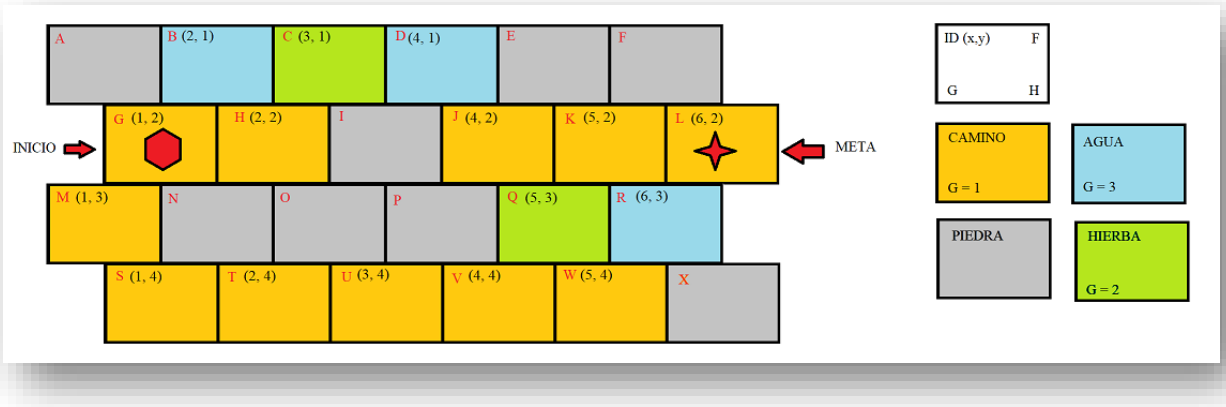
- **G** = Lo que cuesta pasar de un estado a otro.
- **H** = Es la distancia que existe desde un nodo cualquiera al nodo meta, también sirve de heurística para el algoritmo. (Manhattan, Euclídea, mapas hexagonales, …)
- **F** = Es la suma de las variables G y H.

Después de analizar un cierto número de nodos y teniendo en cuenta los criterios que se especificarán en la traza, el algoritmo es capaz de reconstruir un camino óptimo con el cual llegar a la meta.

A continuación, haremos una traza en la cual explicaremos el funcionamiento del algoritmo, para ello usaremos el siguiente mapa, el cual está hecho para generar un camino corto con el que poder hacer una traza corta en la que veremos la mayoría de los criterios que sigue el algoritmo.



Una versión simplificada del mismo mapa, para que sea más fácil hacer a continuación la traza.



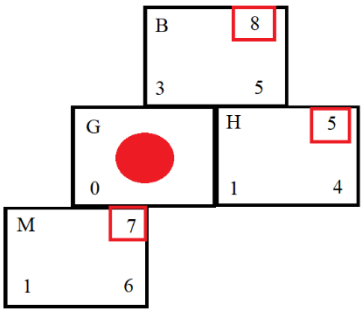
La traza se hará mediante iteraciones y de forma esquematizada para un mayor entendimiento.

Lo primero es antes de comenzar el bucle y las iteraciones hay que inicializar las listas que vamos a utilizar durante el algoritmo: **ListaFrontera** (LF) y **ListaInterior** (LI), en la LI estará la posición inicial, G y en LF estarán los "hijos" / nodos adyacentes de G.

LF = {B, H, M} y LI = {G}

Ahora dentro del bucle, lo próximo que haremos será calcular los valores ***G, H* y *F*** de los hijos de G.

Para calcular la ***H***, usaremos la **Distancia Manhattan** ($|x_2 - x_1| + |y_2 - y_1|$) y habrá que tener en cuenta que para la ***G*** hay que sumar la ***G*** del propio nodo sumado a la ***G*** que tenga su "padre", en este caso G y su ***G*** es 0 debido a que es el nodo inicial por tanto obtenemos los siguientes nodos:

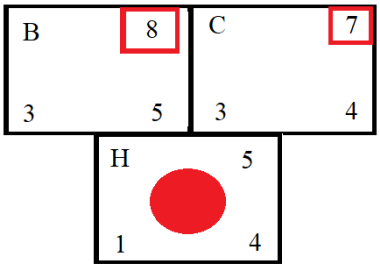


El punto rojo indica donde se encuentra el nodo actual en la iteración del bucle.

1º ITERACIÓN

Se siguen los siguientes pasos:

1. Se elige el nodo con menor *F* de LF → **H.**
2. Se comprueba si algunos de los nodos adyacentes es meta. (En este caso se encuentra en el nodo L).
3. Como ninguno es meta, se genera una lista auxiliar con los hijos de H: **hijosH = {B, C} ***
    - Los valores de B se mantienen porque ir de H a B cuesta MÁS que ir de G a B. (Se tiene en cuenta el valor de *G*)
    - Como C no se encuentra en ListaFrontera, hay que añadirlo.



Ahora las listas han quedado de la siguiente forma:

LF = {B, M, C} y LI = {G, H}

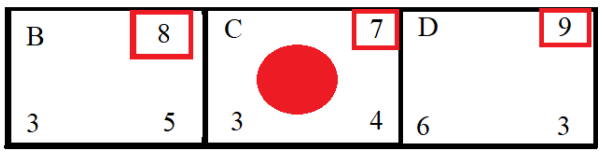
*Como se puede observar, a la hora de generar los hijos de un nodo se omiten los límites del mapa, las piedras y los nodos que ya están incluidos en ListaInterior.

2º ITERACIÓN

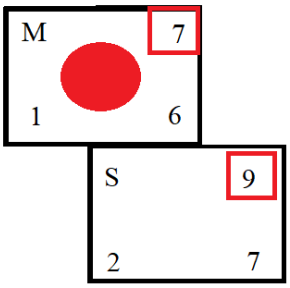
1. Se elige el nodo con menor *F* de LF → **C.**
2. Comprobamos si algún adyacente es meta.
3. Ninguno es meta → Generamos hijos de C: hijosC = {B, D}
    - Mismo razonamiento que antes, ir de C a B es cuesta MÁS que ir de G a B, entonces no se modifica
    - Como D no se encuentra en LF, lo añadimos.

Ahora las listas han quedado de la siguiente forma:

LF = {B, M, D} y LI = {G, H, C}



3º ITERACIÓN



1. Se elige el nodo con menor *F* de LF → **M.**
2. Comprobamos si algún adyacentes es meta.
3. Ninguno es meta → Generamos hijos de M: hjosM = {S}
    - Como S no se encuentra en LF, lo añadimos.

Ahora las listas han quedado de la siguiente forma:

LF = {B, D, S} y LI = {G, H, C, M}

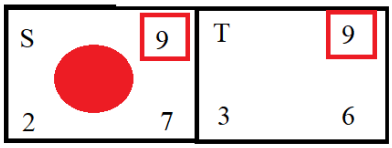
4º ITERACIÓN

1. Se elige el nodo con menor *F* de LF → **B.**
2. Comprobamos si algún adyacentes es meta.
3. Ninguno es meta → Generamos hijos de B, pero como no tiene hijos. (Está rodeado de nodos ya incluidos en ListaInterior, no se tienen en cuenta)

Así que en este caso el único cambio que se realiza es en ListaInterior y obtenemos las siguientes listas:

LF = {D, S} y LI = {G, H, C, M, B}

5º ITERACIÓN

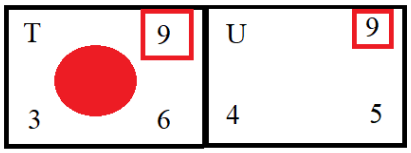


1. Se elige el nodo con menor *F* de LF → **S.** (Tiene el mismo valor que D, pero por la forma de recorrer el bucle elegimos S)
2. Comprobamos si algún adyacentes es meta.
3. Ninguno es meta → Generamos hijos de S: hjosS = {T}
    - Como T no se encuentra en LF, lo añadimos.

Ahora las listas han quedado de la siguiente forma:

LF = {D, T} y LI = {G, H, C, M, B, S}

6º ITERACIÓN

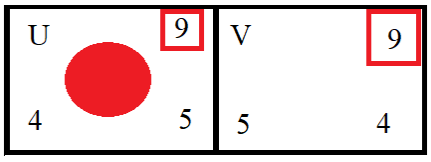


1. Se elige el nodo con menor *F* de LF → **T.** (Tiene el mismo valor que D, pero por la forma de recorrer el bucle elegimos T)
2. Comprobamos si algún adyacentes es meta.
3. Ninguno es meta → Generamos hijos de T: hjosT = {U}
    - Como T no se encuentra en LF, lo añadimos.

Ahora las listas han quedado de la siguiente forma:

LF = {D, U} y LI = {G, H, C, M, B, S, T}

7º ITERACIÓN

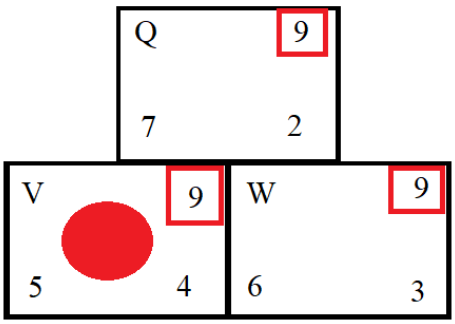


1. Se elige el nodo con menor *F* de LF → **U.** (Tiene el mismo valor que D, pero por la forma de recorrer el bucle elegimos U)
2. Comprobamos si algún adyacentes es meta.
3. Ninguno es meta → Generamos hijos de U: hjosU = {V}
    - Como T no se encuentra en LF, lo añadimos.

Ahora las listas han quedado de la siguiente forma:

LF = {D, V} y LI = {G, H, C, M, B, S, T, U}

8º ITERACIÓN

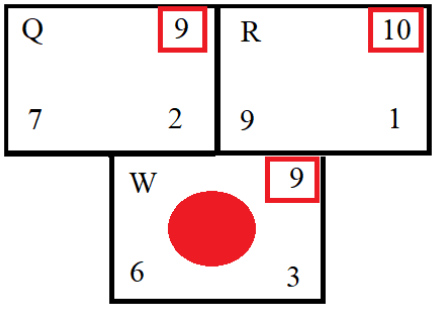


1. Se elige el nodo con menor *F* de LF → **V.** (Tiene el mismo valor que D, pero por la forma de recorrer el bucle elegimos V)
2. Comprobamos si algún adyacentes es meta.
3. Ninguno es meta → Generamos hijos de V: hjosV = {Q, W}
    - Como Q no se encuentra en LF, lo añadimos.
    - Como W no se encuentra en LF, lo añadimos.

Ahora las listas han quedado de la siguiente forma:

LF = {D, Q, W} y LI = {G, H, C, M, B, S, T, U, V}

9º ITERACIÓN

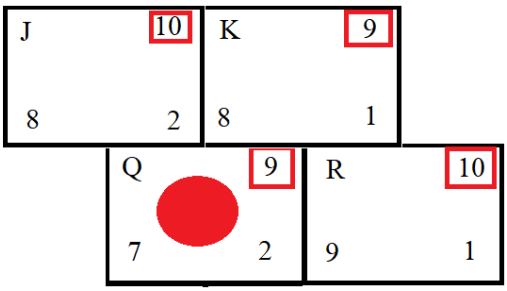


1. Se elige el nodo con menor *F* de LF → **W.** (Tiene el mismo valor que D y Q, pero por la forma de recorrer el bucle elegimos W)
2. Comprobamos si algún adyacentes es meta.
3. Ninguno es meta → Generamos hijos de W: hjosW = {Q, R}
    - Como ir de W a Q cuesta MÁS que ir de V a Q, Q no se modifica.
    - Como R no se encuentra en LF, lo añadimos.

Ahora las listas han quedado de la siguiente forma:

LF = {D, Q, R} y LI = {G, H, C, M, B, S, T, U, V, W}

10º ITERACIÓN



1. Se elige el nodo con menor *F* de LF → **Q.** (Tiene el mismo valor que D, pero por la forma de recorrer el bucle elegimos Q)
2. Comprobamos si algún adyacentes es meta.
3. Ninguno es meta → Generamos hijos de Q: hjosQ = {J, K, R}
    - Como ir de Q a R cuesta MÁS que ir de W a R, R no se modifica.
    - Como J no se enceuntrá en LF, lo añadimos.
    - Como K no se encuentra en LF, lo añadimos.

Ahora las listas han quedado de la siguiente forma:

LF = {D, J, K, R} y LI = {G, H, C, M, B, S, T, U, V, W, Q}

11º ITERACIÓN

1. Se elige el nodo con menor *F* de LF → **K.** (Tiene el mismo valor que D, pero por la forma de recorrer el bucle elegimos K)
2. Comprobamos si algún adyacentes es meta.
3. Un adyacente es meta, lo que significa que hemos encontrado una solución y acaba el bucle además de generar el camino óptimo correspondiente.

2. PREGUNTAS PROPUESTAS

- **¿Se incluye el nodo inicial y el final en el camino?**

El camino que genera el algoritmo sólo incluye el nodo inicial debido a que la condición de parada de este es que algún nodo adyacente sea meta, por tanto ese nodo adyacente no se llega a tener en cuenta en el programa.

- **¿Por tanto se deben mostrar con X en la solución?**

Sí, ya que el objetivo del algoritmo es encontrar un camino completo y óptimo, en caso de la posición meta es otra función externa la que la marca como X.

- **¿El camino explorado empieza por 0 o por 1?**

El camino explorado empieza por 0 debido a que a la casilla donde se encuentra el caballero le asignamos una *G* de valor 0 al ser la incial.

3. ANÁLISIS DE LAS DISTINTAS HEURÍSTICAS

En este apartado haremos un análisis comparativo entre las heurísticas de Manhattan, Euclídea y distancias adaptadas a mapas hexagonales. Para ello analizaremos el número de nodos expandidos (a menor número de nodos expandidos más eficiente es el algoritmo) que genera el algoritmo con las distintas heurísticas.

Las heurísticas han sido implementadas en funciones auxiliares de esta manera para variar el valor de la variable *H* de cada nodo.

```
private int Manhattan(Coordenada actual, Coordenada meta)
{
    int x= Math.abs(meta.getX() - actual.getX());
    int y= Math.abs(meta.getY() - actual.getY());
    return x + y;
}
private int Euclidea(Coordenada actual, Coordenada meta)
{
    int x= (int)Math.pow(meta.getX() - actual.getX(), 2);
    int y= (int)Math.pow(meta.getY() - actual.getY(), 2);
    return (int)Math.sqrt(x + y);
}
private int mapaHexagonal(Coordenada actual, Coordenada meta)
{
    Cubo origen= axial_to_cube(actual);
    Cubo dragon= axial_to_cube(meta);
    return origen.distancia(dragon);
}
public Cubo axial_to_cube(Coordenada coordenada)
{
    int x,y,z;
    x= coordenada.getY() - (coordenada.getX() + (coordenada.getX()&1)) / 2;
    z= coordenada.getX();
    y= -x - z;
    return new Cubo(x, y, z);
}
```

Ha sido necesario crear una clase “Cubo” para poder almacenar las variables x, y, z y calcular la distancia entre dos coordenadas cúbicas.

```
public class Nodo {
    Coordenada coordenada;
    Nodo padre;
    Mundo mundo;
    int g;
    int f;
    int h;
    public class Cubo {
        int x;
        int y;
        int z;

        public Cubo()
        {
            this.x= this.y= this.z= 0;
        }
        public Cubo(int x, int y, int z)
        {
            this.x= x;
            this.y= y;
            this.z= z;
        }
        public int getX()
        {
            return x;
        }
        public int getY()
        {
            return y;
        }
        public int getZ()
        {
            return z;
        }
        public int distancia(Cubo b)
        {
            return (Math.abs(this.x - b.x) + Math.abs(this.y - b.y) + Math.abs(this.z - b.z)) / 2;
        }
    }
```

Después de realizar pruebas en distintos mapas, hemos obtenido la siguiente tabla comparativa:

	mundo 01		mundo defecto		camino unico		invertido vacio		invertido defecto	
	Nº NODOS	COSTE TOTAL	Nº NODOS	COSTE TOTAL	Nº NODOS	COSTE TOTAL	Nº NODOS	COSTE TOTAL	Nº NODOS	COSTE TOTAL
MANHATTAN	11	11	51	17	11	11	25	11	48	17
EUCLÍDEA	24	11	47	16	11	11	36	11	62	17
MAPAS HEXAGONAL	21	11	46	16	11	11	25	11	58	16

*En rojo se ha marcado el menor nº de nodos y coste total de sus respectivas columnas.

Como podemos observar, en el caso en el que sólo hay un camino posible por el cual llegar a la meta da igual la heurística que empleemos porque todas nos van a dar el mismo resultado.

Es necesario recalcar el caso de *mundo 01* e *invertido vacio*, ya que se tratan del mismo mapa vacío pero invirtiendo la posición del caballero y el dragón como si fuera el reflejo de un espejo. Hemos obtenido que en el caso de que el caballero haga un camino de izquierda a derecha la mejor heurística es **Manhattan**, cuando en *invertido vacio* empatan tanto Manhattan como Mapas Hexagonales.

En cambio en el caso de *invertido defecto* y *mundo defecto* la heurística de Manhattan no es admisible debido a que ignora el camino más óptimo e incluso en *mundo defecto* explora más nodos incluso.

También podemos destacar que en el mundo defecto, un mundo en el que hay que distintos caminos y obstáculos para llegar a la meta, la mejor heurística tanto en número de nodos explorados y coste total del camino es la de **Mapas Hexagonales**.

Por tanto, podemos realizar la siguiente clasificación respecto a que heurística es la más recomendable para mapas hexagonales:

1. **Mapas Hexagonales:** Suele explorar más nodos que la Manhattan, pero siempre obtiene una coste total igual o inferior a Manhattan, es más importante un menor coste total que un menor número de nodos explorados.
2. **Manhattan:** Excepto en el caso de invertido vacío, suele obtener un coste total mayor o igual que las otras dos heurísticas, a pesar de que explore menos nodos que el resto, no es capaz de obtener el camino más óptimo.
3. **Euclídea:** Muy empatada con la Manhattan, pero debido a varios casos en el que el coste total es muy parecido a la Manhattan y que sea el peor a la hora de encontrar un camino para mapas que se recorren de derecha a izquierda, (37 y 63 nodos explorados en los dos últimos mapas) la convierten en la peor heurística de estas tres para mapas hexagonales.