SISTEMAS INTELIGENTES

PRÁCTICA 1

Búsqueda heurística

Francisco Javier Pérez Martínez – Grupo 01
74384305M
UNIVERSIDAD DE ALICANTE | CURSO 20/2021

Contenido

Introducción

En esta práctica, se ha desarrollado el algoritmo A* para calcular el camino entre dos personajes de un juego. Dicho juego se sitúa en un tablero de celdas hexagonales en el que aparecen dos personajes: un caballero y un dragón. Se ha implementado el algoritmo mencionado para calcular el camino de menor coste por el cual el caballero puede llegar hasta el dragón y además probar dicho algoritmo con diferentes heurísticas y un análisis con diferentes casuísticas.

Objetivos

- Entender el desempeño de la búsqueda heurística y en concreto del algoritmo.
- Llevar a cabo el algoritmo A* y saber cómo elegir una heurística apropiada al problema.
- Hacer un estudio cuantitativo respecto al número de nodos explorados con este algoritmo.

Algoritmo A*

Explicación código

En este apartado explicaré las distintas funcionalidades del código implementado.

- Clase Estado

Para facilitar el almacenamiento de los datos necesarios al explorar un nodo he creado una clase Estado.

En esta clase, tenemos como atributos el estado padre anterior, la coordenada, el valor de su heurística, el coste del nodo inicial al nodo actual g y f, es decir, la suma de g y h.

```
public class Estado {

    private Estado padre;
    private Coordenada c;
    private int f, g, h;
```

- Estructuras

Para almacenar los estados he utilizado como estructura dos ArrayList, una para ListaInterior y otra para ListaFrontera.

```
//Nodos que vamos seleccionando para expandir, es decir, para que estos generen hijos.
ArrayList<Estado> listaInterior = new ArrayList<>();
//Inicio, almacenar nodos que son adyacentes y aptos para ser evaluados.
ArrayList<Estado> listaFrontera = new ArrayList<>();
```

Inicialmente, añado a listaFrontera el estado del caballero.

```
listaFrontera.add(new Estado(caballero,null));
```

- Explicación básica

En primer lugar, entramos en el bucle while para realizar del cálculo del camino entre el caballero y el dragón.

A continuación, como podemos observar, tenemos el estado inicial, posición 0 de la listaFrontera, es decir, la del caballero y un bucle for para buscar el nodo con menor función de coste.

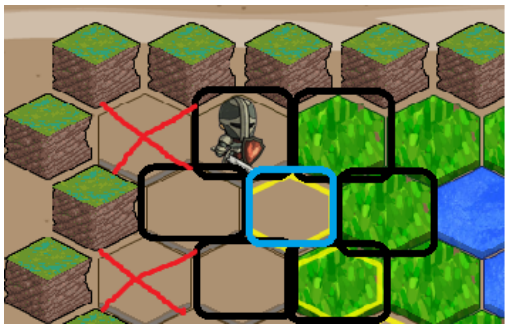
```
//Estado inicial, posicion 0 de la lista, es decir, la del caballero.
Estado n = listaFrontera.get(0);

//Buscar nodo con menor función de coste.
for(Estado menor : listaFrontera){
    if(menor.getF() < n.getF()){
        n = menor;
    }
    else if(menor.getF() == n.getF()){
        if(menor.getG() < n.getG()){
            n = menor;
        }
    }
}
```

Si el dragón ha sido encontrado, igualamos el coste_total a result y dibujamos el camino cambiando de padres hasta llegar a la inicial (caballero).

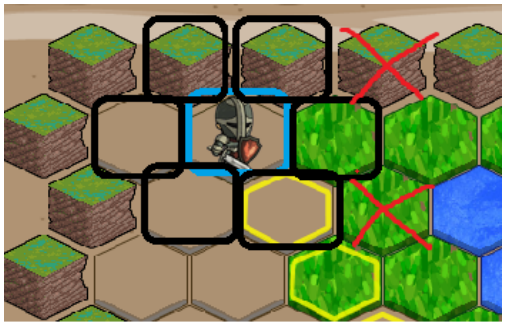
```
//Si el dragon ha sido encontrado.
if(n.getC().getY() == dragon.getY() && n.getC().getX() == dragon.getX()) {
    encontrado = true;
    coste_total = n.getG();
    result = (int) coste_total;

    while(n != null){
        camino[n.getC().getY()][n.getC().getX()] = 'X'; //Dibujar camino
        n = n.getPadre();
    }
}
```

Utilizo un método auxiliar para generar los hijos (Adyacentes()), pero antes de empezar hay que tener en cuenta que el tablero es hexagonal por lo que también debemos tener en cuenta si el estado actual es par o impar.

Si el estado actual es par, los hijos que se podrán evaluar serán los siguientes:



Si el estado actual es impar, los hijos que se podrán evaluar serán los siguientes:



Una vez generado los hijos, guardo en un ArrayList auxiliar los hijos generados para su posterior evaluación que a continuación explicaré.

```
ArrayList<Estado> ady = Adyacentes(n);
```

- Evaluación de los hijos

En primer lugar, debemos comprobar si el hijo ya ha sido evaluado, es decir, si se encuentra o no en listaInterior. En caso de encontrarlo, el hijo quedaría descartado.

```
for(Estado hijo : ady){
    boolean findI = false;

    //para cada hijo m de n que no esté en lista interior
    for(int i=0;i<listaInterior.size();i++){
        if(hijo.getC().getX() == listaInterior.get(i).getC().getX() && hijo.getC().getY() == listaInterior.get(i).getC().getY()){
            findI = true;
        }
    }
```

Si el hijo no se encuentra en ListaInterior entonces es necesario evaluarlo. Primero, debemos calcular el coste de la casilla de ese hijo para obtener posteriormente su g.

Esta parte la he realizado durante la creación de los hijos utilizando el método auxiliar mencionado anteriormente. En cuanto a los diferentes costes:

- Si la celda es de camino entonces su coste es de 1.
- Si la celda es de hierba entonces su coste es de 2.
- Si la celda es de agua entonces su coste es de 3.

Cabe destacar que he añadido como coste 1 si la celda es dragón.

```
    char celda = mundo.getCelda(ady.getC().getX(), ady.getC().getY());
    if(celda != 'p' && celda != 'b') {
        switch(celda){
            case('c'):
                ady.setG(1);
                break;
            case('h'):
                ady.setG(2);
                break;
            case('a'):
                ady.setG(3);
                break;
            case('d'):
                ady.setG(1);
                break;
            default:
                break;
        }
```

A continuación, comprobamos si el hijo que estamos evaluando ya se encuentra en ListaFrontera o no.

En caso de no encontrarse, le asignamos al hijo su g, su h y su padre y añadimos el hijo creado a ListaFrontera.

```
int G = hijo.getG() + n.getG();
int newH = hijo.getH();
```

```
// Si no se encuentra, creamos el hijo y añadimos a listaFrontera.
if(!findF){
    hijo.setG(G);
    hijo.setH(newH);
    hijo.setPadre(n);
    listaFrontera.add(hijo);
```

Si el hijo se encuentra en ListaFrontera, hay que comprobar 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, además de asignarle su g, h y su padre.

```
//si el hijo se encuentra...
else {
    // Si g del hijo menor que el que se encuentra en listaFrontera (mirar el mejor)
    if(G < hijo.getG()){
        hijo.setG(G);
        hijo.setH(newH);
        hijo.setPadre(n);
        listaFrontera.remove(hijo); //Elimino el hijo de listaFrontera.
        listaFrontera.add(hijo); //Añado el hijo evaluado.
    }
```

- Ejecución del programa

Mundo a resolver: nos aparece el mundo seleccionado en el juego y la descripción del entorno del tablero.

Camino: indica el camino tomado por nuestro Algoritmo A* implementado mediante el carácter 'X'.

Camino explorado: indica el orden el que se van expandiendo las celdas, -1 si no se ha expandido en esa coordenada.

Nodos expandidos: indica el número total de nodos expandidos.

```
Mundo a resolver
 b b b b b b b b b b b b b b
b c k h h h h c c c c c c b
 b c c h a a h c c c c c c b
b c c h h a h c c c c c c b
 b c c c h h h c c c c c c 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 d 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  1  0  4 11 20 33 41 47 52 57 63 69 -1
 -1  2  3  7 19 38 43 48 53 58 64 70 75 -1
-1  5  6  8 15 32 42 50 54 59 65 71 76 -1
 -1  9 10 12 21 34 44 51 56 62 67 73 80 -1
-1 13 14 -1 16 26 -1 46 -1 -1 68 74 81 -1
 -1 17 18 22 23 28 39 49 61 72 79 82 -1 -1
-1 24 25 -1 27 29 40 55 66 78 83 84 -1 -1
 -1 30 31 35 36 37 45 60 77 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
Nodos expandidos: 85
```

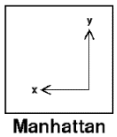
Heurísticas implementadas

- ## Heurística h=0

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:



Manhattan

$$|x_2 - x_1| + |y_2 - y_1|$$

```
private int Manhattan(Coordenada c1, Coordenada c2){

    return abs(c2.getX() - c1.getX()) + abs(c2.getY() - c1.getY());
}
```

- ## 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:

Euclidean

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

```
private int Euclidea(Coordenada c1, Coordenada c2) {
    return (int) sqrt(pow((c1.getX() - c2.getY()), 2) + pow((c1.getY() - c2.getX()), 2));
}
```

- ## Distancias adaptadas a mapas hexagonales

En primer lugar, debemos tener en cuenta que las distancias adaptas a mapas hexagonales es necesario realizar un cambio de sistema de coordenadas, en estos mapas se trabajan con coordenadas cúbicas.

Para realizar la conversión de coordenadas y la distancia adaptadas a mapas hexagonales he usado como referencia una página web que actúa como guía en todo lo relacionado con cuadrículas hexagonales.

Antes que nada, he creado una clase auxiliar Cube con las tres coordenadas x, y, z.

```
public class Cube {

    int x;
    int y;
    int z;
```

Dentro de la clase he implementado un método que calcula la distancia entre dos cubos dados.

```
/**
 * Método para calcular la distancia entre dos cubos.
 * @param a
 * @param b
 * @return
 */
public int distanciaMapasHex(Cube a, Cube b){
    return (abs(a.x - b.x) + abs(a.y - b.y) + abs(a.z - b.z)) / 2;
}
```

A continuación, he implementado un método para realizar la conversión de coordenadas axiales a cúbicas.

Hay que tener en cuenta que, nuestro mapa hexagonal es “even-r” (filas pares). Empuja filas pares a la derecha. Por lo que la conversión es distinta dependiendo del tipo de mapa hexagonal.

"even-r" horizontal layout shoves even rows right

```
/**
 * Método para convertir de coordenadas axiales a coordenadas cúbicas.
 * @param c
 * @return
 */
private Cube conversion(Coordenada c){
    int x = c.getX() - (c.getY() + (c.getY()&1)) / 2;
    int z = c.getY();
    int y = -x-z;
    return new Cube(x,y,z);
}
```

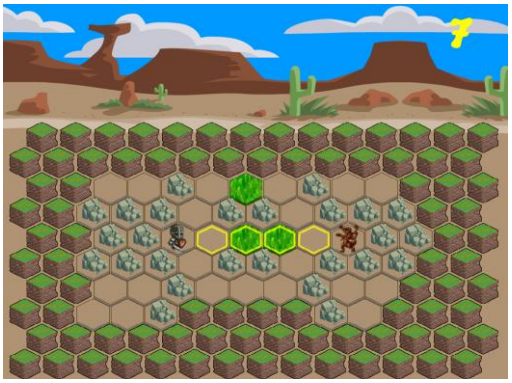
Por último, he implementado otro método en el que llamo al método que calcula la distancia entre dos cubos y le paso las coordenadas cúbicas ya convertidas. Esta función devuelve la distancia entre dos coordenadas cúbicas.

```
private int Cubicas(Coordenada c1, Coordenada c2){
    Cube cubo = new Cube();
    return cubo.distanciaMapasHex(conversion(c1), conversion(c2));
}
```

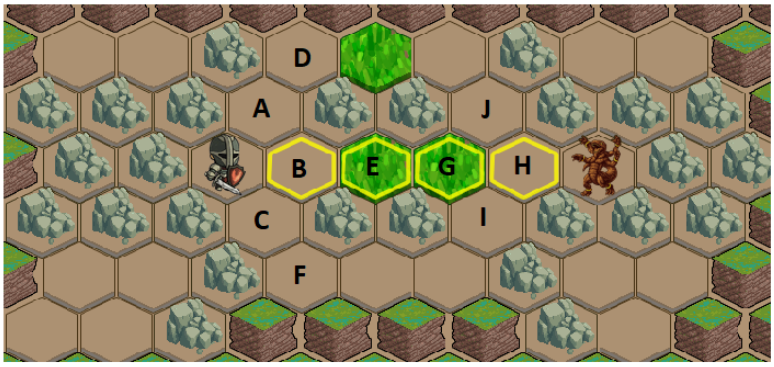
Pruebas

- ## Traza de un problema pequeño

Para realizar la traza de un problema pequeño he utilizado el siguiente mapa:



En primer lugar, voy a mostrar una captura con los nodos adyacentes del caballero enumerados para facilitar la explicación.



Además, se crearán las dos listas, listaFrontera y listaInterior, y guardar como nodo inicial la posición donde se encuentra el caballero.

A continuación, buscará la mejor f, pero como es la primera iteración, cogerá la del caballero y este será nuestro nodo inicial qué posteriormente eliminaremos de listaFrontera y lo añadiremos a listaInterior.

Mediante la siguiente traza explicaré de forma esquematizada lo que falta de cada iteración:

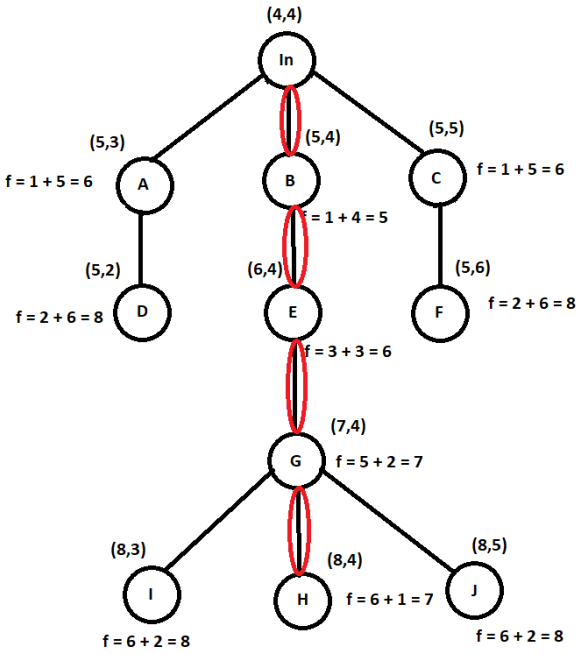
Cosas a tener en cuenta:

LI = listaInterior.

LF = listaFrontera.

Heurística usada para la traza: Manhattan

Coordenada del dragón (9,4) – Coordenada del caballero (4,4)



1º Iteración:

LF = Nodo inicial (caballero).

LI = Nodo inicial (caballero).

G = 0, H = 0, F = 0.

LF = {~~In~~}

LI = {In}

2º Iteración:

LF = {A, ~~B~~, C}

LI = {~~In~~, B}

G = 1, H = 4 → F = G + H = 1 + 4 = 5.

3º Iteración:

LF = {~~A~~, C, E}

LI = {~~In~~, ~~B~~, A}

G = 1, H = 5 → F = G + H = 1 + 5 = 6.

4º Iteración:

LF = {~~C~~, E, D}

LI = {~~In~~, ~~B~~, ~~A~~, C}

G = 1, H = 5, → F = G + H = 1 + 5 = 6.

5º Iteración:

LF = {~~E~~, D, F}

LI = {~~I~~, ~~B~~, ~~A~~, ~~C~~, E}

G = 3, H = 3 → F = G + H = 3 + 3 = 6.

6º Iteración:

LF = {D, F, ~~G~~}

LI = {~~I~~, ~~B~~, ~~A~~, ~~C~~, ~~E~~, G}

G = 5, H = 2 → F = G + H = 5 + 2 = 7.

7º Iteración:

LF = {D, F, ~~H~~, I, J}

LI = {~~I~~, ~~B~~, ~~A~~, ~~C~~, ~~E~~, ~~G~~, H}

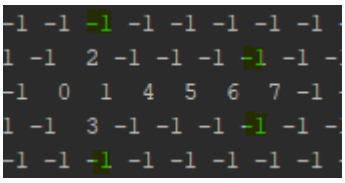
G = 6, H = 1 → F = G + H = 7.

Una vez finalizado la traza, cabe destacar que los nodos D, F, I, J no aparecen explorados en el mapa debido a que nunca tienen mejor f que los hijos de B por lo que estos nodos no se expandirán. También resaltar que los nodos ya evaluados en listaInterior no se vuelven a evaluar como se puede ver en la traza.

Por último, para reconstruir el camino se irá siguiendo los punteros de los nodos elegidos cambiando de padre hasta llegar al nodo inical.

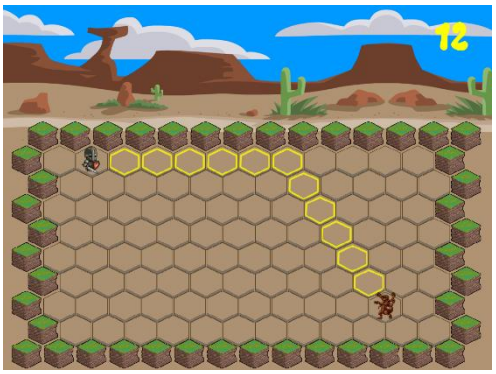
Al usar un mapa con pequeñas dimensiones y Manhattan como heurística también hay que resaltar que ésta heurística expande menos nodos con mapas pequeños.

Manhattan no es admisible, pero cuando encuentra la solución es la que menos expande.

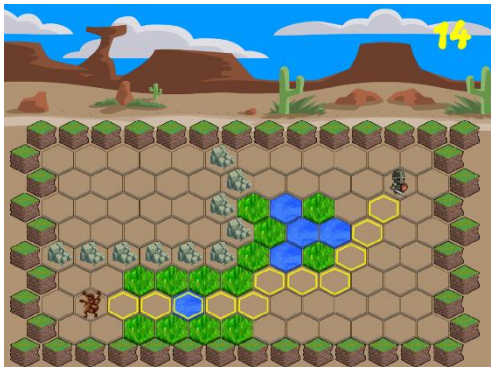


- Mundos

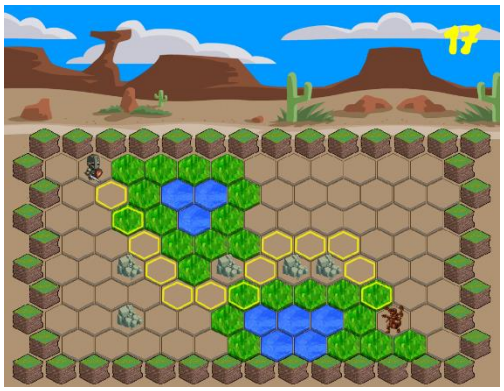
mundo_01



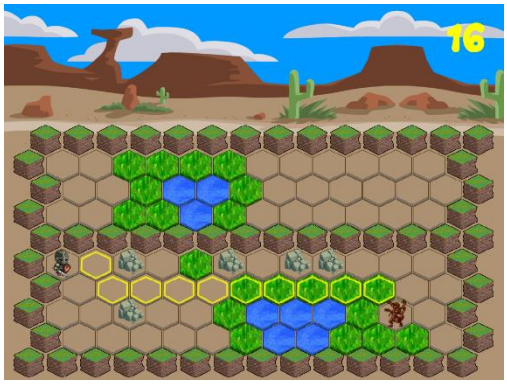
mundo_02



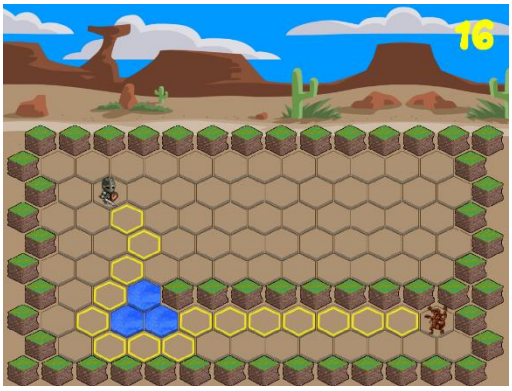
mundo_defecto



mundo_defecto_02

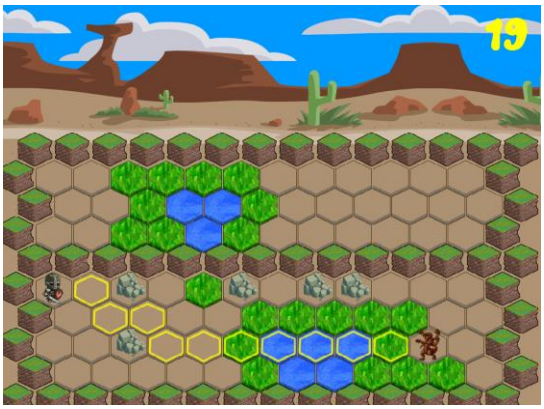


mundo_defecto_03



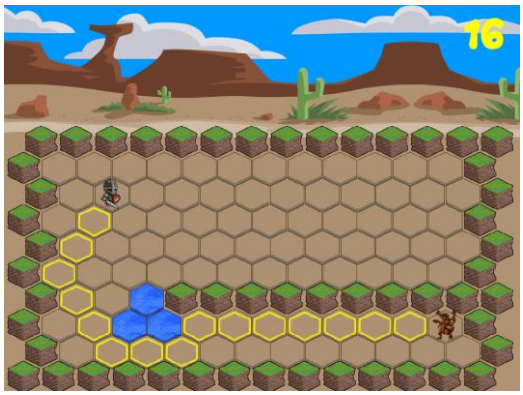
Estos son los mundos utilizados para el posterior análisis de las heurísticas.

Antes de entrar en el análisis, cabe destacar que en el mundo '*mapa_defecto_02'* cuando usamos la heurística de Manhattan el coste del caballero al dragón es mayor que el resto de las heurísticas implementadas.

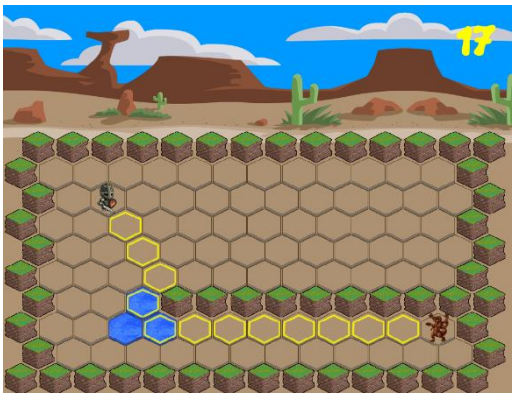


También, en el mundo mapa '*mapa_defecto_03'* según la heurística que se use el recorrido del camino es distinto e incluso con manhattan supera el coste mínimo real h*(n) al igual que el mundo anterior mencionado.

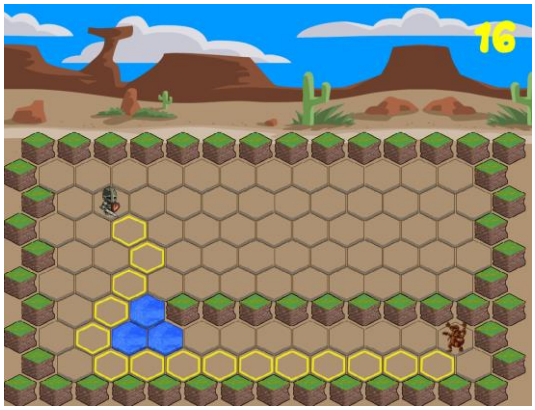
h = 0



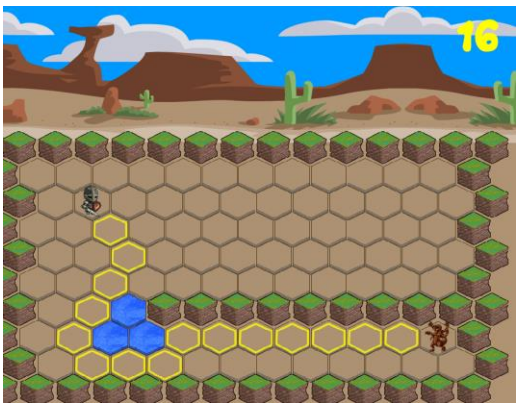
Manhattan



Euclídea



Hexagonal



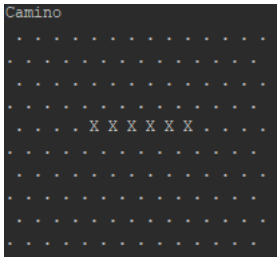
- ## Análisis de las heurísticas

En primer lugar, responderé a las preguntas propuestas en el enunciado de la práctica.

- ¿Se incluye el nodo inicial y el final en el camino? ¿por tanto se deben mostrar con X en la solución?

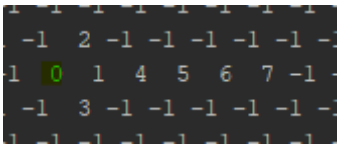
  Incluyo tanto el nodo inicial como el final por lo que el camino marcado por X se muestra desde el inicio (incluido) hasta el final (incluido).

  Ejemplo del camino obtenido en la explicación de la traza de un problema pequeño.



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

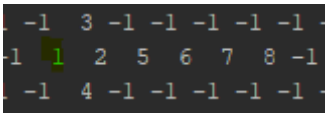
Mi camino explorado empieza por 0 debido a como está organizado el código al asignarle la matriz del camino_explorado a los nodos explorados.




```
//aumentando caminos expandidos
camino_expandido[n.getC().getY()][n.getC().getX()] = expandidos;
expandidos++;
```

Con la imagen anterior se entiende mejor, si pusiésemos el contador donde se almacenan el número de nodos expandidos antes de la matriz que indica el orden en el que se expanden los nodos, en nuestra ejecución nos saldría que nuestro camino explorado empieza por 1, pero al tenerlo como en la imagen pues inicialmente el valor de nodos expandidos es 0.

Ejemplo con el contador antes de la asignación a la matriz:



A continuación, en la siguiente tabla se muestran los diferentes valores del coste que hay entre el dragón y el caballero.

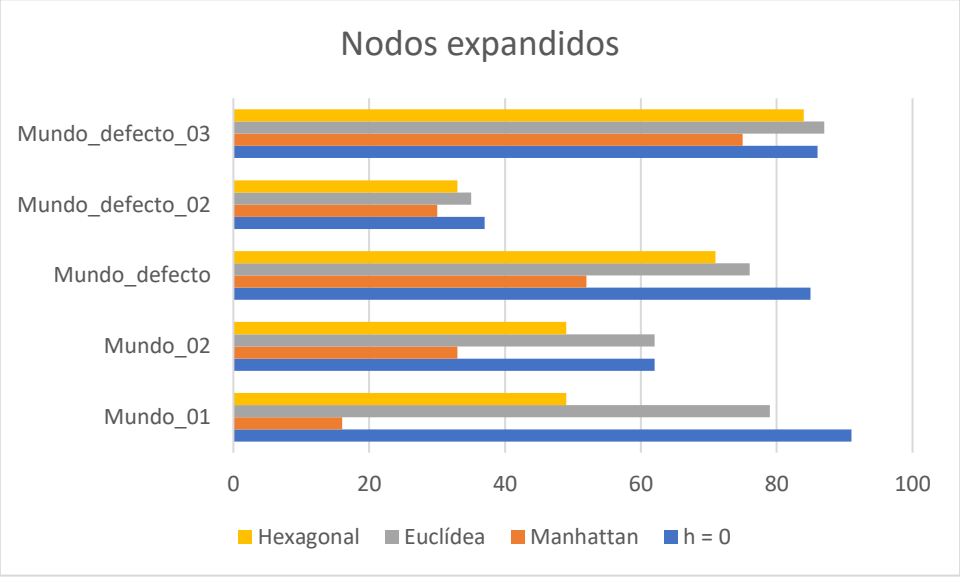
Heurísticas	Mundo_01	Mundo_02	Mundo_defecto	Mundo_defecto_02	Mundo_defecto_03
h = 0	12	14	17	16	16
Manhattan	12	14	17	19	17
Euclídea	12	14	17	16	16
Hexagonal	12	14	17	16	16

Como podemos observar, en la distancia de manhattan en los dos mapas últimos el resultado del coste es superior con respecto al resto. Al ser mayor que los demás y sobrepasar el coste mínimo real h*(n) podemos decir que Manhattan no es una heurística admisible.

Una vez obtenido dicha conclusión, sabiendo que las otras heurísticas sí son admisibles, procedemos a comprobar mediante una tabla en el que se muestren los números de nodos expandidos para poder comprobar cual, de las otras dos, Euclídea y Hexagonal, es la más óptima.

Heurísticas	Mundo_01	Mundo_02	Mundo_defecto	Mundo_defecto_02	Mundo_defecto_03
h = 0	91	62	85	37	86
Manhattan	16	33	52	30	75
Euclídea	79	62	76	35	87
Hexagonal	49	49	71	33	84

Heurísticas	Media
h = 0	68,7
Manhattan	36,2
Euclídea	64,7
Hexagonal	54,3



Por un lado, como se puede apreciar en las tablas, en todos los mapas la heurística hexagonal es la que menos número de nodos expande sin tener en cuenta Manhattan. Para estar más seguros, he realizado la media de números de nodos expandidos por cada heurística y como se puede observar la hexagonal es la que menor media tiene.

Por otro lado, podemos ver que si no utilizamos una heurística los números de nodos que se expanden durante la realización del algoritmo incrementan lo cual hace que haya un coste computacional mayor.

Finalmente, podemos concluir que la heurística Hexagonal sería la más optima y junto la Euclídea se tratan de dos heurísticas admisibles, sin embargo, Manhattan se trata de una heurística no admisible como hemos podido ver anteriormente.