

SISTEMAS INTELIGENTES

CURSO 2018-19

PRACTICA 1



IVÁN MAÑÚS MURCIA - 48729799K

GRUPO JUEVES 9:00-11:00

UNIVERSIDAD DE ALICANTE

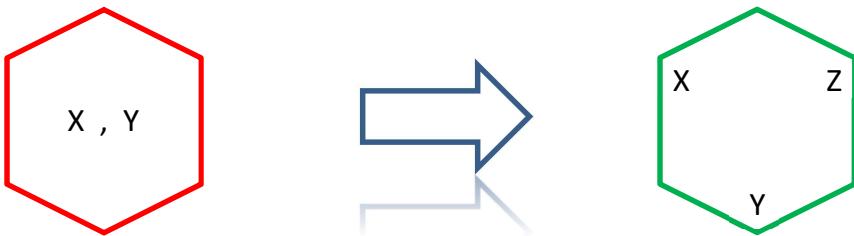
GRADO EN INGENIERÍA INFORMÁTICA

Contenido

Pseudocódigo del algoritmo A*	3
Posibles casos al evaluar nuevos nodos	4
Heurística	5
Heurísticas utilizadas	5
1. h=0	5
2. Manhattan	5
3. Euclidea	6
4. Manhattan adaptada para coordenadas cúbicas	7
5. Euclidea adaptada para coordenadas cúbicas	7
Mejor heurística para el problema	8
¿Qué ocurre cuando h se aleja de h*?	8
¿Rapidez u optimización del coste?	9
Explicación y traza	9
Conjunto de pruebas	16
Bibliografía	25

Documentación de la práctica

-Antes de empezar a explicar la práctica, he de dejar claro que he usado coordenadas cúbicas y no coordenadas cartesianas.



Pseudocódigo del algoritmo A*

```
Algoritmo A*
    Bool encontrado = false;
    Entero result = -1;
    Nodo n;
    Nodo end;
    Entero g = 0;

    Inicializar nodo end con las coordenadas del dragón.
    Inicializar nodo n con las coordenadas del caballero.

    Asignar la heurística al nodo n.

    ArrayList<Nodo> listaInterior;
    ArrayList<Nodo> listaFrontera;

    Añadir(listaFrontera,n);
    mientras listaFrontera no esté vacía hacer
        n = obtenemos el nodo con menor f de listaFrontera;
        si n es end
            encontrado = true;
            result = 1;
            Asignamos el coste_total;
            Sacamos el número de nodos expandidos;
            Sacamos el camino;
        sino
            Borrar(listaFrontera,n);
            Añadir(listaInterior,n);
            ArrayList<Nodo> vecinas = sacamos las vecinas
            transitables de n;
```

```

Para cada vecina transitable que no esté en lista interior.
    g'(vecina) = n.g + c(n,m);
    si la vecina no está en listaFrontera
        la guardamos asignándole la h, la g calculada,
        su f y su padre que será n.
    sino, es decir, si está en listaFrontera
        si la g calculada es mejor que la g anterior
            cambiamos su padre, su g y su f;
    finsi
    finpara
    finsi
finmientras

si encontrado
    Mostramos el camino;
    Mostramos el camino expandido;
    Mostramos el coste total junto al número de nodos
expandidos;
finsi

devolver result;

fin A*

```

Posibles casos al evaluar nuevos nodos

-Al evaluar nodos durante la exploración de A*, nos podemos topar con 2 casos como bien se puede observar en el pseudocódigo.

1. Si el nuevo nodo no está en Lista Frontera

- Si el nuevo nodo que se explora no está en Lista Frontera, se mete calculando su H hasta el dragón, su g, su f y asignamos el padre que será n.

2. Si el nuevo nodo está en Lista Frontera

- Si el nuevo nodo que se explora ya está en Lista Frontera, se recalcula su f y su g y se le asigna un nuevo padre con el que mejora el camino.

Heurística

Heurísticas utilizadas

-Las heurísticas utilizadas para esta práctica han sido:

1. $h=0$

-Es una heurística que convierte a nuestro algoritmo en una búsqueda de coste uniforme no informada.

-Esto quiere decir que la búsqueda es ciega, va expandiendo nodos y eligiendo el mejor.

Es una heurística que expande muchos nodos pero siempre encuentra el camino óptimo, por ende, es una heurística admisible.

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	3	0	6	14	23	36	38	44	49	58	60	70	-1			
-1	2	1	10	25	41	45	43	52	57	61	69	72	-1			
-1	4	5	9	18	37	46	51	50	56	62	68	73	-1			
-1	8	7	13	24	35	47	53	55	63	67	74	-1	-1			
-1	11	12	-1	15	31	-1	42	-1	-1	66	75	-1	-1			
-1	17	16	22	21	28	39	54	64	76	-1	77	-1	-1			
-1	19	20	-1	27	26	40	59	71	-1	-1	-1	-1	-1			
-1	30	29	32	34	33	48	65	-1	-1	-1	-1	-1	-1			
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

2. Manhattan

- La heurística Manhattan para mapas con coordenadas cartesianas es:

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

Por lo tanto, en este mapa de celdas hexagonales, y siguiendo la implementación de la clase nodo, me ha resultado un tanto mala, puesto que suele explorar pocos nodos, pero en cambio nunca, o muy pocas veces me saca el coste total óptimo, por lo tanto, la comprendo cómo inadmisible.

```

Camino explorado
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 36 0 31 39 43 48 49 -1 -1 -1 -1 -1 -1 -1
-1 36 29 -1 -1 -1 50 -1 -1 -1 -1 -1 -1 -1
-1 32 15 2 28 45 -1 -1 51 -1 -1 -1 -1 -1
-1 16 27 3 26 41 -1 -1 52 53 54 -1 -1 -1
-1 23 17 -1 4 14 -1 37 -1 -1 -1 55 -1 -1
-1 18 33 13 5 12 34 38 44 47 -1 56 -1 -1
-1 22 19 -1 11 6 10 35 46 -1 -1 -1 -1
-1 21 20 24 9 7 8 25 40 42 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
Nodos expandidos: 56
18.0

```

3. Euclidea

- La heurística Euclidea para mapas con coordenadas cartesianas es:

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

Con esta heurística, se obtienen mejores resultados que con la Manhattan, ya que la Manhattan solo mira en 4 direcciones y esta hace una línea recta entre los dos puntos. Nunca me ha salido un coste total superior al óptimo, por lo tanto es admisible.

```

Camino explorado
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 10 0 4 16 28 37 44 50 -1 -1 -1 -1 -1 -1
-1 5 1 6 29 51 -1 46 52 -1 -1 -1 -1 -1 -1
-1 20 2 3 11 35 47 -1 -1 -1 -1 -1 -1 -1 -1
-1 18 7 8 19 36 -1 38 45 48 -1 -1 -1 -1
-1 34 15 -1 9 17 -1 27 -1 1 49 53 -1 -1
-1 31 23 24 12 14 26 33 43 -1 54 -1 -1
-1 -1 30 -1 22 13 21 39 -1 -1 -1 -1 -1 -1
-1 -1 40 41 32 25 42 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

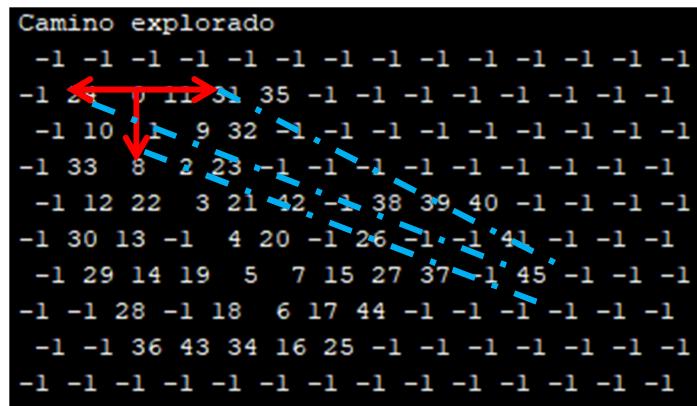
```

4. Manhattan adaptada para coordenadas cúbicas

- La heurística Manhattan para mapas con coordenadas cúbicas es:

$$\text{MAX}(|x_2 - x_1|, |y_2 - y_1|, |z_2 - z_1|)$$

Siguiendo la implementación realizada y adaptándolo a celdas hexagonales, esta heurística es admisible y muy buena, ya que saca el menor coste en todos los mapas probados.



5. Euclidea adaptada para coordenadas cúbicas

- La heurística Euclidea para mapas con coordenadas cúbicas es:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Siguiendo la implementación realizada y adaptándolo a celdas hexagonales, esta heurística es admisible y muy buena, pero no saca el menor coste en todos los mapas, le pasa algo semejante a la euclidean de coordenadas cartesianas, si sigue el camino en línea recta, pero hay algo en medio que hace que sea mayor el coste, le da igual, sigue su camino (lo veremos después en el apartado de casuísticas).

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	26	0	10	33	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	12	1	2	27	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	11	3	15	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	25	21	4	15	1	-1	28	29	30	-1	-1	-1	-1	-1
-1	-1	-1	-1	5	13	-1	-1	-1	31	-1	-1	-1	-1	-1
-1	-1	-1	22	6	7	8	18	-3	-1	34	-1	-1	-1	-1
-1	-1	-1	-1	24	9	17	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	32	14	20	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Mejor heurística para el problema

- Por lo tanto, después de investigar por internet y hacer pruebas con distintos mapas, la mejor heurística para mapas de celdas hexagonales es la Manhattan adaptada para mapas hexagonales.

¿Qué ocurre cuando h se aleja de h^* ?

Si $h(n)$ es 0, entonces solo $g(n)$ desempeña un papel, y A * se convierte en el algoritmo de Dijkstra, que está garantizado para encontrar una ruta más corta.

Si $h(n)$ siempre es menor que (o igual a) el costo a la meta, entonces A * está garantizado para encontrar el camino más corto. Cuanto más bajo $h(n)$ es, más se expande el nodo A *, lo que lo hace más lento.

Si $h(n)$ es exactamente igual al costo hasta la meta, entonces A * solo seguirá el mejor camino y nunca expandirá nada más, lo que lo hace muy rápido. Aunque no podemos hacer que esto suceda en todos los casos, podemos hacerlo exacto en algunos casos especiales.

Si $h(n)$ a veces es mayor que el costo hasta la meta, no se garantiza que A * encuentre el camino más corto, pero puede correr más rápido.

En el otro extremo, si $h(n)$ es muy alto en relación con $g(n)$, entonces solo $h(n)$ desempeña un papel.

¿Rapidez u optimización del coste?

-Ha habido una lucha continua por mi parte con esta práctica intentando mejorar el número de nodos a expandir y mantener el coste total uniforme, probando a elegir de la lista frontera el primer nodo con mejor f, el ultimo...

Pero sin duda, la implementación más significativa, fue después de probar que aparte de que tuviera la mejor f, fuera la mejor h.

En algunos mapas funcionaba bien, puesto que escogía el nodo que más se acercaba al objetivo, pero en otros funcionaba mal, puesto que si había hierba o agua de por en medio, le daba igual coger el camino de mayor coste.

Esta mejora podría ser correcta si, al igual que A*épsilon, realizara una lista focal donde contener los nodos con mejor f.

Explicación y traza

Empieza el algoritmo.

```
//Nodo actual  
Nodo n;  
  
//Dragón  
Nodo end;  
  
//Peso  
int g = 0;  
  
end = new Nodo(this.mundo.getDragon(),g,0);  
n = new Nodo(this.mundo.getCaballero(),g,0);
```

Tenemos las dos celdas, la inicial y la del dragón.

Llevando a cabo la transformación a coordenadas cúbicas en el constructor de Nodo, y añadiendo su g y su h, todas a 0 inicialmente.

```

Nodo(Coordenada c, int g, int h){
    this.g = g;
    this.h = h;
    this.f = this.g + this.h;
    this.x = c.getX() - (c.getY() + (c.getY()&1)) / 2;
    this.z = c.getY();
    this.y = -this.x-this.z;
    this.padre = null;
}

n.setHCMannhattan (end);

```

Añadimos la heurística hasta el dragón del nodo n.



```

Caballero está en: 1,2
En coordenadas cúbicas: COORDENADAS: [1,2] ;CUBICAS: X= 1; Z= 1; Y= -2; F = 0.0; G= 0; H= 12.0; 
El dragón se encuentra en: 7,11
En coordenadas cubicas: COORDENADAS: [7,11] ;CUBICAS: X= 7; Z= 7; Y= -14; F = 0.0; G= 0; H= 0.0;

```

```

ArrayList<Nodo> listaInterior = new ArrayList<Nodo>();
ArrayList<Nodo> listaFrontera = new ArrayList<Nodo>();
listaFrontera.add(n);

```

Creamos las listas, frontera e interior y añadimos a la frontera el nodo n, dando inicio al algoritmo.

```
LISTA INTERIOR  
LISTA FRONTERA  
COORDENADAS: [1,2] ;CUBICAS: X= 1; Z= 1; Y= -2; F = 0.0; G= 0; H= 12.0; FATHER = null
```

```
n = n.obtenerMenorf(listaFrontera);  
  
↓  
  
public Nodo obtenerMenorf(ArrayList<Nodo> lf){  
    Nodo nod;  
    if(lf.size() == 1){  
        nod = lf.get(0);  
    }  
    else{  
        nod = lf.get(0);  
        for(int i=1;i<lf.size();i++){  
            if(lf.get(i).f <= nod.f){  
                nod = lf.get(i);  
            }  
        }  
    }  
    return nod;  
}
```

Obtenemos el nodo con menor f de la lista frontera, como aquí solo hay 1, lo cogemos a él.

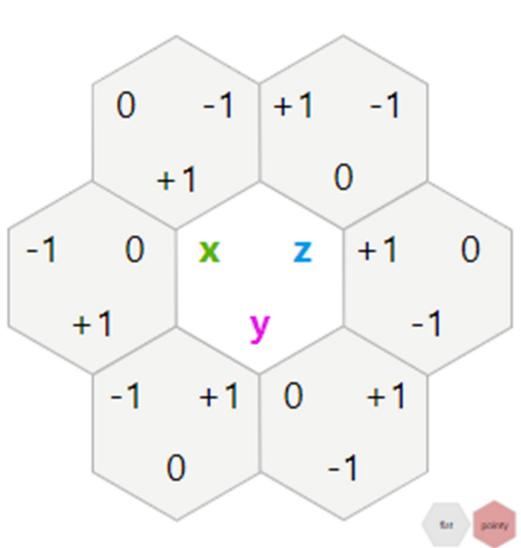
```
if(n.equals(end))...
```

Si, n, es el final, paramos, sino...

```
listaFrontera.remove(n);  
listaInterior.add(n);
```

Lo borramos de listaFrontera y lo metemos a listaInterior.

```
ArrayList<Nodo> vecinas = n.misVecinitas();  
  
public ArrayList misVecinitas(){  
    ArrayList<Nodo> vecinas;  
    vecinas = new ArrayList<Nodo>();  
    int vecinass[][][] = new int[][][] {  
        {1,-1,0},  
        {1,0,-1},  
        {0,1,-1},  
        {-1,1,0},  
        {-1,0,1},  
        {0,-1,1}  
    };  
  
    for(int[] vecinas1 : vecinass)  
        vecinas.add(new Nodo(this.x + vecinas1[0], this.y +  
                            vecinas1[1], this.z + vecinas1[2]));  
    return vecinas;  
}
```



Sacamos las vecinas del nodo.

```

vecinas = eligiendoVecinas(vecinas,listaInterior);

public ArrayList<Nodo> eligiendoVecinas(ArrayList<Nodo> vec, ArrayList<Nodo> li){
    ArrayList<Nodo> elegidas = new ArrayList<Nodo>();
    for(int i=0;i<vec.size();i++){
        //Si la vecina no está en lista interior
        if(!vec.get(i).estaEn(li)){
            //Filtro para no coger vecinas intransitables
            if(this.mundo.getCelda(vec.get(i).deCubicaAOffset().getX(),
                vec.get(i).deCubicaAOffset().getY()) != 'b' &&
                this.mundo.getCelda(vec.get(i).deCubicaAOffset().getX(),
                vec.get(i).deCubicaAOffset().getY()) != 'p'){

                elegidas.add(vec.get(i));
            }
        }
    }
    return elegidas;
}

```

Elegimos las vecinas transitables como podemos ver...



```

for(int i=0;i<vecinas.size();i++){
    g = vecinas.get(i).setG(n);

    //Si no está en lista frontera, lo metemos
    if(!vecinas.get(i).estaEn(listaFrontera)){
        vecinas.get(i).setHCMannhattan(end);
        vecinas.get(i).g = g;
        vecinas.get(i).setF();
        vecinas.get(i).setFather(n);
        listaFrontera.add(vecinas.get(i)); //Añadimos a LF
    }

    //Si está, comparamos si su g es mejor de la que tenía y
    le asignamos el nuevo padre
    else{
        if(g < vecinas.get(i).g){
            vecinas.get(i).setFather(n);
            vecinas.get(i).g = g;
            vecinas.get(i).setF();
        }
    }
}

```

Seleccionamos las vecinas que queremos meter en lista Frontera...

```

LISTA INTERIOR
COORDENADAS: [1,2] ;CUBICAS: X= 1; Z= 1; Y= -2; F = 0.0; G= 0; H= 12.0; FATHER = null
LISTA FRONTERA
COORDENADAS: [1,3] ;CUBICAS: X= 2; Z= 1; Y= -3; F = 13.0; G= 2; H= 11.0; FATHER = COORDENADAS: [1,2]
COORDENADAS: [1,1] ;CUBICAS: X= 0; Z= 1; Y= -1; F = 14.0; G= 1; H= 13.0; FATHER = COORDENADAS: [1,2]
COORDENADAS: [2,1] ;CUBICAS: X= 0; Z= 2; Y= -2; F = 13.0; G= 1; H= 12.0; FATHER = COORDENADAS: [1,2]
COORDENADAS: [2,2] ;CUBICAS: X= 1; Z= 2; Y= -3; F = 12.0; G= 1; H= 11.0; FATHER = COORDENADAS: [1,2]

```

- Y volveríamos a hacer la misma operación hasta que Lista Frontera estuviera vacía o hubiéramos llegado a la meta.

```

LISTA INTERIOR
COORDENADAS: [1,2] ;CUBICAS: X= 1; Z= 1; Y= -2; F = 0.0; G= 0; H= 12.0; FATHER = null
COORDENADAS: [2,2] ;CUBICAS: X= 1; Z= 2; Y= -3; F = 12.0; G= 1; H= 11.0; FATHER = COORDENADAS: [1,2]
LISTA FRONTERA
COORDENADAS: [1,3] ;CUBICAS: X= 2; Z= 1; Y= -3; F = 13.0; G= 2; H= 11.0; FATHER = COORDENADAS: [1,2]
COORDENADAS: [1,1] ;CUBICAS: X= 0; Z= 1; Y= -1; F = 14.0; G= 1; H= 13.0; FATHER = COORDENADAS: [1,2]
COORDENADAS: [2,1] ;CUBICAS: X= 0; Z= 2; Y= -2; F = 13.0; G= 1; H= 12.0; FATHER = COORDENADAS: [1,2]
COORDENADAS: [2,3] ;CUBICAS: X= 2; Z= 2; Y= -4; F = 13.0; G= 3; H= 10.0; FATHER = COORDENADAS: [2,2]
COORDENADAS: [3,2] ;CUBICAS: X= 0; Z= 3; Y= -3; F = 13.0; G= 2; H= 11.0; FATHER = COORDENADAS: [2,2]
COORDENADAS: [3,3] ;CUBICAS: X= 1; Z= 3; Y= -4; F = 13.0; G= 3; H= 10.0; FATHER = COORDENADAS: [2,2]

```

...

Pequeño problema

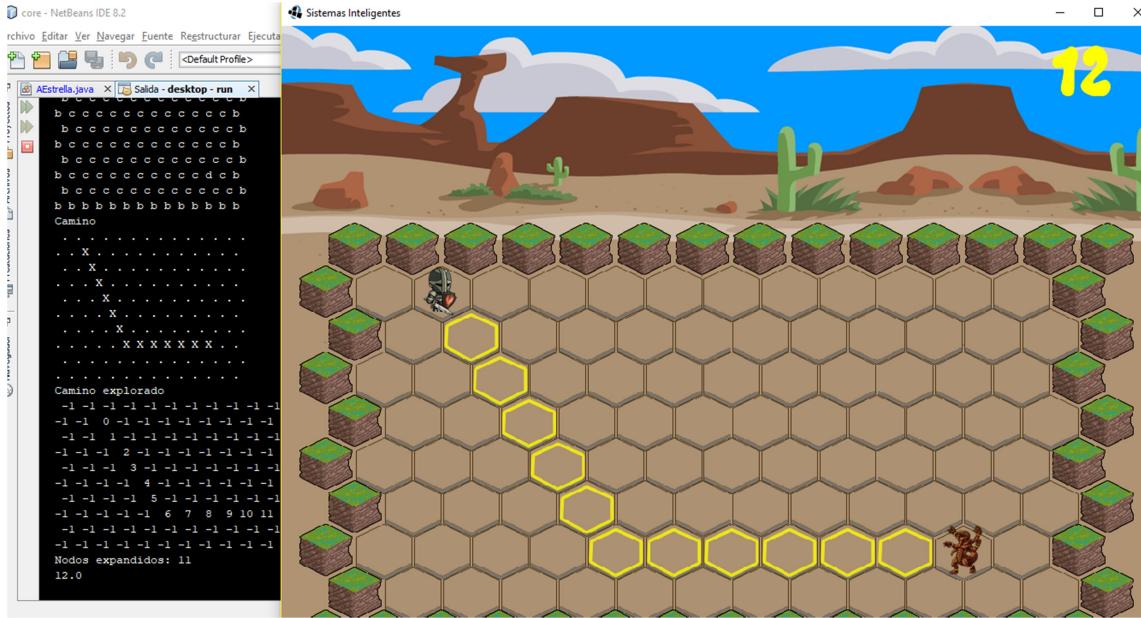


Traza:

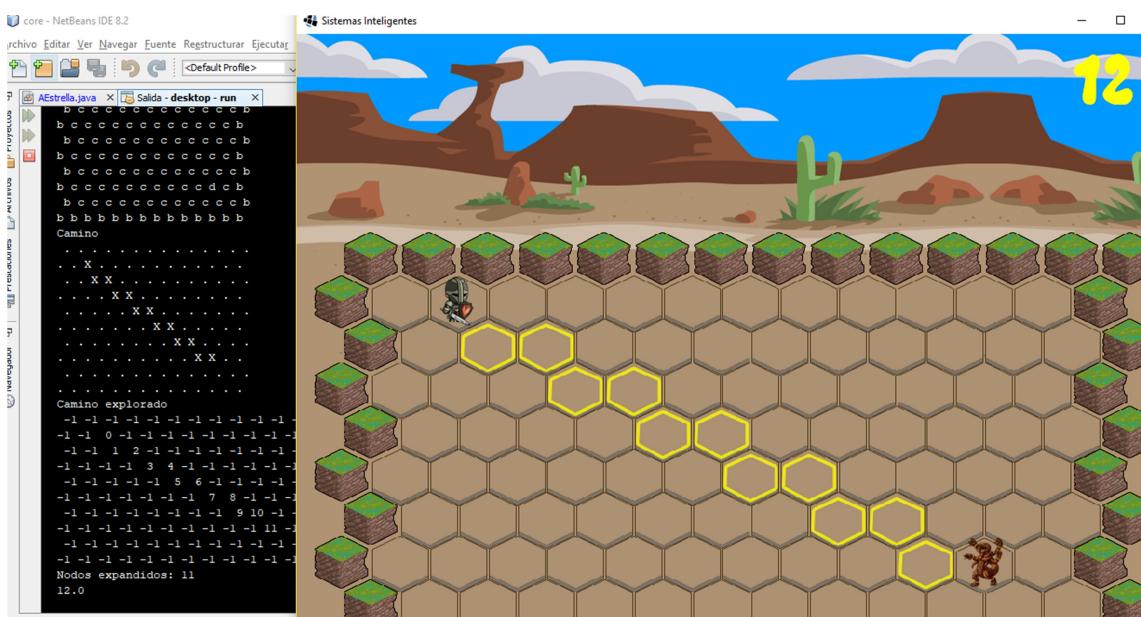
Conjunto de pruebas

- Para este apartado vamos a usar 4 mapas, algunos diseñados por mí y otros cedidos por agradecidos compañeros por haberles ayudado en varias dudas y, obviamente, para comprobar quien exploraba menos nodos.
- A continuación presento el resultado de los distintos mapas con las distintas heurísticas.

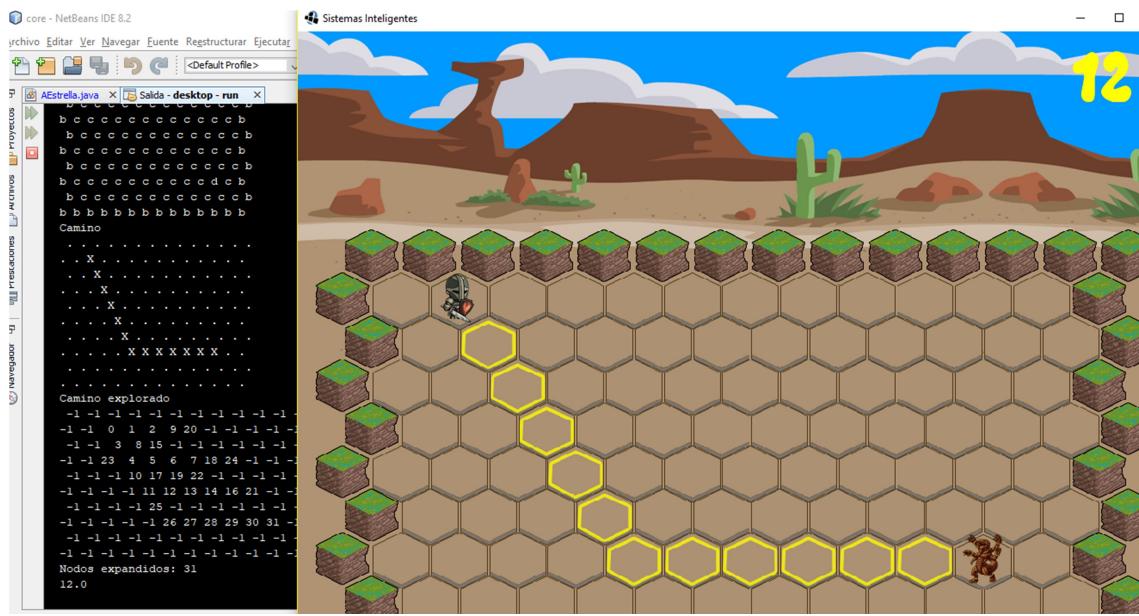
Heurística Manhattan adaptada a mapas hexagonales



Heurística Euclídea adaptada a mapas hexagonales

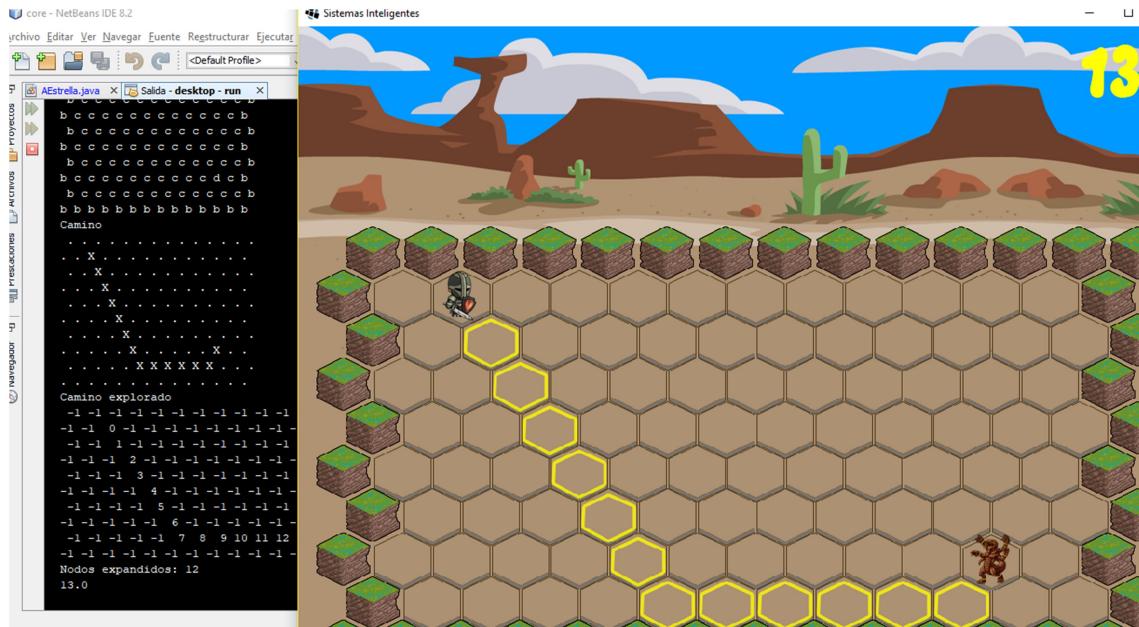


Heurística Euclídea

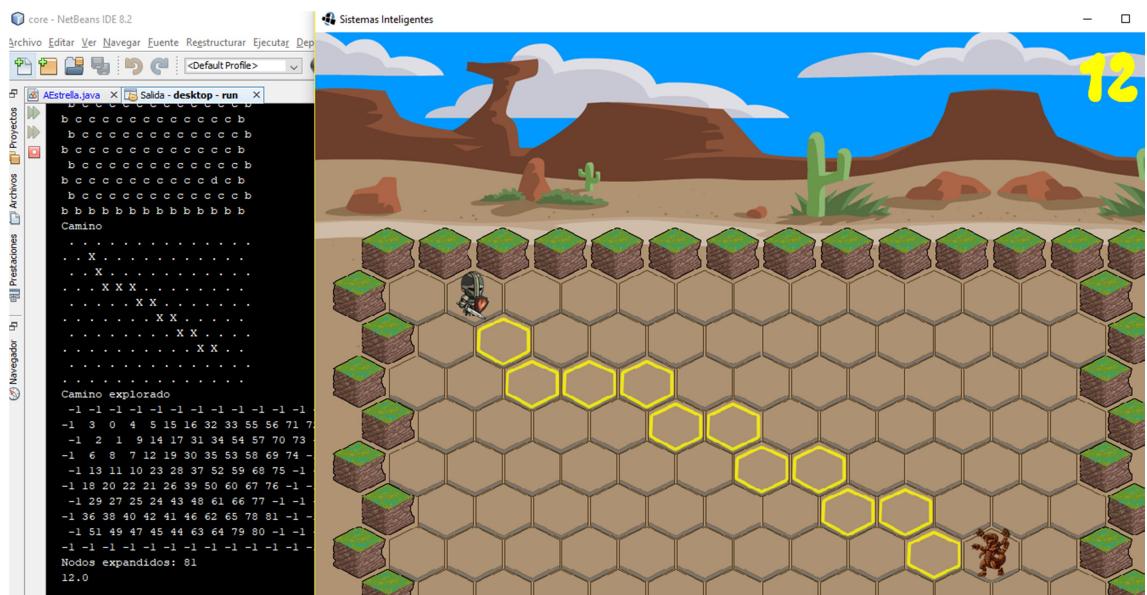


Heurística Manhattan

Como podemos observar , la heurística Manhattan es inadmisible en este mapa.

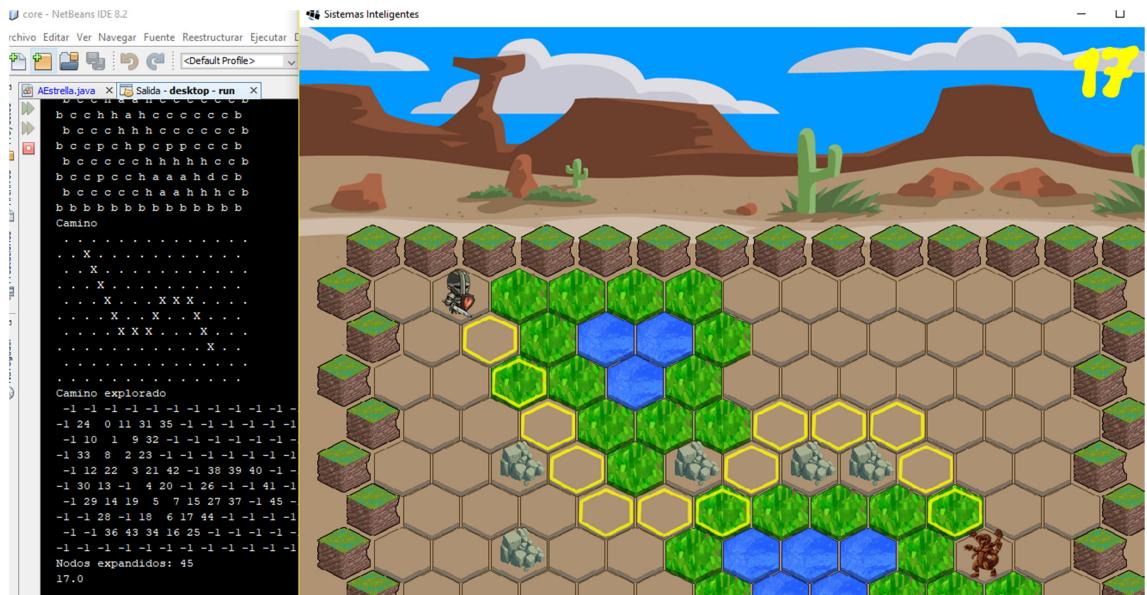


Heurística h=0

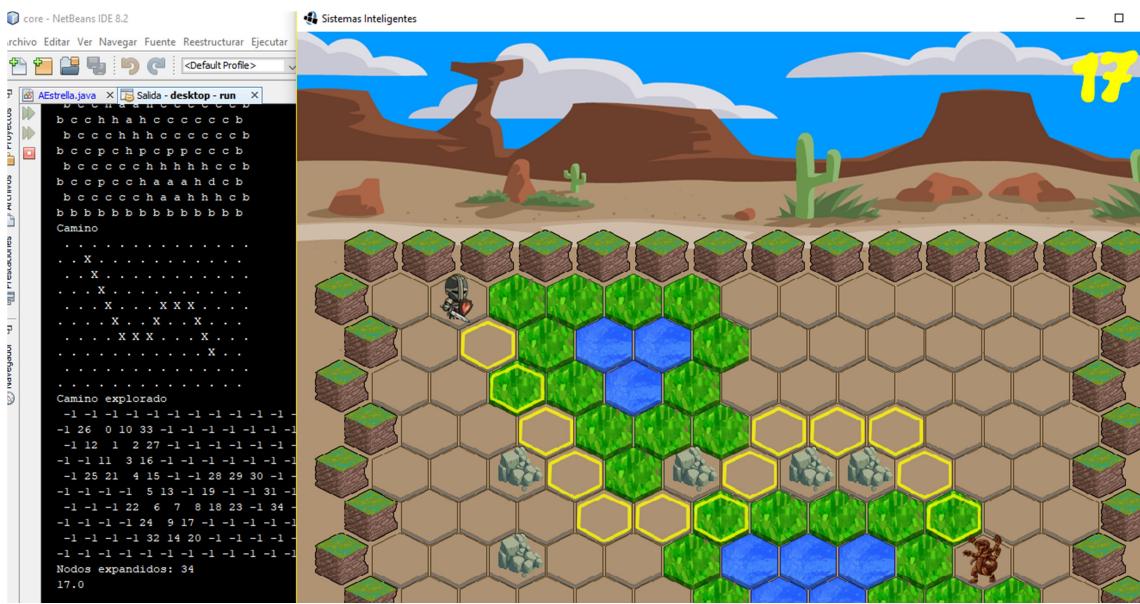


Mundo_01

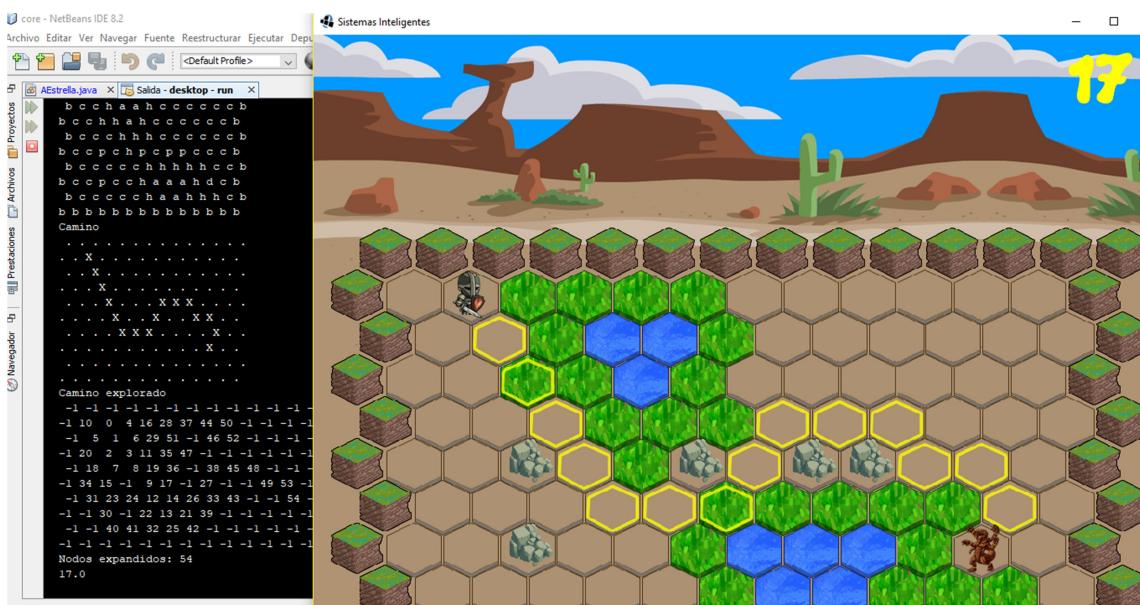
Heurística Manhattan adaptada a mapas hexagonales



Heurística Euclídea adaptada a mapas hexagonales

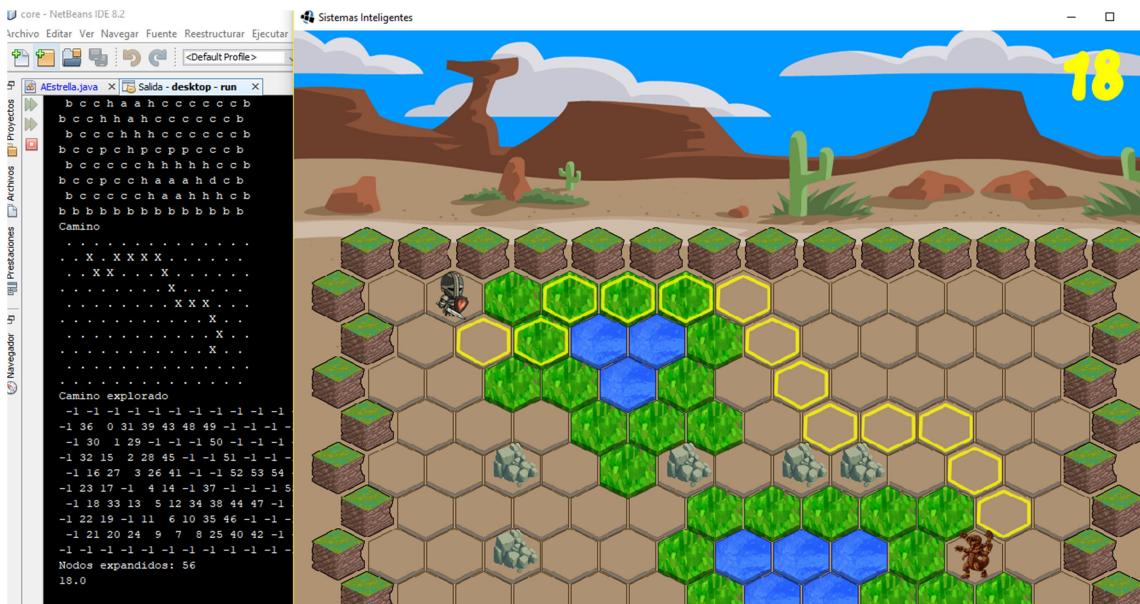


Heurística Euclídea

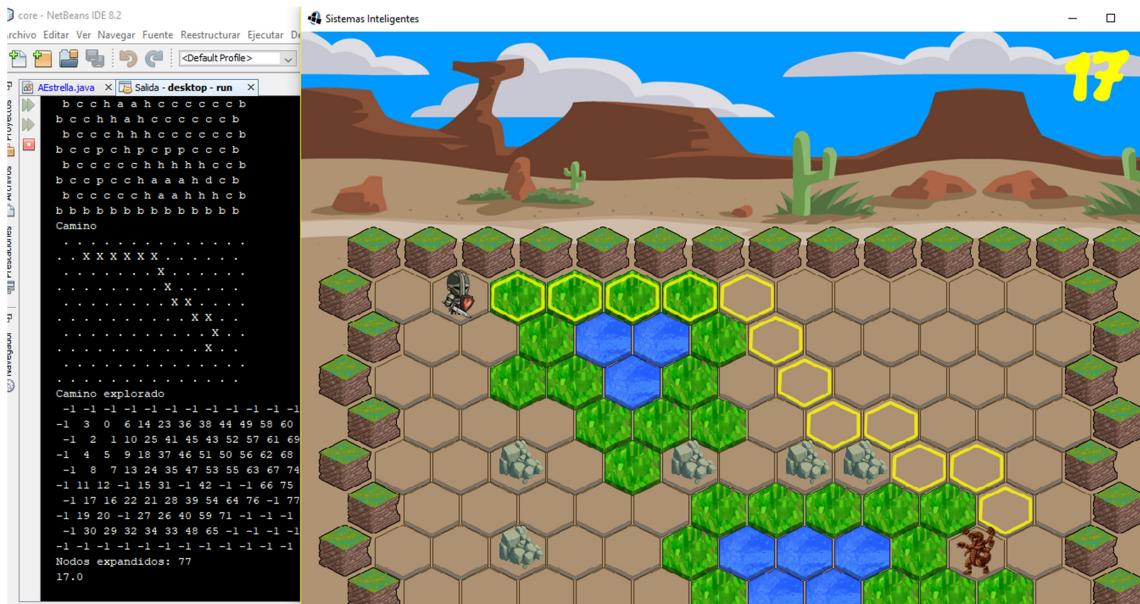


Heurística Manhattan

Como podemos comprobar, otra vez inadmisible.

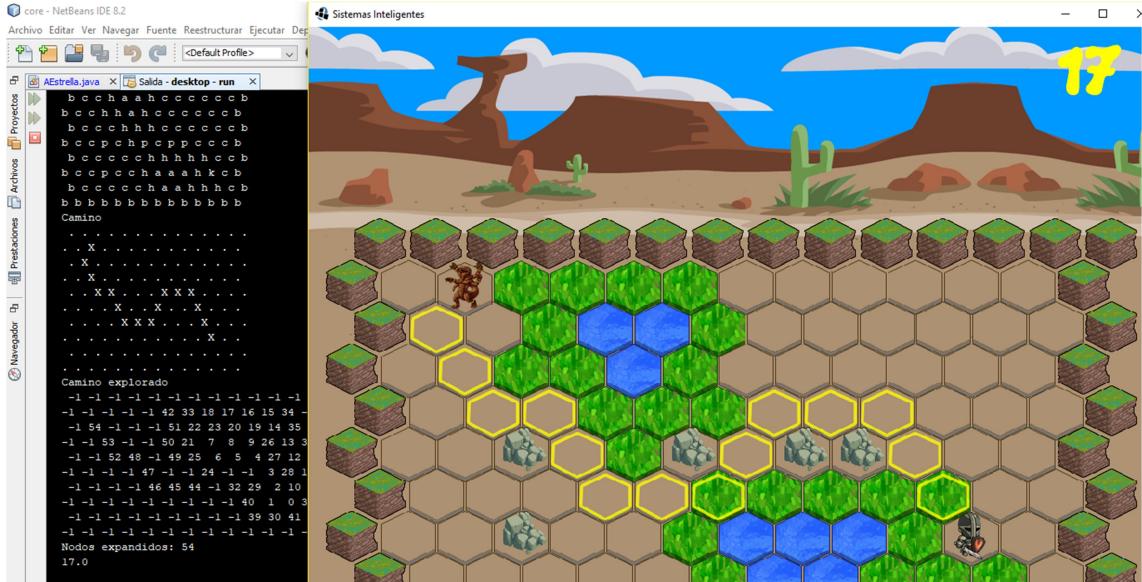


Heurística $h=0$



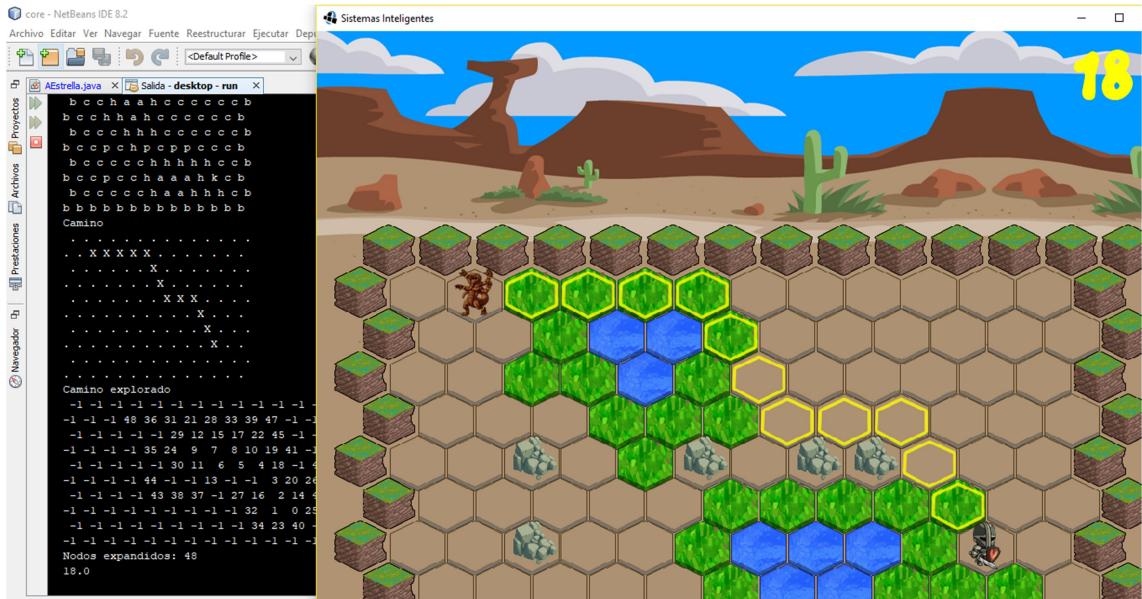
mundo_02 : cambiando de lugar el caballero y el dragón.

Heurística Manhattan adaptada a mapas hexagonales

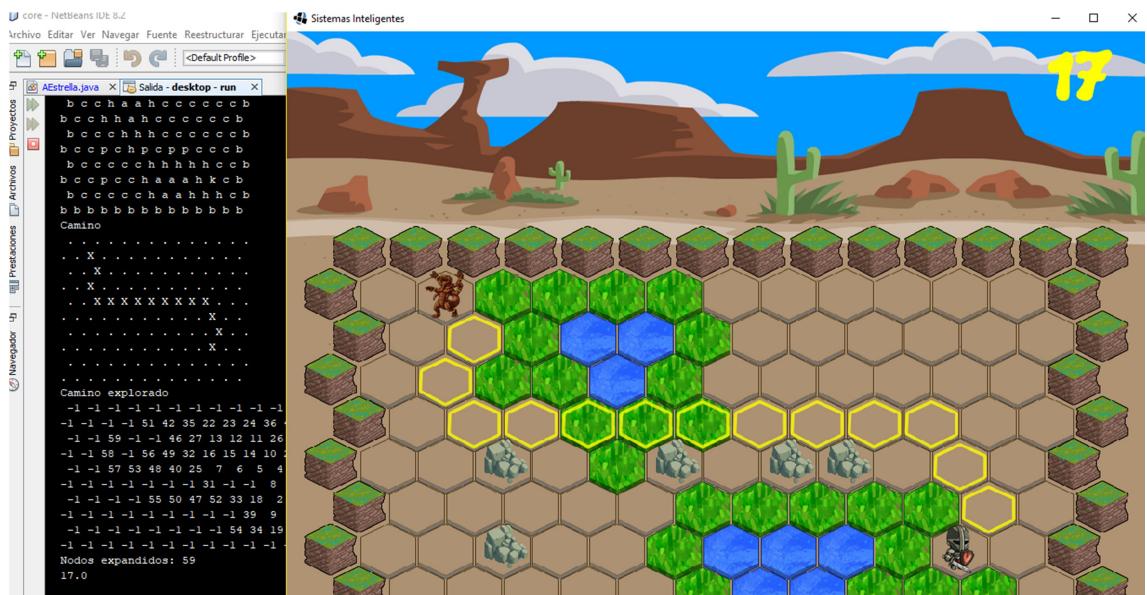


Heurística Euclídea adaptada a mapas hexagonales

Como podemos ver, expande menos nodos que la Manhattan HEX, pero no es admisible en este mapa.

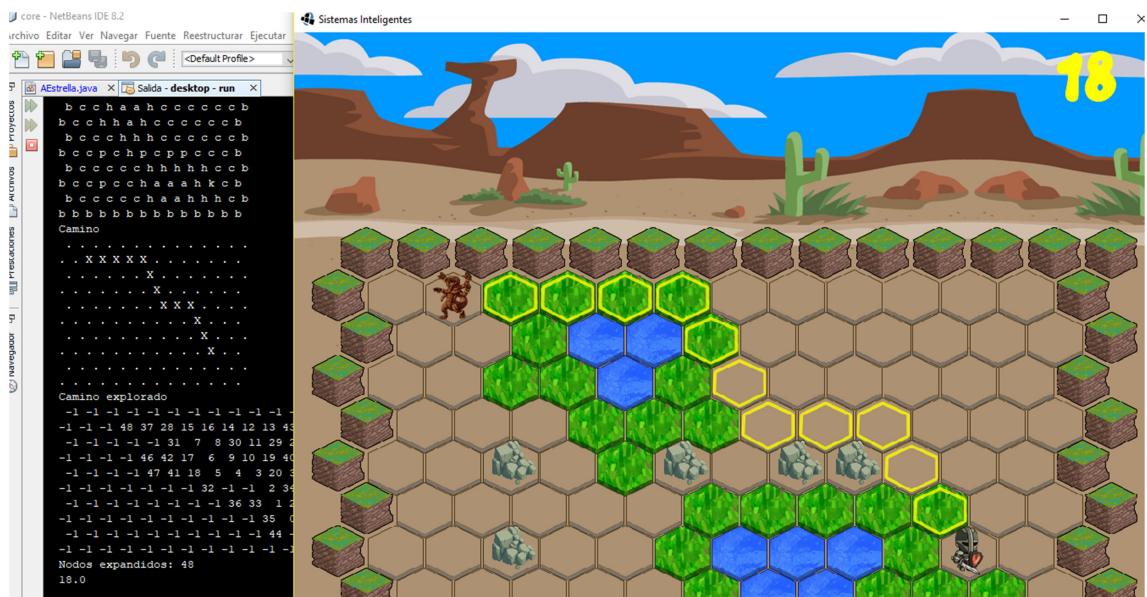


Heurística Euclidea

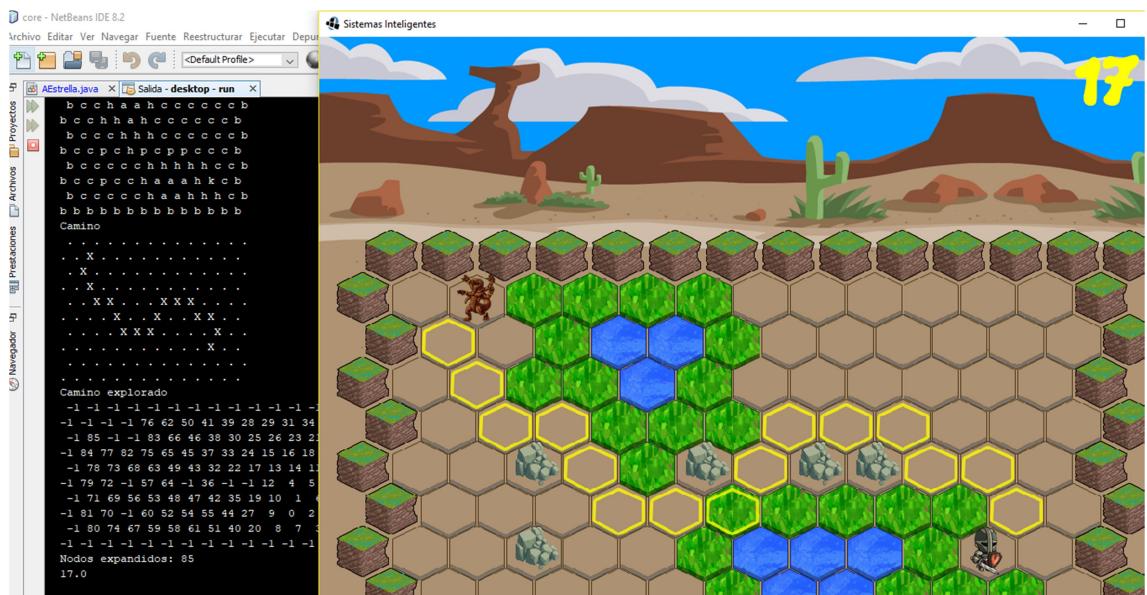


Heurística Manhattan

Otra vez inadmissible.

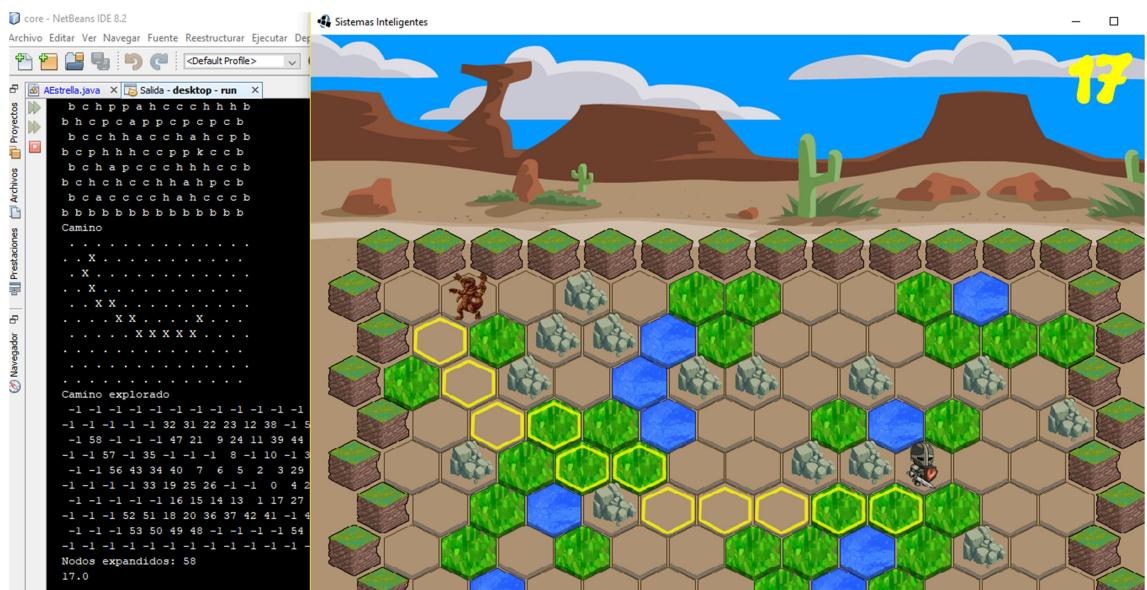


Heurística h=0

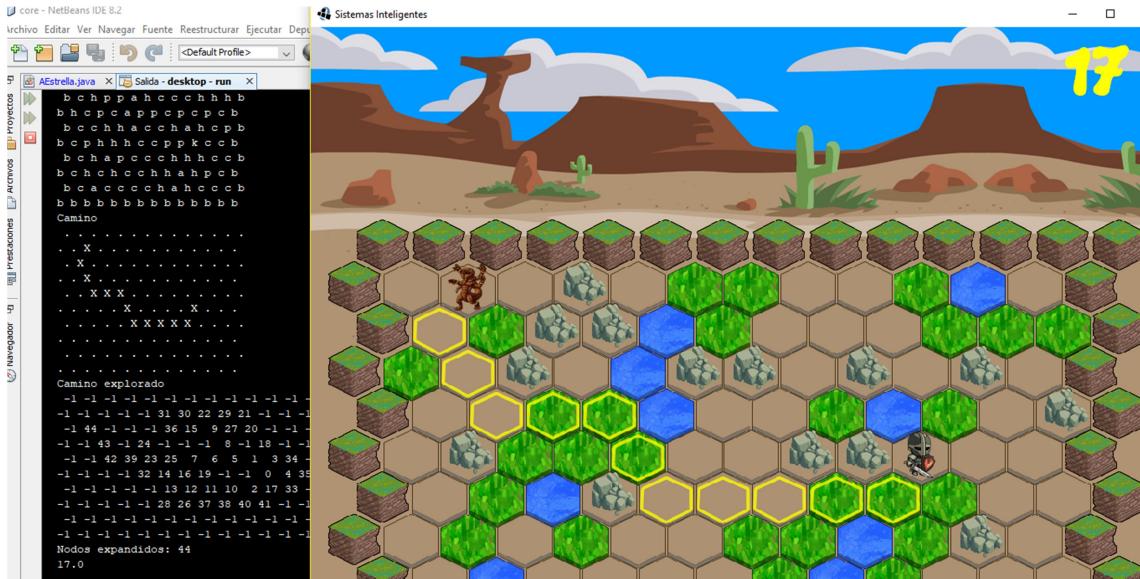


Mundo_03

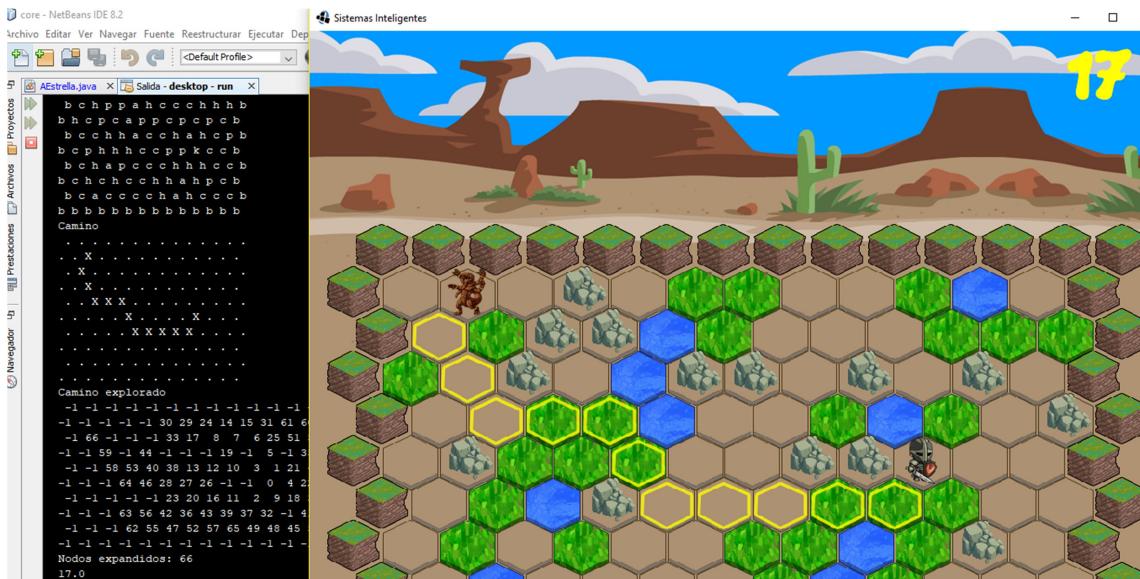
Heurística Manhattan adaptada a mapas hexagonales



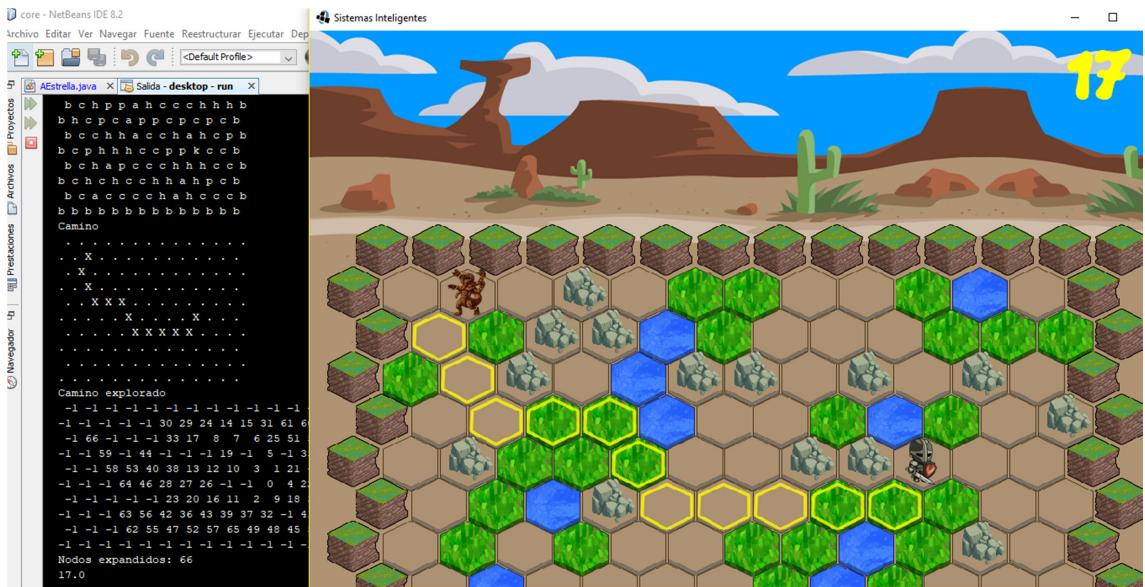
Heurística Euclídea adaptada a mapas hexagonales



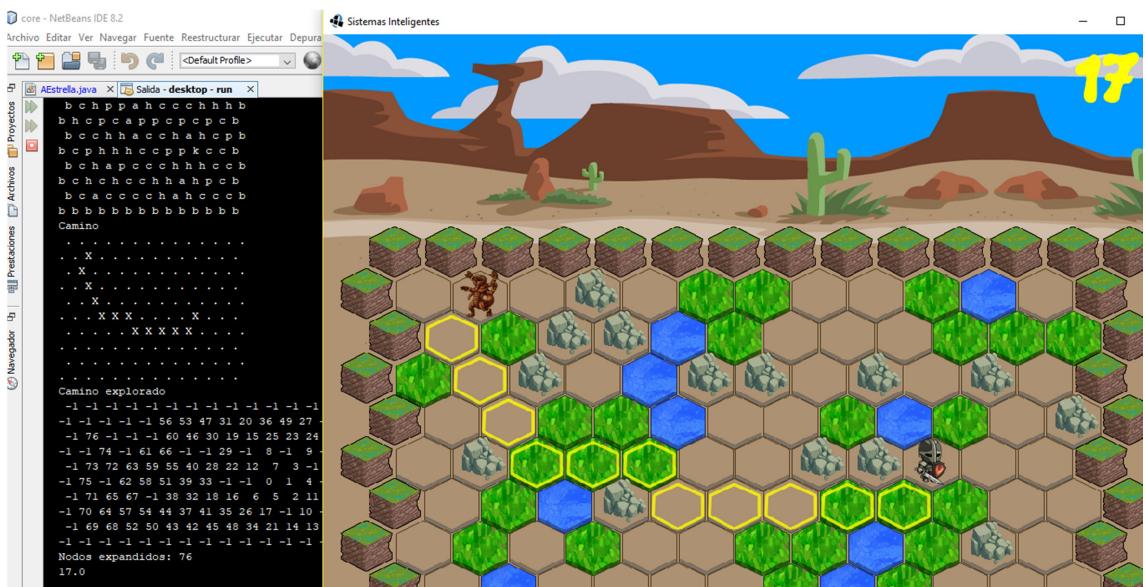
Heurística Euclídea



Heurística Manhattan



Heurística h=0



Bibliografía

<http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>

https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%A1squeda_A%2A

<https://www.redblobgames.com/grids/hexagons/pre-index.html>