Francisco Joaquín Murcia Gómez
48734281H
Grado en ingeniería informática UA

Sistemas Inteligentes

Practica 1: Búsqueda heurística

Contenido

Introducción

Esta práctica consiste en diseñar un algoritmo de búsqueda de caminos A* y probarlo con diferentes heurísticas (Manhattan, Euclídea y Distancias cubicas), para ello hacemos uso de un mapa con celdas hexagonales con diferentes tipos de celdas con costes distintos de tal forma que el camino resultante no sea el mas corto, si no el de menor coste. Los distintos tipos de celdas son los siguientes:

- Camino: coste 1, símbolo "c".
- Hierba: coste 2, símbolo "h".
- Agua: coste 3, símbolo "a"
- Piedra: intransitable, símbolo "p"
- Bloque: intransitable, símbolo "b" (límites del mundo)

Para probar las heurísticas se han creado diferentes mapas con características y situaciones diferentes.

Programa

Funcionamiento del juego

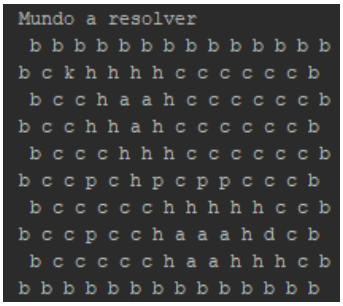
Juego

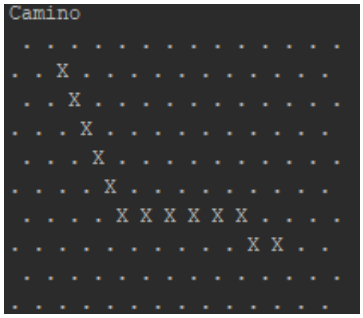
EL programa consiste en un juego, hay que conseguir que el camino sea superior a un coste de 20 (teniendo en cuenta que el caballero hará el camino más optimo posible), para ello colocaremos piedras en su camino para obligarle a recorrer otras rutas.

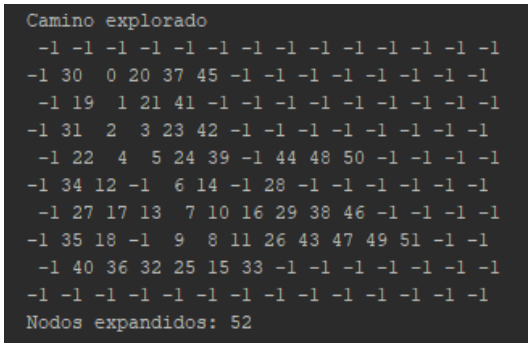
Nada más le demos a comenzar nos pedirá el mapa, luego, nos paridera el camino y después se nos colocará el mapa como podemos ver en las imágenes de arriba

Mapas mundo camino y explorados

En la terminal nos aparecerá tres representaciones del mapa:







El "Mundo a resolver" es la representación del mundo, el "Camino" es el camino elegido por el algoritmo, y el "Camino explorado" es todas las casillas que revisa el algoritmo hasta encontré la ruta mas optima y la cantidad total de nodos

Algoritmo A*

Clase "Estado"

He creado una clase auxiliar llamada "Estado" en la cual almaceno la coordenada de ese nodo, El estado anterior (el padre), el coste de la celda, el coste de la heurística, el coste del camino, y el coste de la función (heurística + camino).

```
private int F;//coste de la funcion
private int G;//coste del camino
private int H;//heuristica
private Coordenada N;//coordenada
private Estado Padre;
private int costeCelda;
```

En el constructor inicializamos la heurística el padre y la coordenada, el coste del camino le colocamos 0 ya que se actualizará mas adelante y el coste de la función como camino + heurística.

```
public Estado(int costeCelda, int H, Coordenada N, Estado Padre) {
    this.G=0;
    this.H = H;//la heuristica
    this.N = N;//corrodenada
    this.Padre = Padre;
    this.costeCelda=costeCelda;
    this.F = G+H;//coste de la funcion
}
```

Los geters y seters son estándar a excepción del setH() y setG() ya que cuando se actualiza la heurística o el coste del camino la función se actualiza:

```
public void setG(int G) {
    this.G = G;
    this.F=G+H;//cuando ac
```

```
public void setH(int H) {
    this.H = H;
    this.F=G+H;//cuando ac
}
```

Celdas adyacentes

Para saber las celdas vecinas a la celda actual he diseñado la función "hijos()" la cual le paso por parámetro un estado y me devuelve una lista de las celdas adyacentes.

Con dos bucles recorremos las coordenadas y dependiendo si la fila es par o no escogeremos unas coordenadas u otras, una vez elegida una coordenada, comprobamos con un switch que tipo de celda es esa ya que hay celdas intransitables y con diferentes costes, de tal forma que si es una celda de tipo bloque ("B") o piedra ("P") o detecta al caballero ("K") se descartara la celda.

Cuando se selecciones una celda crearemos un estado nuevo con esa coordenada calcularemos su heurística y le asignaremos un coste dependiendo del tipo que sea, después se añadirá el estado a una lista donde se guardan las demás celdas validas como podemos observar en la siguiente imagen con una celda de agua

```
case 'a':
    a.setN(new Coordenada(X+i,Y+j));
    h=heuristica(a.getN());
    a.setH(h);
    a.setCosteCelda(3);
    Hijos.add(a);
    break;
```

Al finalizar los bucles devolvemos dicha lista

Funcionamiento

En primer lugar, se ha creado dos listas de tipo estado, interior y frontera, en esta ultima se inicializa con el nodo padre, el caballero, las listas guardan los nodos prometedores y los nodos adyacentes respectivamente.

```
ArrayList<Estado>listaInterior=new ArrayList<>();
ArrayList<Estado>listaFrontera=new ArrayList<>();
Estado padre=new Estado(0,heuristica(mundo.getCaballero()) ,mundo.getCaballero(),null);
listaFrontera.add(padre);
```

El bucle principal del algoritmo iterará hasta que no tenga más nodos adyacentes o haya encontrado al dragón.

En primer lugar, obtenemos el nodo mas prometedor de los adyacentes, es decir el que tiene menor coste:

```
for(Estado e:listaFrontera){// ol
    //System.out.println("{X="+e
    if(n.getF()>e.getF())n=e;
}
```

A partir de ese nodo se comprueba si es la coordenada del dragón, en ese caso acabaría el algoritmo y reconstruiría el camino desde el caballero hasta ese nodo puesto que ha encontrado al dragón, en caso contrario seguirá buscando

```
if(n.getN().getX()==mundo.getDragon().getX() && n.getN().getY()==mundo.getDragon().getY()){//s
    encontrado=true;
    coste_total=n.getG();
    result=(int) coste_total;

    while (n!=null){//reconstruir camino desde la meta al inicio siguiendo los punteros
        camino[n.getN().getY()][n.getN().getX()]='X';
        n=n.getPadre();
    }
```

En el caso contrario, se mueve el nodo seleccionado a "listaInterior" y se borra en "listaFrontera", después generamos en una lista nueva ("Hijos") los nodos adyacentes del nodo seleccionado con la función ya mencionada en el apartado anterior.

En segundo lugar, recorremos nos nodos hijos para actualizar sus pesos siempre y cuando no estén en la lista de los prometedores (interior). Si el hijo no estaba en la lista de los adyacentes, se actualizan los costes y se añade.

```
int costeHijo=hijo.getCosteCelda()+n.getG();
int H=hijo.getH();
if(!estaEnLista(hijo,listaFrontera)){//si el hijo no esta en lista frontera

    hijo.setG(costeHijo);
    hijo.setH(H);
    hijo.setPadre(n);
    listaFrontera.add(hijo);
```

En el caso contrario, si tiene menor coste, se borra el hijo de la frontera, se actualiza el padre y los costes y se añade a frontera

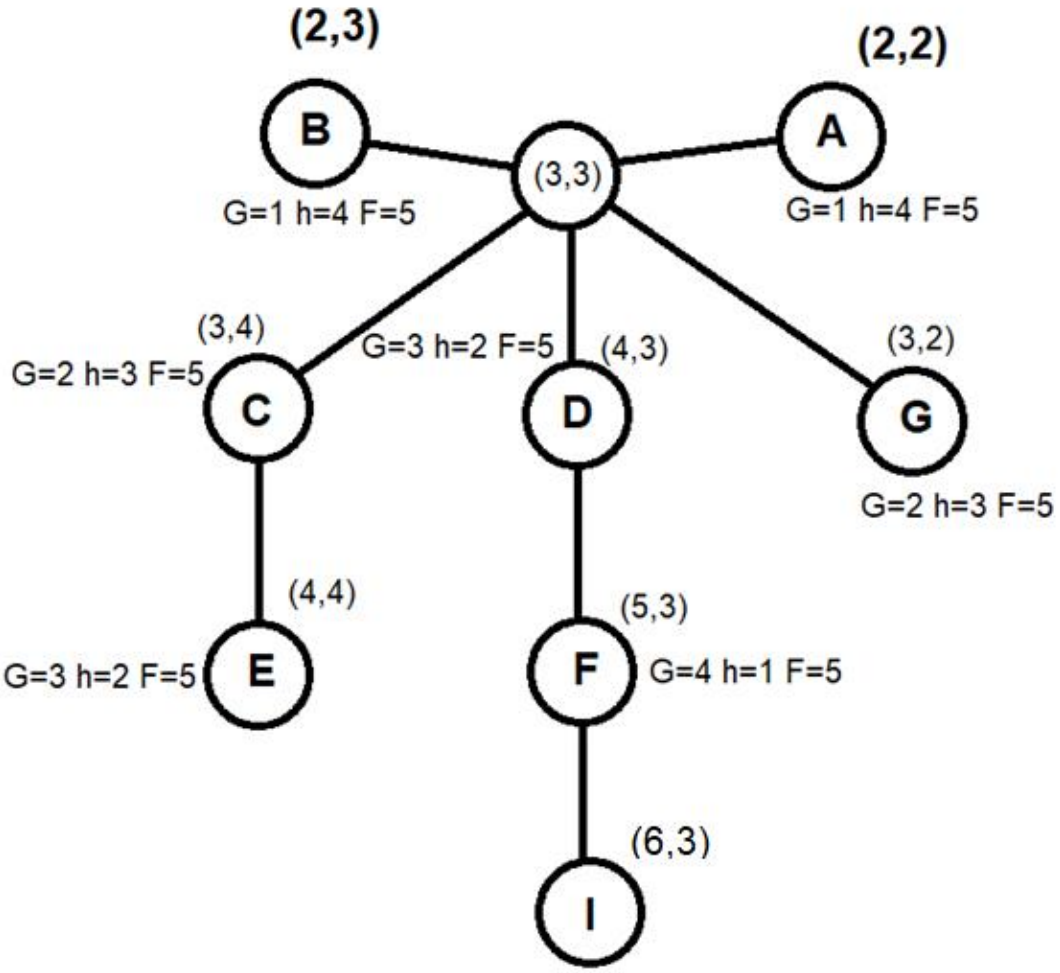
```
}else if(hijo.getG()>costeHijo){//si

    listaFrontera.remove(hijo);
    hijo.setPadre(n);
    hijo.setG(costeHijo);
    hijo.setH(H);
    listaFrontera.add(hijo);
}
```

Una vez hecha estas comprobaciones se vuelve a iterar

Ejemplo del funcionamiento

Para explicar el funcionamiento he diseñado un mapa más simple usando la heurística euclídea

A continuación, el árbol de nodos explorados, siendo el nodo (3,3) el caballero y el nodo "I" el dragón.



1ª iteración

Generamos los vecinos al caballero, se nos generan A, B, C, D, G, se le asigna a cada uno su peso y se guardan en lista frontera.

2ª iteración

Evaluamos A: al hacer los vecinos de B nos percatamos que ya hemos visitado esos nodos así que se descarta de lista frontera y se guarda en lista interior.

3ª iteración

Evaluamos B: al hacer los vecinos de B nos percatamos que ya hemos visitado esos nodos así que se descarta de lista frontera y se guarda en lista interior.

4ª iteración

Evaluamos G: generamos sus vecinos H y D, D se descarta ya que esta en lista frontera y tiene peor coste, sin embargo, H se le asigna su peso y se añade a lista frontera.

5ª iteración

Evaluamos C: generamos sus vecinos E, D y solo nos quedamos con E ya que D ya estaba en lista frontera y tiene peor coste, se le asignan a E su peso y lo guardamos en lista Frontera y se descarta C de lista frontera y se guarda en lista interior.

6ª iteración

Evaluamos D: generamos sus vecinos C, E, H, G, F, descartamos C, G, H y E ya que se han evaluado y poseen peor coste, así que se pasan a lista interior, asignamos F su peso y lo guardamos en lista frontera

7ª iteración

Evaluamos E: generamos sus vecinos F, J, D y C, descartamos D y C ya estaban en lista interior y tiene peor coste, asignamos los pesos de J y lo guardamos en lista frontera, F al estar ya en lista frontera y no mejorar se deja igual

8ª iteración

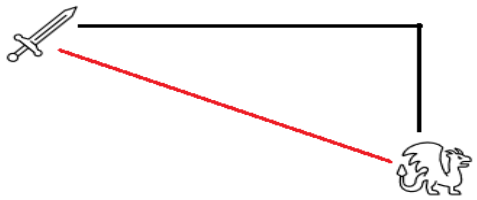
Evaluamos F: generamos sus vecinos H, D, E, J, I descartamos D, H y E ya que se han evaluado y poseen peor coste, así que se pasan a lista interior, nos damos cuenta de que el nodo I es el dragón así que finalizamos

Heurísticas

Tipos de heurísticas

Distancia Manhattan

La distancia Manhattan es la suma de las diferencias de la coordenada.



La línea negra seria la representación gráfica de la distancia manhattan.

Para calcular la distancia h (Manhattan) seria:

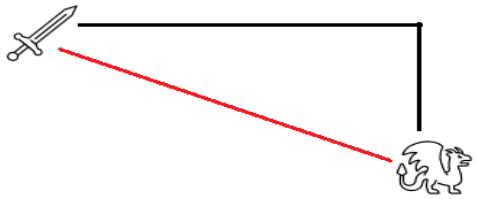
$$\mathrm{h} = |\Delta X| + |\Delta Y|$$

Si por ejemplo contamos que la coordenada 1 es la inicial y la 2 la objetivo tenemos que:

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

Distancia Euclídea

La distancia euclídea es la aplicación del teorema de Pitágoras:



Las líneas negras serían las correspondientes a las coordenadas "X" e "Y" y la roja la representación gráfica de la distancia euclídea.

Para calcular la distancia h (Euclídea) seria:

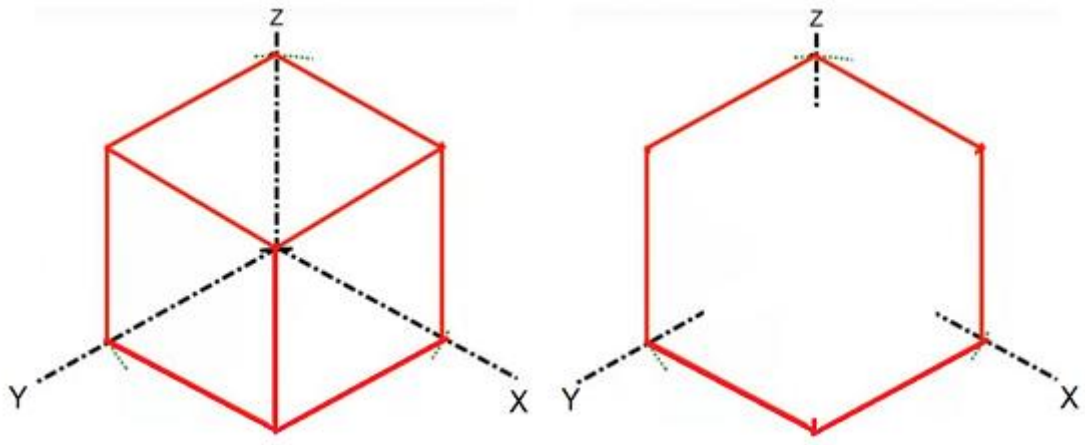
$$h = \sqrt{(\Delta X)^2 + (\Delta Y)^2}$$

Si por ejemplo contamos que la coordenada 1 es la inicial y la 2 la objetivo tenemos que:

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

Distancia cubica

Nuestro sistema de coordenadas es bidimensional, muchos mapas hexagonales usan coordenadas con 3 dimensiones (cubicas) ya que la planta de un cobo visto desde perspectiva isométrica es un hexágono.



Hemos tenido que convertir las coordenadas en cubicas, una vez convertidas la distancia se obtendría así siendo la coordenada 1 es la inicial y la 2 la objetivo y empleando la fórmula de distancias entre puntos en un plano tridimensional tenemos que:

$$h = \frac{|x_2 - x_1| + |y_2 - y_1| + |z_2 - z_1|}{2}$$

Implementación

Función “int heuristica()”

Para aplicar las diferentes distancias, he implementado una función con las tres distancias, de tal forma que cambiando el valor de una variable manualmente se aplicara una heurística u otra.

```java
    private int heuristica(Coordenada a){

        /*
        Con la siguiente variable se puede cambiar de heurstica
        simplemente hay que cambiar el numero de la variable "tipoDeHeuristica"
        */
        int tipoDeHeuristica=1;
        /*
        DISTANCIAS:
        Manhattan --> 1
        Euclídea ---> 2
        Cubica -----> 3
        En el caso de no selecionar ninguno de estos valores, no se empleara ninguna heuristica
        */

        int h=0;
        switch (tipoDeHeuristica){
            case 1://distancia manhattan
                h=abs(mundo.dragon.getX()-a.getX())+abs(mundo.dragon.getY()-a.getY());
                break;
            case 2://distancia Euclídea
                int X=mundo.dragon.getX()-a.getX();
                int Y=mundo.dragon.getY()-a.getY();
                h=(int) Math.sqrt(X*X+Y*Y);
                break;
            case 3://distancia cubica
                Cubo c= new Cubo();
                h= distanciaCuboToCubo(axialToCubica(mundo.dragon),axialToCubica(a));
                break;
        }
        return h;

    }
```

En el caso 1 y el 2 (manhattan y euclídea) la función esta implementada directamente, en el caso de la cubica primero convierte las coordenadas a cubicas con una función externa y luego saca la distancia con otra función.

Funciones cubicas

```java
public class Cubo {

    private int x;
    private int y;
    private int z;


    public Cubo(){
        x=0;
        y=0;
        z=0;
    }

    public Cubo(int X,int Y, int Z){
        x=X;
        y=Y;
        z=Z;
    }


     public int getX(){...3 lines }
    public int getY(){...3 lines }
     public int getZ(){...3 lines }
```

Para implementar las coordenadas cubicas he creado una clase similar a Coordenada llamada Cubo, pero con 3 variables, “X”, “Y” y “Z”.

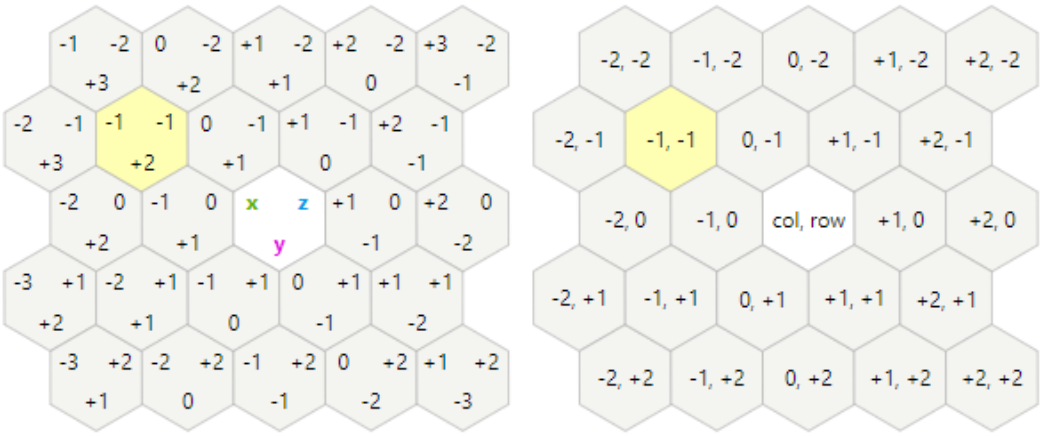
En esta tenemos lo básico, dos constructores y los geter.

Para convertir de las coordenadas axiales a las cubicas he implementado la función Cubo axialToCubica().

Para realizar la conversión he utilizado el código de la siguiente web que explica todo lo relacionado con coordenadas cubicas AQUI

```
private Cubo axialToCubica(Coordenada a){

    int x=a.getX()-(a.getY()+(a.getY()&1))/2;
    int z=a.getY();
    int y=-x-z;
    Cubo c=new  Cubo(x,y,z);
    return c;
}
```

Hay que tener en cuenta que nuestro mapa corresponde a un mapa hexagonal de tipo “even-r”

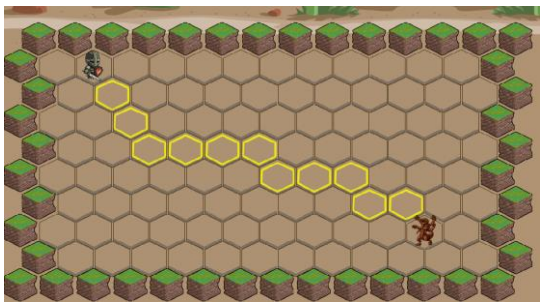


Análisis heurístico

Introducción

Para realizar una comparación de las diferentes heurísticas, he diseñado diversos mapas con características diferentes para probar todos los casos posibles, los mapas son los siguientes:

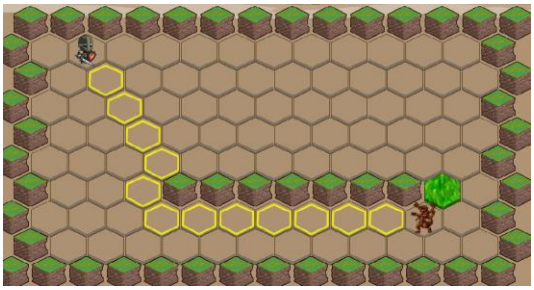
Mundo_01



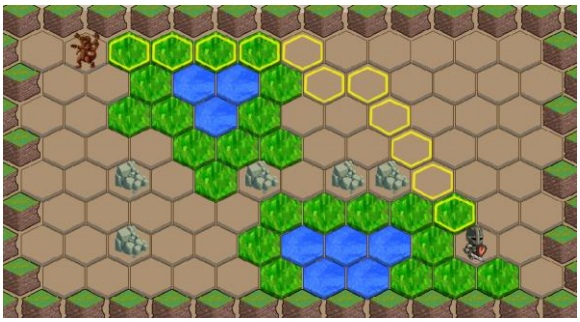
Mundo_defecto

Mundo_prueba1

Mundo_prueba2



Mundo_inverso

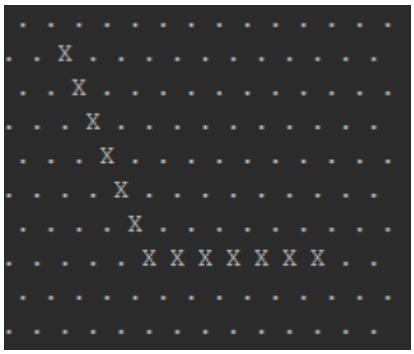


Mundo_prueba3

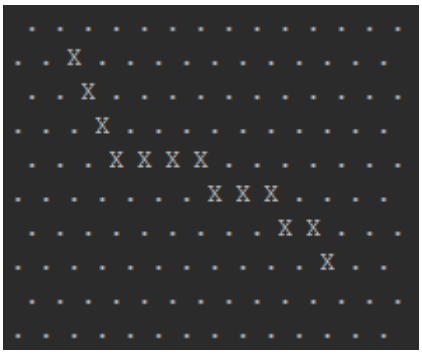
Análisis

Al analizar los diferentes mapas notamos las diferencias de comportamiento en las diferentes euristicas, en el "mundo_01" al ser un mundo vacio se nota mas que manhattan utiliza mas los catetos del triangulo formado po los ejes y la diagonal entre el dragon y el caballero, y con la euclidea tiende a asemejarse a la hipotenusa, esto esta explicado en el apartado "Tipos de heuristicas"

Manhattan



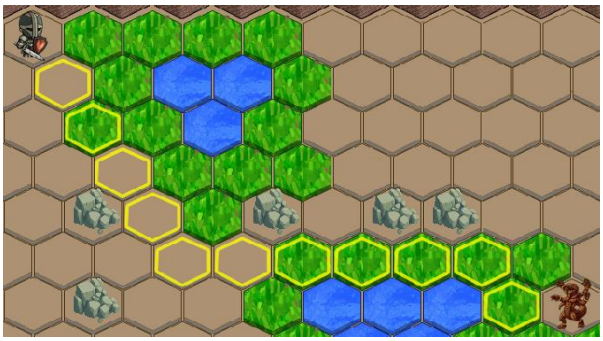
Euclídea



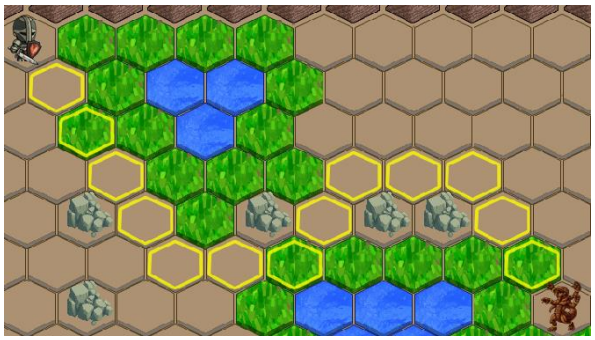
En el mapa "mundo_defecto" los trazados son muy diferntes, pero todos menos la manhattan generan el camino con peso minimo (17), manhattan sin embargo aporta 18, esto es debido a que manhattan no es una heuristica admisible, en el nodo (6,5) en vez de escoger la celda de tipo camino como hace la euclidea y la

cubica, escoge la celda de delante centrandose mas en la heuristica que en el peso de las celdas

Manhattan

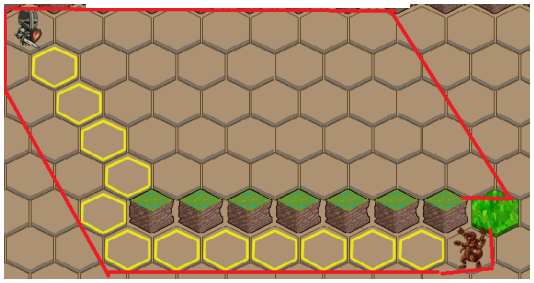


Euclídea/Cubica

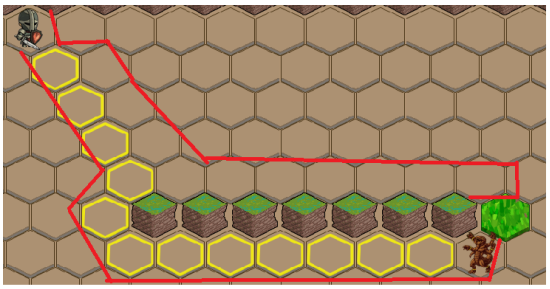


En el mapa “mundo_prueba2”se nota la poca cantidad de nodos que explora el manhattan, a continuación, el área explorada por la cubica y la manhattan:

Cubica



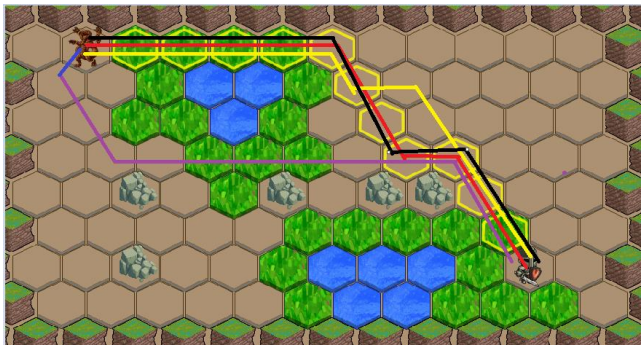
Manhattan



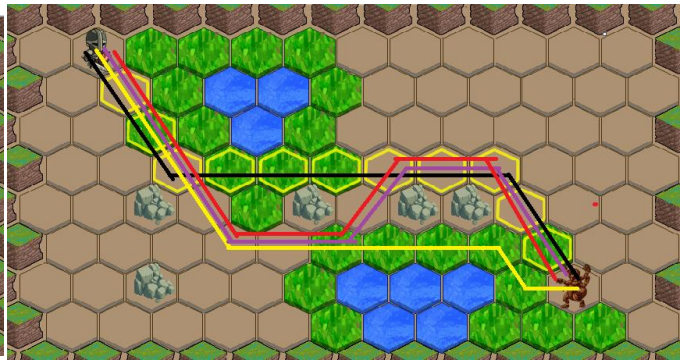
Se puede observar que Manhattan explora menos área que la cubica y la euclídea ya que esta ultima es muy similar a la cubica, no obstante, el camino son los mismos en todas

El mapa “mundo_inverso” es una versión del mapa “mundo_defecto” solo que se ha intercambiado el caballero por el dragón, de forma inesperada los caminos generados son muy distintos al de “mundo_defecto”, y en esta ocasión manhattan ha conseguido el camino más corto (17), a continuación, una comparativa de los caminos de inverso y defecto, siendo la línea amarilla manhattan, la línea negra sin heurística la roja cubica y la línea lila la euclidea

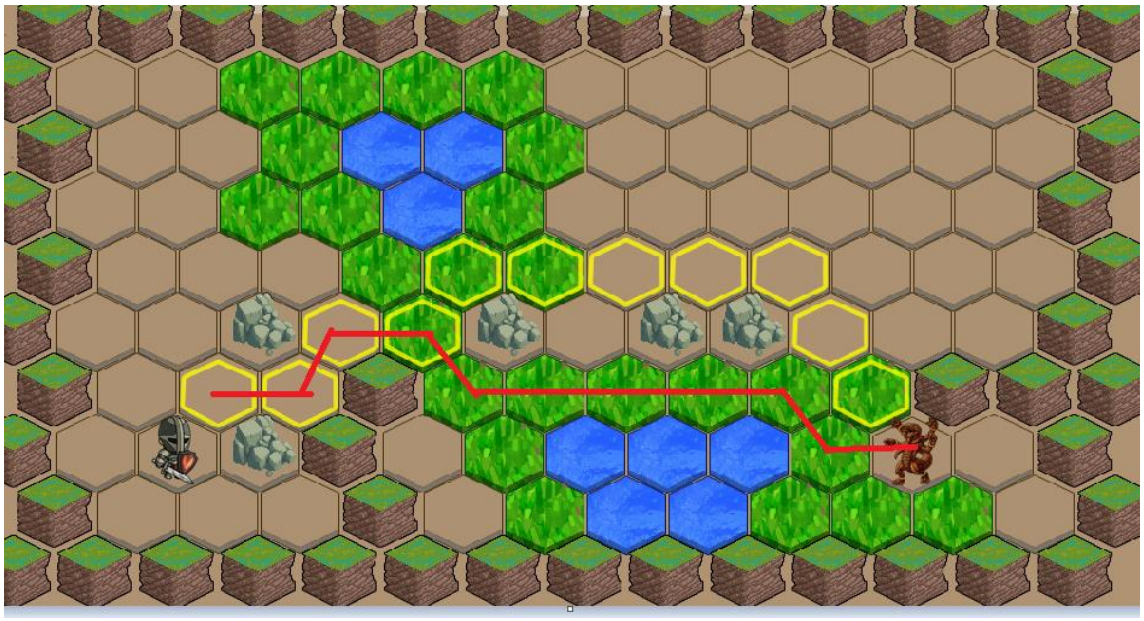
Mundo_inverso



Mundo_defecto

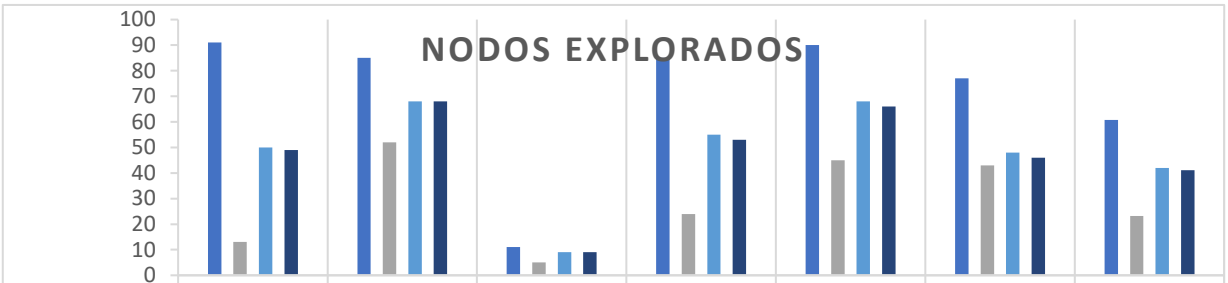


Por último, a analizar "Mundo_prueba3" ocurre algo similar que con "mundo_defecto", todas las heurísticas hacen el camino mas corto (16) menos manhattan que hace 18. En este caso ocurre porque cuando estamos en la celda (5,5), por el valor de la heurística elige la celda de abajo pese a que por esa ruta hay muchas casillas de hierba. A continuación, una comparativa de lo que hace manhhatan(rojo) con lo que hacen el resto de las heurísticas



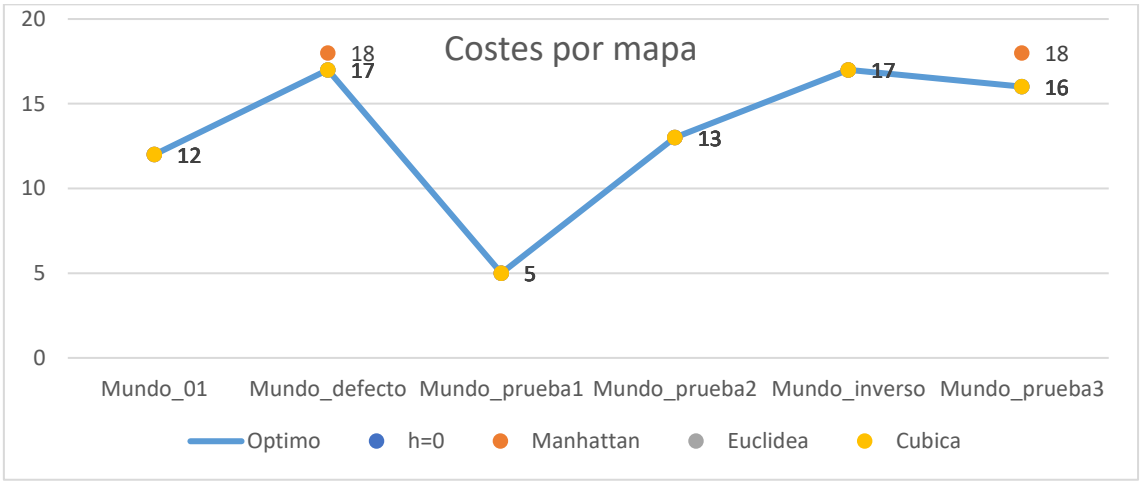
Conclusión

Si hacemos un análisis de todo lo dicho encontramos que manhattan es la que menos nodos hace como se puede observar:



	Mundo_01	Mundo_defecto	Mundo_prueba1	Mundo_prueba2	Mundo_inverso	Mundo_prueba3	MEDIA
h=0	91	85	11	85	90	77	60,72
Manhattan	13	52	5	24	45	43	23,23
Euclidea	50	68	9	55	68	48	42,01
Cubica	49	68	9	53	66	46	41,11

Sin embargo, pese a que la manhattan explore menos, no es admisible ya que no siempre encuntra el camino mas optimo



Tambien el hecho de no utilizar heuristica hace que incremente el coste computacional ya que explora mas y tarda ma sen encontrar solucion, pese a que la solucion sea precisa.

Manhattan seria buena opcion para aquellos sitios donde se requiera poco coste computacional y no importe un aprecision del 100% ya que las la falta de precision no es mucha,

A si mismo, la cubica y la euclidea serian las mas adecuadas ya que nos aporta exactitud a un numero de nodos aceptable, sin embrago yo me quedaria con la cubica, ya que pese a no ser mucha diferencia explora menos nodos