2018

Práctica 1. Búsqueda heurística

SISTEMAS INTELIGENTES

PEDRO GIMÉNEZ ALDEGUER

Clase:Viernes (09:00-11:00)

DNI: 15419933C

UNIVERSIDAD DE ALICANTE | Departamento de Ciencia de la Computación e Inteligencia Artificial

Práctica 1:

1. Algoritmo A*:

La definición del Algoritmo A* de Wikipedia es el siguiente: "El algoritmo de búsqueda A* se clasifica dentro de los algoritmos de búsqueda en grafos. Presentado por primera vez en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael, el algoritmo A* encuentra, siempre y cuando se cumplan unas determinadas condiciones, el camino de menor coste entre un nodo origen y uno objetivo."

El algoritmo A* utiliza una función de evaluación:

$$\boxed{\mathbf{A^*: f^*(n) = g^*(n) + h^*(n)}}$$

• $g^*(n) = c(s, n)$:

- Coste del camino de coste mínimo desde el nodo inicial s al nodo n.

- Estimada por g(n).

• $h^*(n)$:

- Coste del camino de coste mínimo de todos los caminos desde el nodo n a cualquier estado solución tj.

- Estimada por h(n).

• $f^*(n)$:

- Coste del camino de coste mínimo desde el nodo inicial hasta un nodo solución condicionado a pasar por n.

- $f^*(n) = g^*(n) + h^*(n)$.

- Estimada por f(n).

• C^*:

- Coste del camino de coste mínimo desde el nodo inicial a un nodo solución.

Como todo algoritmo de búsqueda en amplitud, A* es un algoritmo completo: en caso de existir una solución, siempre dará con ella.

Para garantizar la optimización del algoritmo, la función h(n) debe ser heurística admisible, esto es, que no sobrestime el coste real de alcanzar el nodo objetivo. Cuanto más correctamente estimemos h(n) menos nodos de búsqueda generaremos.

Una función heurística h(n) se dice que es admisible (garantiza la obtención de un camino de coste mínimo hasta un objetivo) cuando se cumple:

$$\boxed{\mathbf{h(n) \leq h^*(n)\ \forall n}}$$

Si nuestra función heurística nos devuelve un valor superior a h*, para algún nodo, no se puede garantizar que encontremos la solución óptima.

2. Pseudocódigo A*:

Ha continuación muestro el pseudocódigo del algoritmo de búsqueda A*, además de enseñar parte de mi código para cada funcionalidad del Algoritmo.

Pseudocódigo

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

      mientras listaFrontera no esté vacía

            n = obtener nodo de listaFrontera con menor f(n) = g(n) + h(n)

            si n es meta devolver
                  reconstruir camino desde la meta al inicio siguiendo los
                  punteros
                  Salir
            sino
                  listaFrontera.del(n)
                  listaInterior.add(n)

                  para cada hijo m de n que no esté en lista interior
                        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
      Error, no se encuentra solución
falg
```

- Clase Nodo:

Para mayor comodidad he creado una clase Nodo para almacenar los datos necesarios de cada nodo explorado.

La clase Nodo tiene como atributos la coordenada, el valor de su heurística, su nodo padre anterior, su coste del nodo inicial al nodo actual (g) y la suma de g y h que es f.

```
class Nodo{
    private Nodo papa;
    private Coordenada n;
    private int f, g, h;
```

En el constructor del nodo le paso como parámetro el coste de la casilla del nodo actual para poder calcula la g, e inicializo cada atributo con los parámetros necesarios.

```
public Nodo(Coordenada n, int h, Nodo p, int coste){
    if(p == null){
        this.g = coste;
    }
    else{
        this.g = coste + p.getG();
    }
    this.f = g + h;
    this.h = h;
    this.n = n;
    papa = p;
}
```

- Explico el inicio y lo básico:

Creo dos ArrayList para ListaInterior y para ListaFrontera, además de guardar en coordenadas al caballero y al dragón.

- ✓ **ListaInterior** para almacenar los nodos ya evaluados y son aptos para la creación del camino.
- ✓ **ListaFrontera** para almacenar los nodos que son adyacentes y aptos para evaluaros.

```
ArrayList<Nodo> listaInterior = new ArrayList<Nodo>();
ArrayList<Nodo> listaFrontera = new ArrayList<Nodo>();

//Guardo las coordenadas de caballero y el dragón
Coordenada c = mundo.getCaballero();
Coordenada d = mundo.getDragon();
```

Añado a ListaFrontera antes de entrar al bucle el nodo del caballero a lista Frontera y hago que el caballero en el camino_expandido sea 0 como incial.

```
int h = recursivo(c, d , 0);

//Añadimos la posicion del caballero en listaFrontera
listaFrontera.add(new Nodo(c, h, null, 0));
camino_expandido[c.getY()][c.getX()] = 0;
```

Nos metemos en el bucle while para calcular el camino hasta encontrar al dragón.

```
//Mientras listaFrontera no este vacia y no se haya encontrado al dragón
while(!listaFrontera.isEmpty() && !encontrado){

    if(c.getX() == d.getX() && c.getY() == d.getY()){
        return result;
    }
```

Buscamos el mejor nodo con mejor f:

```
//Busco el nodo con mejor F() para cogerlo y hacer el camino
for(int i = 0; i < listaFrontera.size(); i++){
    Nodo co = listaFrontera.get(i);
    if(co.getF() <= n.getF()){
        n = listaFrontera.get(i);
    }
}
```

Si se encuentra al dragón, dibujo el camino cambiando de padres hasta llegar al padre del caballero que es null, además de igualar el coste_total con result.

```
//Si se ha encontrado el dragón, dibujo el camino y voy cambiando de padres
//(Además añado el coste_total como resultado)
if(n.getN().getX() == d.getX() && n.getN().getY() == d.getY()){
    coste_total = n.getG();
    encontrado = true;
    result = (int)coste_total;
    while(n != null){
        camino[n.getN().getY()][n.getN().getX()] = 'X';
        n = n.getPapa();
    }
}
```

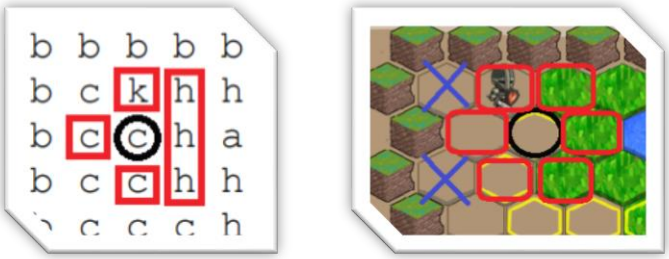
Si el nodo actual con mejor coste de f no es el dragón entonces eliminamos el nodo actual de listaFrontera y lo añadimos a listaInterior porque ya lo hemos evaluado.

```
//Si no se ha encontrado el dragón
else{
    //Eliminamos el nodo de listaFrontera y lo añadimos en listaInterior
    listaFrontera.remove(n);
    listaInterior.add(n);
```

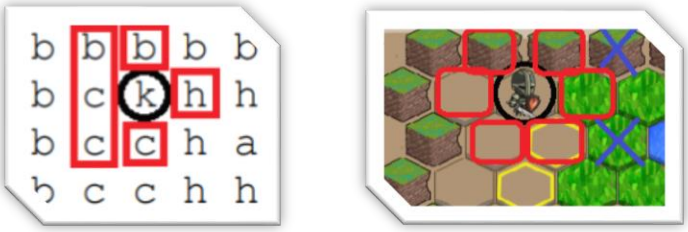
Buscamos los hijos del nodo actual y los evaluamos.

Como la tabla no es cuadrática, sino que es hexagonal, la selección de hijos varía dependiendo de la posición del caballero.

- ✓ Si el nodo actual es Par los hijos que podrá evaluar son los siguientes:



- ✓ Si el nodo actual es Impar los hijos que podrá evaluar son los siguientes:



Para evaluarlos he creado un método aparte llamado “alrededor”:

```
//Posiciones para guardar en listaFrontera
alrededor((new Coordenada(n.getN().getX(), n.getN().getY()+1)), listaInterior, listaFrontera, n, d);
alrededor((new Coordenada(n.getN().getX(), n.getN().getY()-1)), listaInterior, listaFrontera, n, d);
alrededor((new Coordenada(n.getN().getX()+1, n.getN().getY())), listaInterior, listaFrontera, n, d);
alrededor((new Coordenada(n.getN().getX()-1, n.getN().getY())), listaInterior, listaFrontera, n, d);

//Cuando es par la posicion
if(n.getN().getY()% 2 == 0){

    alrededor((new Coordenada(n.getN().getX()+1, n.getN().getY()+1)), listaInterior, listaFrontera, n, d);
    alrededor((new Coordenada(n.getN().getX()+1, n.getN().getY()-1)), listaInterior, listaFrontera, n, d);
}
//Cuando es impar
else{
    alrededor((new Coordenada(n.getN().getX()-1, n.getN().getY()-1)), listaInterior, listaFrontera, n, d);
    alrededor((new Coordenada(n.getN().getX()-1, n.getN().getY()+1)), listaInterior, listaFrontera, n, d);
}
```

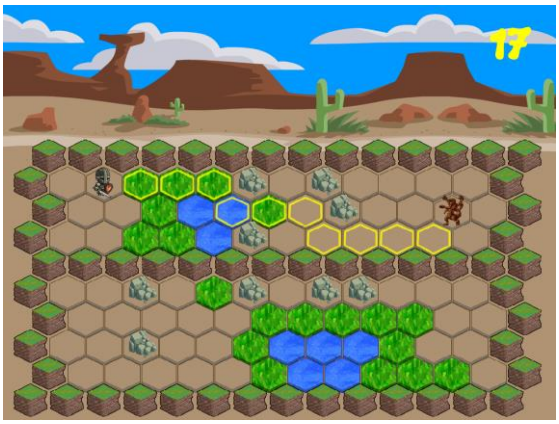
Todo esto se repetirá hasta que hayamos encontrado al dragón o si listaFrontera se queda vacía y no encuentra el dragón. Si el caballero no llega a escapar del dragón entonces perderá igual que sí no encuentra un camino hacia el dragón.

Al ejecutar por primera vez el programa deberá aparecer lo siguiente:

```
Mundo a resolver
 b b b b b b b b b b b b b b
b c k h h h p c p c c c c b
 b c c h a a h c p c c d c b
b c c h h a p c c c c c c b
 b b b b b b b b b b b b b b
b c c p c h p c p p c c c b
 b c c c c c h h h h h c c b
b c c p c c h a a a h c c b
 b c c c c c h a a h h h c b
b b b b b b b b b b b b b b
Camino
 . . . . . .
. . X X X X . . . . . .
 . . . . . X X X . . . X . .
. . . . . X X X X . .
 . . . . . .
. . . . . .
 . . . . . .
. . . . . .
 . . . . . .
. . . . . .
Camino explorado
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1  3  0  2 10 12 -1 18 -1 -1 -1 -1 -1 -1
 -1  4  1  6  8 13 15 16 -1 21 23 25 -1 -1
-1 11  5  7  9 14 -1 17 19 20 22 24 26 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
Nodos expandidos: 26
```

- **Mundo a resolver:** nos aparecerá el mundo que hemos seleccionado. Contiene toda la descripción del entorno.
- **Camino:** indica las celdas que forman parte del camino con una ‘X’.
- **Camino explorado:** indica el orden en el que se expanden las celdas, -1 si no se ha expandido por esa coordenada.
- **Nodos expandidos:** Un entero que indica el total de nodos expandidos.

En la **ventana** del juego aparece lo siguiente:



- Evaluación de los nuevos nodos:

Lo primero que hay que ver es si el hijo ya ha sido evaluado, osea si se encuentra ya en listaInterior. Si es así el hijo queda descartado y no hay que hacer nada con él.

```
private void alrededor(Coordenada c, ArrayList<Nodo> listaI, ArrayList<Nodo> listaF, Nodo n, Coordenada d){
    //int h = 0;
    //int h = Manhattan(c, d);
    //int h = Euclidea(c, d);
    int h = recursivo(c, d , 0);
    boolean es = true;

    //Para cada hijo m de n(padre) que no esté en lista interior
    for(int i = 0; i < listaI.size() && es; i++){
        if(c.getX() == (listaI.get(i).getN().getX()) && c.getY() == (listaI.get(i).getN().getY())){
            es = false;
        }
    }
```

Si no se encuentra en listaInterior entonces es que es necesario evaluarse y hay que calcular el coste de la casilla de ese hijo para calcular más adelante su g.

- Si la casilla es **agua** el coste será 3.
- Si la casilla es **hierba** el coste será 2.
- Si la casilla es **camino** el coste será 1.
- Si la casilla es **dragón** el coste será 1.

El resto, como el **bloque o la piedra** será coste 99 ya que luego se van a descartar y da igual.

```
//Coste de las posiciones g'(m)
if(es){
    int coste;
    switch(mundo.getCelda(c.getX(), c.getY())){
        case('a'):
                coste = 3;
            break;
        case('h'):
                coste = 2;
            break;
        case('c'):
                coste = 1;
            break;
        case('d'):
                coste = 1;
            break;
        default:
                coste = 99;
            break;
    }
```

```
    //Si es piedra o bloque no lo pongo
if(mundo.getCelda(c.getX(), c.getY()) != 'b' && mundo.getCelda(c.getX(), c.getY()) != 'p'){
```

Después de calcular el coste de cada casilla miramos si el hijo que estamos evaluando ya se encuentra en listaFrontera.

Si no es el caso creamos el nodo hijo y lo añadimos a listaFrontera, además de ir aumentando los caminos expandidos.

```
//Mirar si existe en listaF
for(int i = 0; i < listaF.size() && es; i++){
    if(c.getX() == listaF.get(i).getN().getX() && c.getY() == listaF.get(i).getN().getY()){
        es = false;
        posicion = i;
        mejor = listaF.get(i).getG();
    }
}
if(es){
    listaF.add(new Nodo(c, h, n, coste));

    //Vamos añadiendo el camino expandido
    expandidos++;
    camino_expandido[c.getY()][c.getX()] = expandidos;//h;//expandidos;
}
```

Si el hijo se encuentra en listaFrontera, es necesario ver si la g del hijo es menor que el que se encuentra en listaFrontera. Si la g del hijo que estamos evaluando es mejor entonces eliminamos el que estaba en listaFrontera y añadimos el hijo evaluado, para que tenga la mejor g y allá cambiado el padre.

```
else if((coste + n.getG()) < mejor){
    listaF.remove(posicion);
    listaF.add(new Nodo(c, h, n, coste));
}
```

3. Mejor heurística:

- Explicación de las heurísticas utilizadas:

• **Una primera heurística $h = 0$.** Es decir, que la función heurística evaluara con 0 cada celda. Con la cual obtendríamos resultados de una búsqueda con coste uniforme.

• **Distancia Manhattan**. Esta distancia es la suma de las diferencias absolutas de las coordenadas de la celda origen y de la celda destino. Es decir, la distancia entre la celda 1 y la celda 2, sería:

$$|x2 - x1| + |y2 - y1|$$

```
private int Manhattan(Coordenada x, Coordenada y){
    int sol;
            sol = abs(x.getX() - y.getX()) + abs(x.getY() - y.getY());
    return sol;
}
```

• **Distancia Euclídea**. Define la línea recta entre dos puntos. Se deduce a partir del Teorema de Pitágoras y se calcularía la distancia entre la celda 1 y la celda 2 como:

$$\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

```
private int Euclidea(Coordenada x, Coordenada y){
    int sol;
            sol = (int)sqrt(pow((x.getX() - y.getX()),2) + pow((x.getY() - y.getY()),2));
    return sol;
}
```

• **Distancias adaptadas a mapas hexagonales**. Estas distancias están adaptadas a mapas con casillas hexagonales. Es posible, que para calcular estas distancias sea necesario un cambio de sistema de coordenadas, ya que, en muchas ocasiones, en mapas hexagonales se trabaja con coordenadas cúbicas.

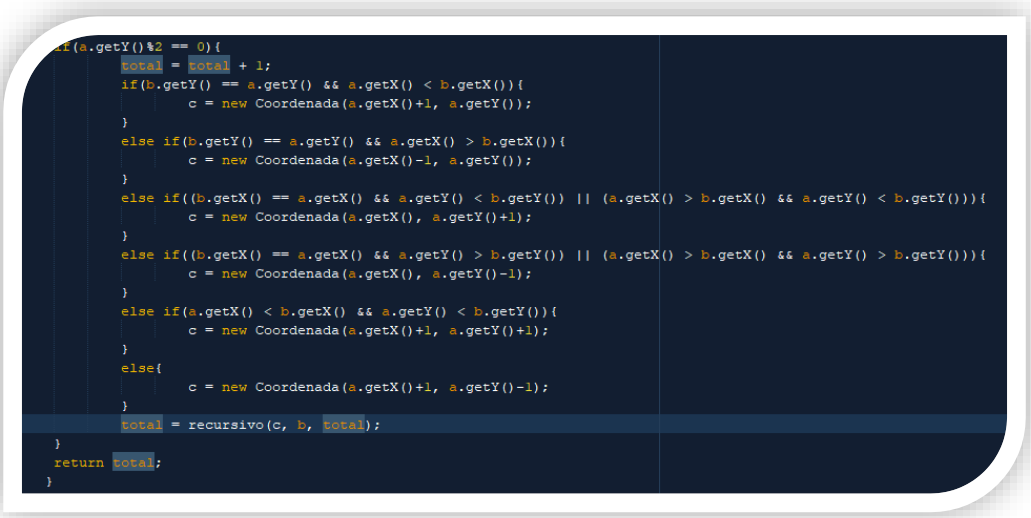
Para mapas hexagonales he hecho un método recursivo que dependiendo de donde esté el dragón recursivamente buscará el nodo actual al dragón aumentando en cada casilla el coste de h.

Si la casilla con la que busca al dragón es impar, hará lo siguiente:

```
ic int recursivo(Coordenada a, Coordenada b, int total){
Coordenada c = new Coordenada();

if(a.getY() == b.getY() && a.getX() == b.getX()){
        return total;
}
if(a.getY()%2 != 0){
        total = total + 1;
        if(b.getY() == a.getY() && a.getX() < b.getX()){
                c = new Coordenada(a.getX()+1, a.getY());
        }
        else if(b.getY() == a.getY() && a.getX() > b.getX()){
                c = new Coordenada(a.getX()-1, a.getY());
        }
        else if((b.getX() == a.getX() && a.getY() < b.getY()) || (a.getX() < b.getX() && a.getY() < b.getY())){
                c = new Coordenada(a.getX(), a.getY()+1);
        }
        else if((b.getX() == a.getX() && a.getY() > b.getY()) || (a.getX() < b.getX() && a.getY() > b.getY())){
                c = new Coordenada(a.getX(), a.getY()-1);
        }
        else if(a.getX() > b.getX() && a.getY() < b.getY()){
                c = new Coordenada(a.getX()-1, a.getY()+1);
        }
        else{
                c = new Coordenada(a.getX()-1, a.getY()-1);
        }
        total = recursivo(c, b, total);
}
```

Si la casilla con la que busca al dragón es par, hará lo siguiente:



```
if(a.getY()%2 == 0){
        total = total + 1;
        if(b.getY() == a.getY() && a.getX() < b.getX()){
                c = new Coordenada(a.getX()+1, a.getY());
        }
        else if(b.getY() == a.getY() && a.getX() > b.getX()){
                c = new Coordenada(a.getX()-1, a.getY());
        }
        else if((b.getX() == a.getX() && a.getY() < b.getY()) || (a.getX() > b.getX() && a.getY() < b.getY())){
                c = new Coordenada(a.getX(), a.getY()+1);
        }
        else if((b.getX() == a.getX() && a.getY() > b.getY()) || (a.getX() > b.getX() && a.getY() > b.getY())){
                c = new Coordenada(a.getX(), a.getY()-1);
        }
        else if(a.getX() < b.getX() && a.getY() < b.getY()){
                c = new Coordenada(a.getX()+1, a.getY()+1);
        }
        else{
                c = new Coordenada(a.getX()+1, a.getY()-1);
        }
        total = recursivo(c, b, total);
    }
    return total;
}
```

- ## La heurística admisible:

Una heurística es **admisible** si nunca sobrestima el coste de alcanzar el objetivo. Es decir, si el valor que da para cada celda es siempre menor o igual que el coste mínimo para alcanzar el objetivo. Si la función heurística **no es admisible**, puede ser que el algoritmo A* encuentre un camino, pero que este no sea el óptimo.

$$h(n) \leq h^*(n)$$

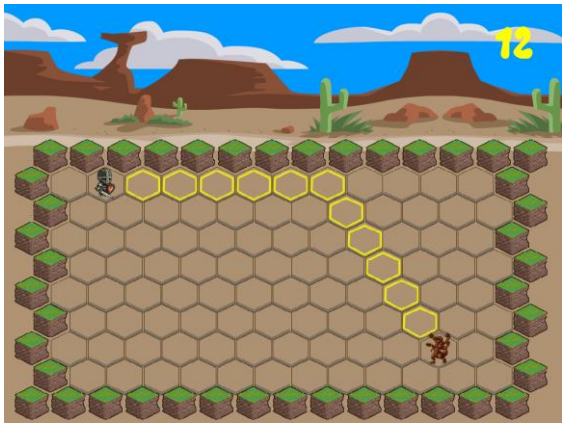
Siendo $h^*(n)$ el coste mínimo real de la celda n al objetivo.

Si h se aleja del coste mínimo real h*(n) la heurística ya tendría coste mayor al mínimo real y no se podría considerar una heurística admisible, si h es igual o menor al coste mínimo real entonces se trata de una heurística admisible.

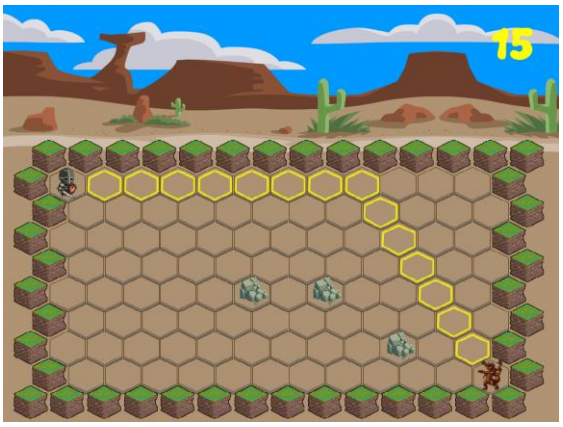
- ## Conjunto de pruebas:

Para conocer la heurística óptima de la práctica, he probado en diferentes mundos y en diferentes heurísticas, para comprobar cuales son admisibles y cuales son, además de la óptima.

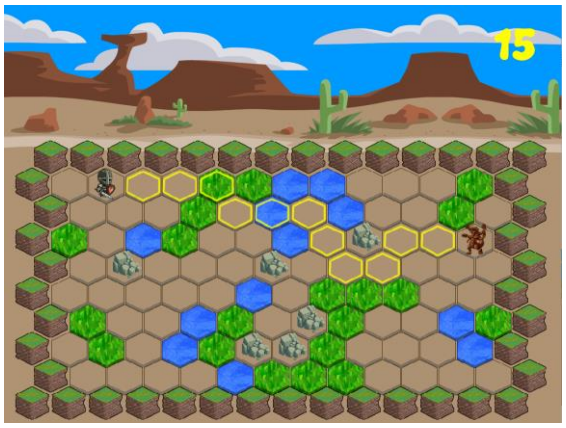
• **Mundo 01:**



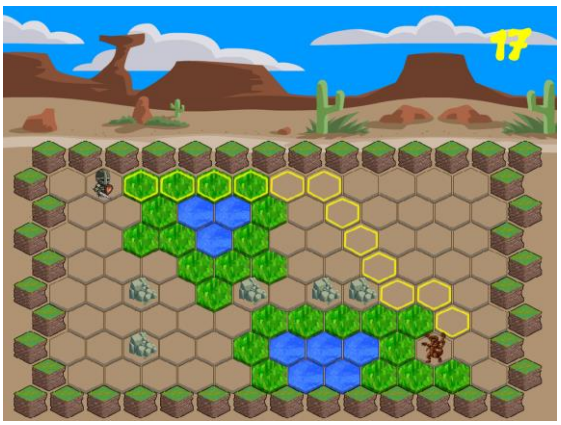
• **Mundo 02:**



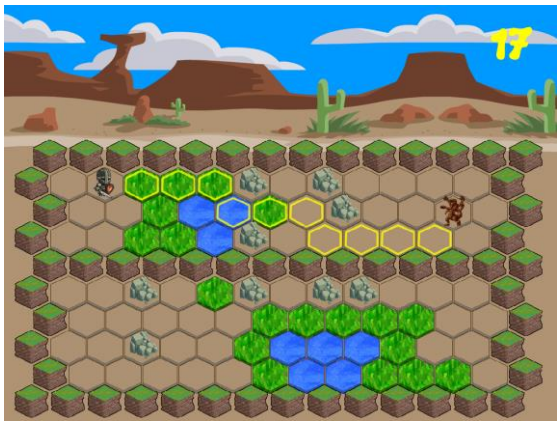
• **Mundo 03:**



• **Mundo defecto:**



• **Mundo defecto2:**



He ejecutado para mapa mundo cada uno de los métodos heurísticos y he sacado el numero de casillas, además de sus nodos expandidos.

El 17 que se muestra en la ejecución del programa es el coste de la g del caballero al dragón.

- Análisis:

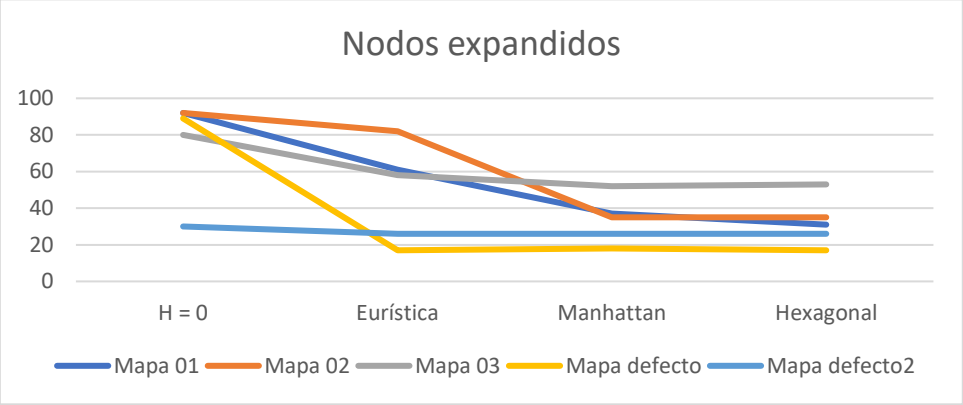
Como con h = 0 no hay estimación y te fijas con la g, lo vamos a tomar como el mejor camino base.

	Mundo 01	***Mundo 02***	***Mundo 03***	***Mundo defecto***	***Mundo defecto2***
H = 0	12	15	15	17	17
Euclídea	12	15	15	17	17
Manhattan	15	18	15	18	17
Hexagonal	12	15	15	17	17

Como podemos observar Manhattan no es una heurística admisible, el coste es mayor al de los demás, sobrepasando el coste mínimo real h*(n).

Los métodos restantes sí son heurísticas admisibles, por lo que vamos a ver como repercuten en el número de nodos expandidos para sacar la más optima entre Euclídea y Hexagonal:

	Mundo 01	***Mundo 02***	***Mundo 03***	***Mundo defecto***	***Mundo defecto2***	***Media***
H = 0	92	92	80	89	30	76,6
Euclídea	61	82	58	17	26	48,8
Manhattan	37	35	52	18	26	33,6
Hexagonal	31	35	53	17	26	32,4



Tras hacer la media vemos que la Hexagonal recorre mucho menos nodos expandidos que la Euclídea.

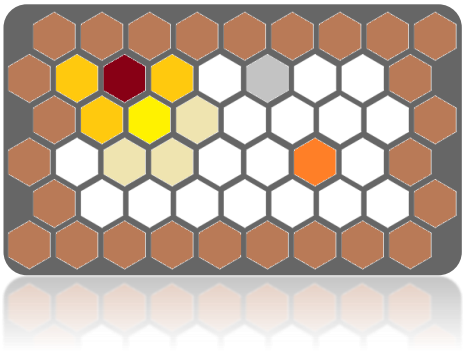
Como conclusión, el método Hexagonal es la óptima, y junto a la Euclídea se tratan de heurísticas admisibles, dejando como no admisible Manhattan.

4. Traza de un problema pequeño:

A continuación, explicaré y trazaré un problema pequeño donde se observe el funcionamiento del algoritmo A*. Como anteriormente ya he explicado como funcionaba el algoritmo A* con el pseudo código, lo explicaré de manera resumida.

El problema pequeño que voy a explicar y trazar es el siguiente:

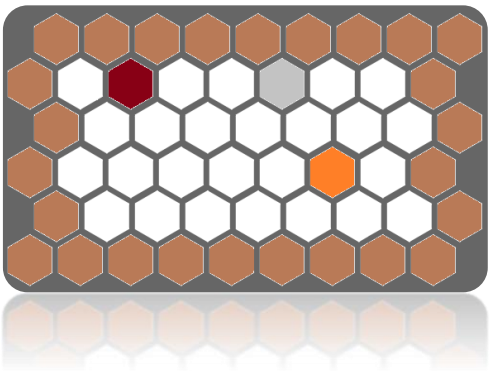
- ✓ El caballero es de color rojo.
- ✓ El dragón de color naranja.
- ✓ El nodo actual de color amarillo.
- ✓ Los hijos del nodo actual de color beis.
- ✓ Los hijos ya vistos de color naranja claro.
- ✓ Las piedras de color gris.
- ✓ Los bloques de color marrón.



• Lo primero que se hará es crear listaInterior y listaFrontera.

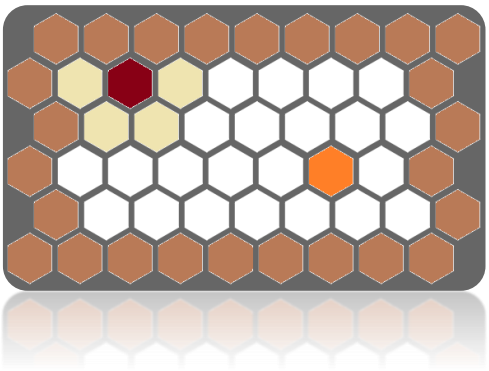
Poner como nodo inicial, antes de entrar al bucle, al caballero.

Al entrar al bucle buscará la mejor f, como solo está el caballero lo escogerá y será nuestro nodo inicial, lo eliminará de listaFrontera y lo añadirá a listaInterior.

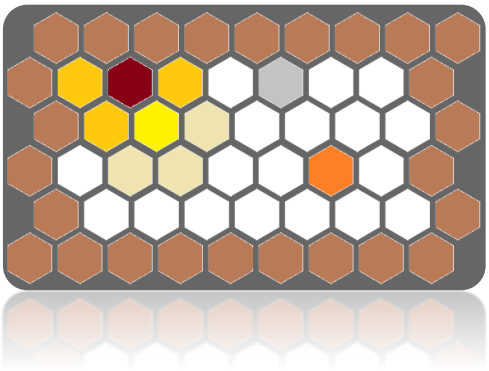


Buscaremos los hijos de ese nodo actual y empezaremos a evaluar.

• Como es la primera ejecución del bucle no los hijos no se encontrarán en listaInterior, ni existirán en listaFrontera. Entonces se calculará el coste de la casilla y se almacenaran en listaFrontera. Si el hijo se trata de un bloque o de una piedra no se añadirán a listaFrontera.



• Al volver al inicio del bucle, obtendrá el menor nodo de listaFrontera (coloreado en amarillo), lo eliminará de listaFrontera y lo añadirá a listaInterior.



Buscaremos de nuevo los hijos de ese nodo actual y empezaremos a evaluar.

• Al evaluar, como un hijo ya se encuentra el listaInterior (el caballero) no se deberá hacer nada con él. También hay dos hijos que ya se encuentran en listaFrontera entonces tendremos que mirar si la g del hijo es mejor que el que estaba. Si es así entonces tendremos que eliminar el que se encontraba en listaFrontera y poner el hijo actual. Además, también hay hijos que no se encuentran en listaFrontera entonces deberemos añadirlos.

• Tendremos que hacer este proceso, hasta que se encuentre al dragón, reconstruir todo siguiendo los punteros de los nodos y salir del bucle.