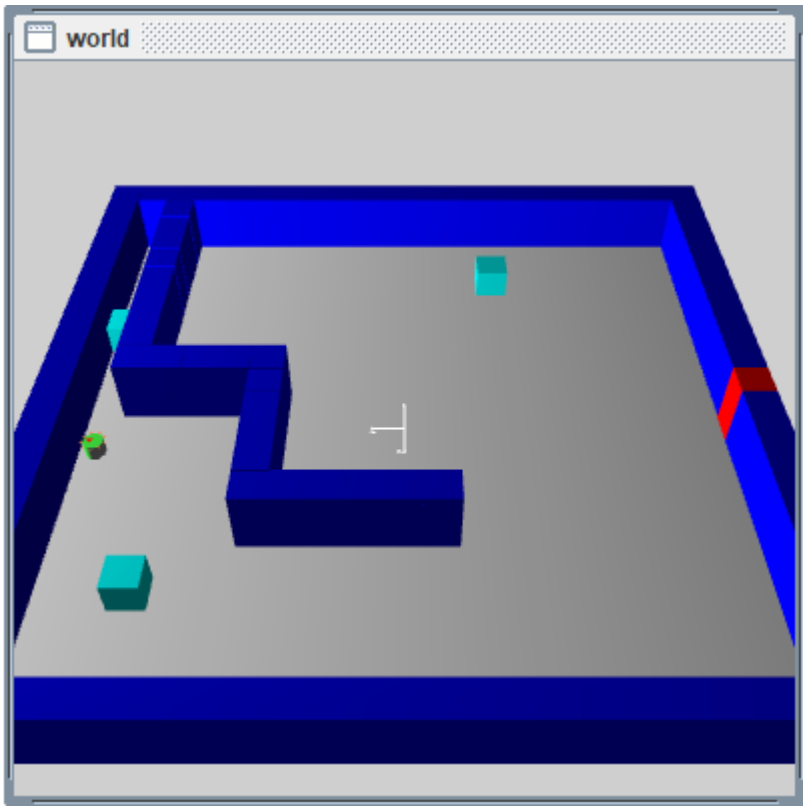
PRÁCTICA 1
Búsqueda y sistemas probabilísticos



Alejandro Jesús Aliaga Hyder 48765284V

Grupo 6 práticas
Viernes 09:00 - 11:00

ÍNDICE

1. INTRODUCCIÓN A*

En esta primera prática, de la asignatura sistemas inteligentes, tenemos como objetivo comprender el funcionamiento de la búsqueda heurística, y en concreto del algoritmo A*.

La implementación del algoritmo A*, conlleva la búsqueda de una heurística apropiada al problema, por lo que tendremos que compreneder que es una heurística y analizar nuestro problema para saber cual será la mejor posible.

Para ello, primera se hará una explicación general en que consiste el algoritmo a aplicar, luego encontrar los posibles casos que se darán durante la ejecución, y para terminar como obtener la mejor heurísttica.

2. ALGORITMO A*

Los algoritmos A, presentan una función de evaluación de la forma f(n) = g(n) + h(n), done g(n) representa la estimación del coste del camino de coste mínimo desde el estado inicial hasta el nodo n, y h(n) es la estimación del coste del camino de coste mínimo desde n hasta algún nodo meto, esta función incluye el conocimiento heurístico sobre el problema a resolver.

Este algoritmo lo que hace es recorrer todos los posibles estados y comprobar cual de ellos es la mejor solución para llegar al destino. Debido a que recorren todos los estados, computacionalmente es muy lento y cuando los problemas son muy grandes imposibles de llevar a cabo, pero a cambio de asegurar una solución óptima.

2.1 Búsqueda

Los elementos que usa el algoritmo para ir buscando la solucion son:

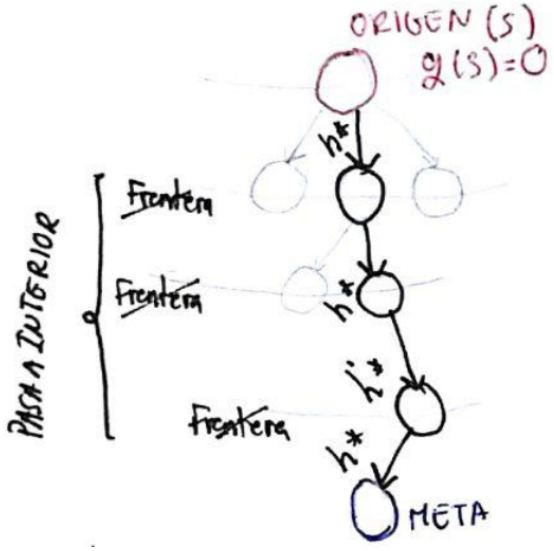
- Un estado inicial S, desde el cual partirá el algoritmo
- Una lista interior con los nodos ya expandidos
- Una lista frontera, con los nuevo nodos a los que se podría ir

La búsqueda trata de ir llegando a nuevos estados, obteniendo los sucesores al estado en el que se encuentra, tras obtenerlos, de los disponibles comprobamos cual de ellos implica menos coste y se selecciona. Tras llegar a un nuevo estado, este se añade a la lista interior.

Es necesario añadir los nodos expandidos a la lista interior, ya que al obtener los sucesores del estado actual debemos comprobar si ya han sido expandidos al llegar desde otro nodo. Si no hacemos esto se puede hacer un bucle infinito.

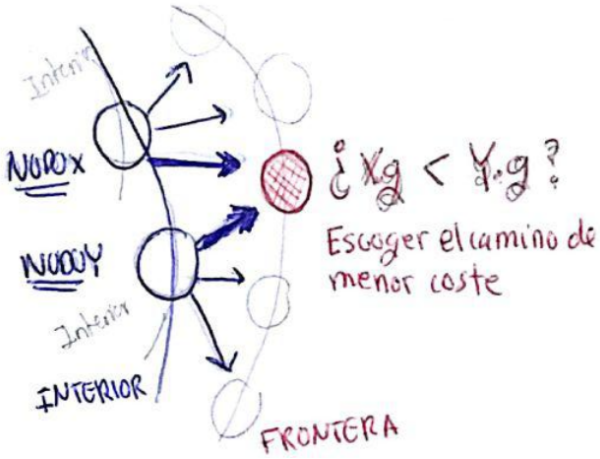
2.2 Posibles casos a encontrar

El primer caso: es una exploración en la que no encontrarmos ningún problema:



Es la situación explicada anteriormente, obtenemos los sucesores del nodo en el que nos encontramos, y añadimos a lista frontera los nodos que no estén en lista interior.

El segundo caso: durante la exploración uno de los nodos ya se encontraba en lista frontera.



En esta situación a la expandir uno de los nodos ya se encuentra en lista frontera. Esto nos ayuda ya que implican distintos camino para llegar a un nodo. Durante la ejecución del programa podremos elegir de ellas cual es de menor coste.

Tercer caso: nos quedamos sin nodos en la lista frontera

Esto puede ser causa de errores en la implenentación o lógica del algoritmo. Una mala heurística puede llevarnos a explorar nodos que no nos beneficie.

3. PSEUDOCÓDIGO DEL A*

A continuación voy a explicar el algoritmo mediante el pseudocódigo.

```
INICIO algoritmo A*

    // inicializar la lista interior con el primer nodo a expandir, el origen
    listaInterior.add( nodoOrigen )

    // inicializar lista frontera con los sucesores del nodo inicial
    listaFrontera.add ( hijos( nodoOrigen) )

    // explorar todos los nodos, es decir, añadir todos a la lista interior porque
    // estos han sido expandidos sin encontrar el nodo destino, implica que el algoritmo
    // ha fallado. En caso contrarió al encontrar el nodo se terminará el algoritmo
    MIENTRAS listaFrontera no esté vacia

        // obtenemos el nodo de menor f(n), este nodo será el nuevo nodo al que iremos
        // y expandiremos, es decir, obtener los sucesores
        n = listaFrontera con menor f(n)

        SI n es meta
            // si hemos encontrado el nodo meta fin del algoritmo
        FIN SI

        // una vez expandido el nodo n, este ya ha sido explorado por lo que se
        // añadirá a la lista interior y se borrará de la lista frontera
        listaFrontera.borrar( n );
        listaInterior.add( n );

        // para cada sucesor de n obtenido que no esté en lista interior, ya que si no
        // causaremos un bucle infinito...
        PARA CADA hijo m de n que no estén en lista interior

            // Guardamos la g(m) del sucesor a explorar
            g(m) = n.g + costede n a m

            // si el hijo m no ha sido expandido anteriormente desde otro nodo
            // tendremos que añadirlo a al lista frontera
            SI m no está en lista frontera
                m.n = calcularHeurística(origen, destino)
                m.padre = n
            FIN SI
```

```
            // si m se encuentra en lista frontera, implica que tendremos otro camino hacia m.
            // para comprobar cual de ellos es mejor, hay que comprarar el coste
            // del nuevo camino con el del coste del nodo padre de m.
            SINO SI n.g < m.padre.g
                m.padre = n
                m.g = n.g + coste de n hasta n
            FIN SI

        FIN PARA CADA

    FIN MIENTRAS
```

4. HEURÍSTICA

En programación se dice que un algoritmo es heurístico cuando la solución no se determina de forma directa, sino mediante pruebas y ensayos.

Por tanto la heurística consiste en generar candidatos de soluciones posibles a un patrón, los candidatos son sometidos a pruebas de acuerdo a un criterio. Cuando uno de los estados no es viable se genera y comprueba otro. Esto es lo que hacemos al ir expandiendo la lista frontera.

Por tanto la heurística se usa para controlar el comportamiento del A*. Siendo esta es la información a tener en cuenta a la hora de diseñar la heurística:

- Si h(n) es 0, tan solo g(n) se usará para calcular f(n), lo que nos llevaría a una búsqueda en profundidad, garantizando el camino más corto que se puede encontrar

- si h(n) es siempre menor al coste de mover n a la meta, A* garantiza encontrar un camino corto, esta es la opción que más nodos expande.

- Si h(n) es igual al coste de mover n a la meta, expande el camino de menor coste, siendo muy rápido.

- Si h(n) es a veces mayor que el coste de mover n a la meta, A* no garantiza encontrar el camino más corto, pero puede ser más rápido.

- Por último si h(n) es muy alto en relación a g(n), entonces solo h(n) influye, y A* se convierte en un algoritmo voraz.

4.1 Heurística más apropiada para nuestro programa

Teniendo en cuenta los puntos explicados anteriormente, para esta parte de la práctica necesitamos calcular una heurística que de valores por debajo del coste del nodo n a meta, para garantizar encontrar un camino corto.

En cuanto al cálculo de heurísticas en cuadrículas, hay algunas bien conocidas para usar:

- Cuando solo se permiten 4 direcciones, **Manhattan distance**
- Cuando se permiten 8 direcciones, **Distancia diagonal**
- Cuando permites cualquier dirección, **Distancia Euclidea**

Como en nuestro programa solo hemos permitido 4 direcciones, la distancia Manhattan es la más adecuada. Ya que si escogemos la diagonal o euclidea tendremos encuenta el coste de direcciones que nunca haremos.
A continución voy a hacer una comparativa de las tres heurísticas comentadas, en mi implementación del algortimo A*, siendo las fórmulas usadas las siguientes:

El cálculo de la **distancia manhattan** heurística es:

```
dx = abs( origen.x – destino.x )
dy = abs( origen.y – destino.y )

return dx + dy;
```

El cálculo de la **distancia diagonal** es:

```
dx = abs( origen.x – destino.x )
dy = abs( origen.y – destino.y )
D = D2 = 1

return D * ( dx + d y )  + ( D2 – 2 * D) * min(dx, dy) ;
```

El cálculo de la **distancia Euclidea** es:

```
dx = abs( origen.x – destino.x )
dy = abs( origen.y – destino.y )

return sqrt( dx * dx + dy * dy )
```

Aplicando distancia Manhattan

```
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
.xxxxx............x.
.....x............x.
.....x............x.
.....x............x.
.....x............x.
.....xxxxxxxxxxxxxx.
....................
....................
....................
....................
```

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

Aplicando distancia diagonal

```
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
.xxxxx......xxxxxxx.
.....x......x.......
.....x......x.......
.....x......x.......
.....x......x.......
.....xxxxxxxx.......
....................
....................
....................
....................
```

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

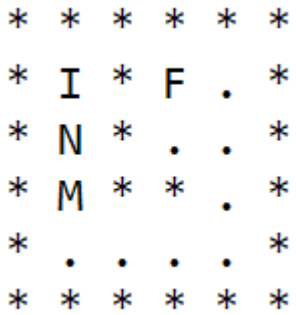
Distancia Euclidea

```
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
.xxxxx............x.
.....x...........xx.
.....x..........xx..
.....x.........xx...
.....x........xx....
.....xxxxxxxxxx.....
....................
....................
....................
....................
```

```
-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 -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 17 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 11 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 1 2 3 4 -1 -1 -1 -1 -1 -1 98 100 102 104 106 108 109 -1
-1 6 7 8 9 10 -1 -1 -1 -1 -1 -1 78 80 82 84 88 90 107 -1
-1 12 13 14 15 16 -1 -1 -1 -1 -1 -1 61 64 67 71 75 87 105 -1
-1 18 19 20 21 22 -1 -1 -1 -1 -1 91 52 55 58 60 70 83 103 -1
-1 23 24 25 26 27 -1 -1 -1 -1 -1 -1 43 47 50 57 66 81 101 -1
-1 28 29 30 31 32 33 34 35 36 37 38 39 41 46 54 63 79 99 -1
-1 40 42 -1 44 45 48 49 51 53 56 59 68 74 86 93 -1 -1 -1 -1
-1 62 65 69 72 73 76 77 85 89 92 96 -1 -1 -1 -1 -1 -1 -1 -1
-1 94 95 97 -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
```

5. TRAZA DEL ALGORITMO EN UN EJEMPLO

Para hacer la traza del algoritmo más sencilla, voy a usar un mapa parecido al de ejemplo en el pdf de la práctica pero con unas dimensiones mucho menores. Siendo lf y li, lista frontera y lista interior:



Lf (2,1)
Li (1,1)

N=(2,1)
n.g = 0

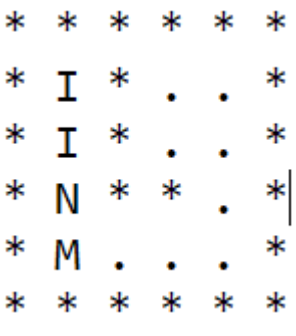
Lf.delete(n)
Li.add(n)

M = (3,1)

M.g = n.g + 1 = 1

m.h = heuristica(m)
m.padre = n

lf.add(m)



Lf~~(2,1)~~(3,1)
Li (1,1)(2,1)

N=(3,1)

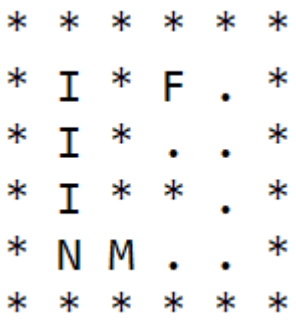
Lf.delete(n)
Li.add(n)

M = (4,1)

M.g = n.g + 1 = 2

m.h = heuristica(m)
m.padre = n

lf.add(m)



Lf~~(3,1)~~(4,1)
Li (1,1)(2,1)(3,1)

N=(4,1)

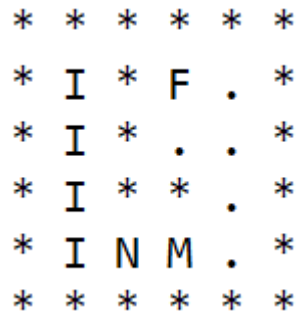
Lf.delete(n)
Li.add(n)

M = (4,2)

M.g = n.g + 1 = 3

m.h = heuristica(m)
m.padre = n

lf.add(m)



Lf~~(4,1)~~ (4,2)
Li (1,1)(2,1)(3,1)(4,1)

N=(4,2)

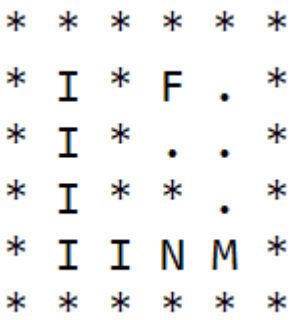
Lf.delete(n)
Li.add(n)

M = (4,3)

M.g = n.g + 1 = 4

m.h = heuristica(m)
m.padre = n

lf.add(m)



Lf~~(4,2)~~ (4,3)
Li (1,1)(2,1)(3,1)(4,1)(4,2)

N=(4,3)

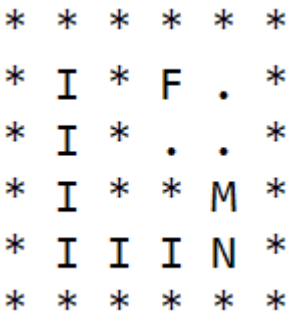
Lf.delete(n)
Li.add(n)

M = (4,4)

M.g = n.g + 1 = 5

m.h = heuristica(m)
m.padre = n

lf.add(m)



Lf~~(4,3)~~ (3,4)
Li (1,1)(2,1)(3,1)(4,1)(4,2)
(4,3)

N=(3,4)

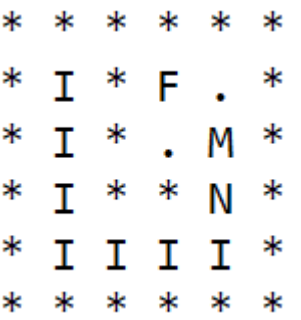
Lf.delete(n)
Li.add(n)

M = (2,4)

M.g = n.g + 1 = 6

m.h = heuristica(m)
m.padre = n

lf.add(m)



Lf~~(3,4)~~ (2,4)
Li (1,1)(2,1)(3,1)(4,1)(4,2)
(4,3)(3,4)

N=(2,4)

Lf.delete(n)
Li.add(n)

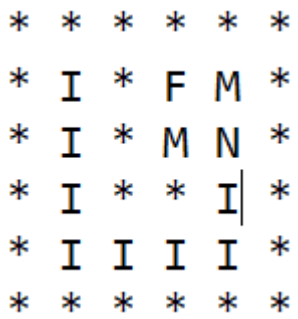
M = (1,4), (2,3)

M.g = n.g + 1 = 7

m.h = heuristica(m)
m.padre = n

lf.add(m)

CRITERIO: nodo superior primero



Lf~~(2,4)~~ (2,3)(1,4)
Li (1,1)(2,1)(3,1)(4,1)(4,2)
(4,3)(3,4)(2,4)

N=(1,4)

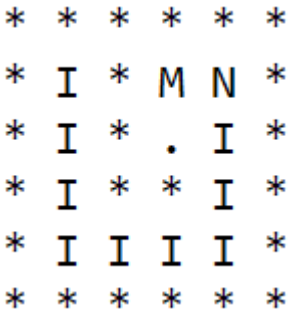
Lf.delete(n)
Li.add(n)

M = (1,3)

M.g = n.g + 1 = 8

m.h = heuristica(m)
m.padre = n

lf.add(m)



Lf~~(1,4)~~ (2,3)(1,3)
Li (1,1)(2,1)(3,1)(4,1)(4,2)
(4,3)(3,4)(1,4)

N=(1,4)

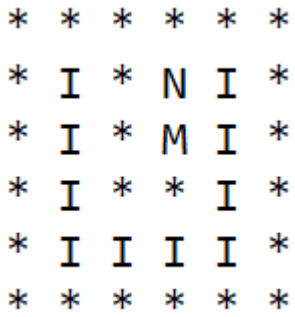
Lf.delete(n)
Li.add(n)

M = (1,3)

M.g = n.g + 1 = 9

m.h = heuristica(m)
m.padre = n

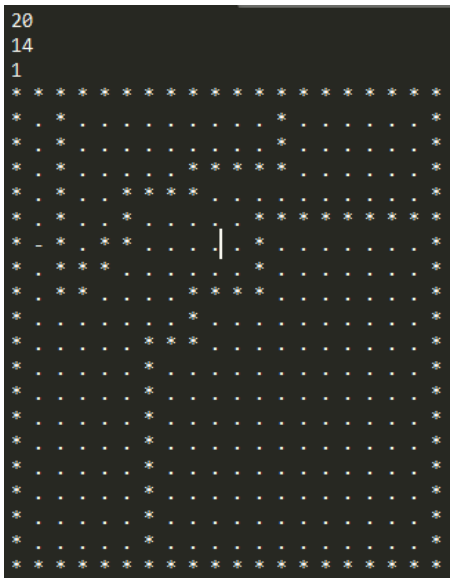
lf.add(m)

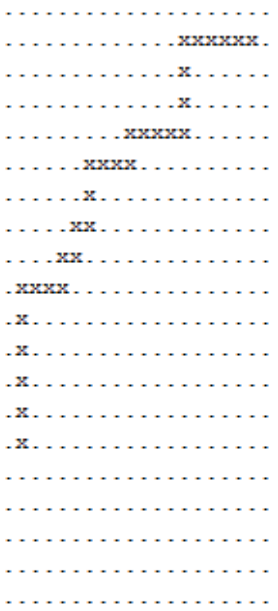


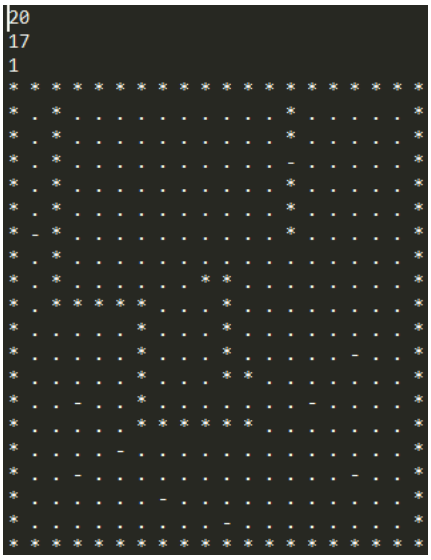
N = nodoMeta => FIN A*

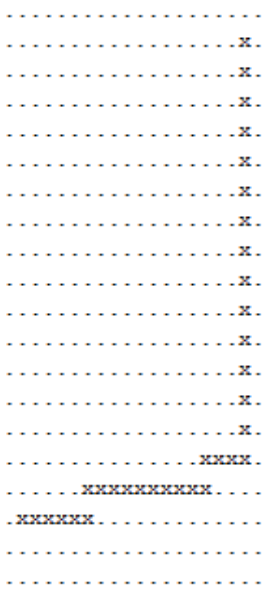
6. PRUEBAS UNITARIAS

Para realizar las pruebas unitarias en las que se de toda los posibles casos. Voy a hacer 3 mapas propios, mostrando la solución y la cantidad de nodos expandidos. En estos mapas se darán los tres casos en el punto anterior.

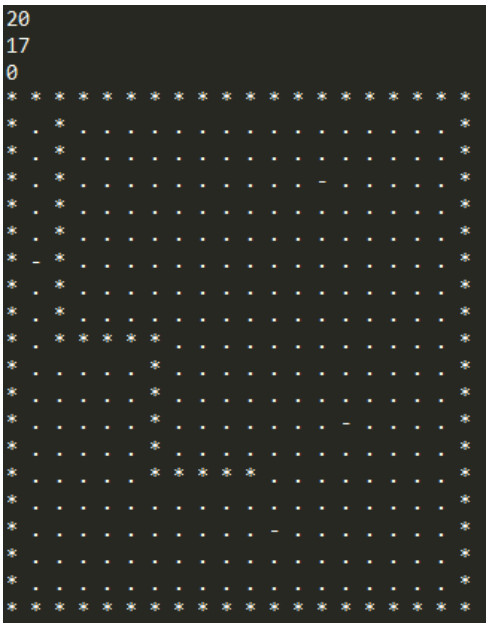


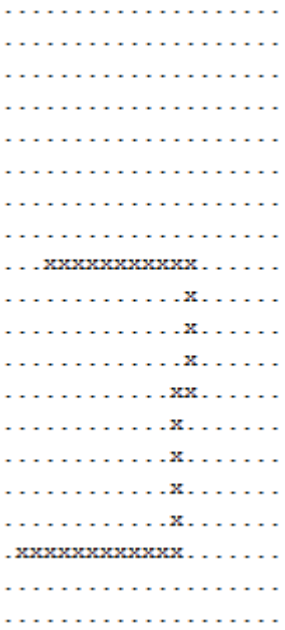






Posicion imposible de alcanzar.





Matriz de expansión, recorre todo el mapa

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 -1
-1 -1 -1 275 262 246 217 184 168 152 123 113 112 120 91 69 51 68 33 -1
-1 -1 -1 280 267 256 232 192 176 157 141 114 111 134 94 72 50 67 32 -1
-1 -1 -1 283 274 261 245 216 183 167 151 122 110 -1 119 90 49 66 31 -1
-1 -1 -1 286 279 266 255 231 191 175 156 140 109 107 133 89 48 65 30 -1
-1 -1 -1 288 282 273 260 244 215 182 166 150 121 106 132 88 47 64 29 -1
-1 -1 -1 290 285 278 265 254 230 190 174 155 139 105 131 87 46 63 28 -1
-1 238 -1 291 287 281 272 259 243 214 181 164 138 104 130 86 45 62 27 -1
-1 237 -1 292 289 284 277 264 253 229 189 163 137 103 129 85 44 61 26 -1
-1 236 -1 -1 -1 -1 -1 271 258 241 188 162 136 102 128 84 43 60 25 -1
-1 235 207 203 199 200 -1 276 263 240 187 161 135 101 127 83 42 59 24 -1
-1 248 219 205 198 201 -1 270 252 212 173 149 108 100 126 82 41 58 23 -1
-1 224 234 206 197 202 -1 269 251 211 172 148 98 99 -1 81 40 57 22 -1
-1 223 247 218 196 204 -1 268 250 210 171 147 97 118 75 80 39 56 21 -1
-1 222 225 233 194 195 -1 -1 -1 -1 -1 146 96 117 74 79 38 55 20 -1
-1 221 226 242 193 213 177 180 158 165 142 145 95 116 73 78 37 54 19 -1
-1 220 227 208 185 178 169 159 153 143 124 -1 92 76 70 52 35 34 18 -1
-1 -1 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 -1
-1 257 249 239 228 209 186 179 170 160 154 144 125 115 93 77 71 53 36 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

7. SISTEMA EXPERTO DIFUSO

En esta segunda parte de la prática vamos a elaborar un sistema experto difuso que recorra el camino obtenido por el A*. El lenguaje de programación usado es FCL (Fuzzy Controller Language).

Sistema experto difuso: en inteligencia artificial, un sistema experto es un sistema computacional que emula la capacidad de tomar decisiones de un humano experto.

Fuzzy Controller Language: lenguage basado en la lógica fuzzy, es un sistema matemático que analiza los valores de las entradas en términos de variables lógicas que toman valores continuos entre 0 y 1.

7.1 ESPECIFICACIÓN DE LOS CONJUNTOS PARA LAS VARIABLES

El controlador difuso consta de **10 variables de entrada, 9 sensores y una variable "sig", y de dos variables de salida, "vel" y "rot"**. Como se ha explicado anteriormente a cada variable se le asigna un grado de pertenencia entre 0 y 1.

Sensores (s0 hasta s8): se utilizan para determinar la distancia a otros objetos. Toman un rango de valores entre 0 y 1,5. Siendo 0 que está pegado al objeto y 1,5 no detecta ningun objeto.

Los sensores están compuestos por tres términos, a los que asignamos tuplas con el valor del sensor y el grado de pertenencia. Por ejemplo:

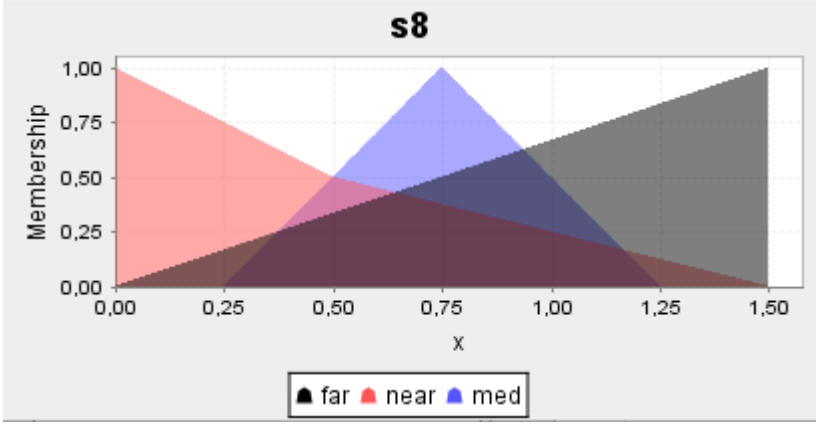
Term near := (1.5, 0); Para la distancia 1.5 metros (no se detecta nada), se le asigna un grado de pertenencia 0 ya que no pertenece a la definición del término cerca.

La cantidad de términos y valores asignados a cada término, son los mismos en todos los sensores. No han variado respecto al sensor dado de ejemplo. Un ejemplo de sensor sería el siguiente:

```
FUZZIFY s7
    TERM near   := (0,1) (0.5,0.5) (1,0.25) (1.5,0);
    TERM med    := (0.25,0) (0.75,1) (1.25,0) (1.5,0);
    TERM far    := (0,0) (1.5,1);
END_FUZZIFY

FUZZIFY s8
    TERM near   := (0,1) (0.5,0.5) (1,0.25) (1.5,0);
    TERM med    := (0.25,0) (0.75,1) (1.25,0) (1.5,0);
    TERM far    := (0,0) (1.5,1);
END_FUZZIFY
```

Para cada implementación de los sensores la gráfica que muestra la relación entre el grado de pertenencia y la distancia es la siguiente:



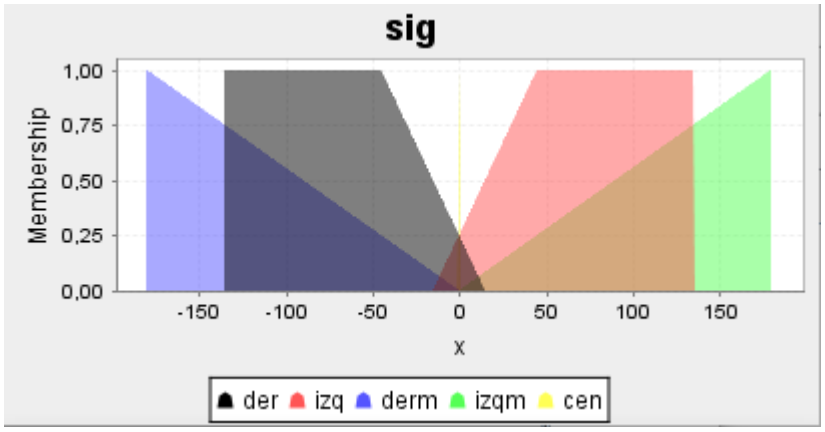
Esta gráfica sirve para ver aquellos rangos de valores en lo que no asginamos términos o si los valores de cada término se superponen demasiado entre ellos lo que llevaría a un mal funcionamiento de la variable. De esta forma he ido ajustando las tuplas.

Variable "sig": determina lo que debe girar el robot para encarar la parte delantera a la siguiente dirección en la que se encuentra la "X" del camino, toma valores entre -180 a 180.

A esta variable le he asignado los mismos términos dados en la variable "rot" de ejemplo, para poder asignar en las reglas a cada rotación su respectivo término de "sig". El valor de las tuplas grados y pertenencia, los he calculado a partir de los extremos usando reglas de tres, los cuales posteriormente he ido ajustando cuando he hecho las reglas.

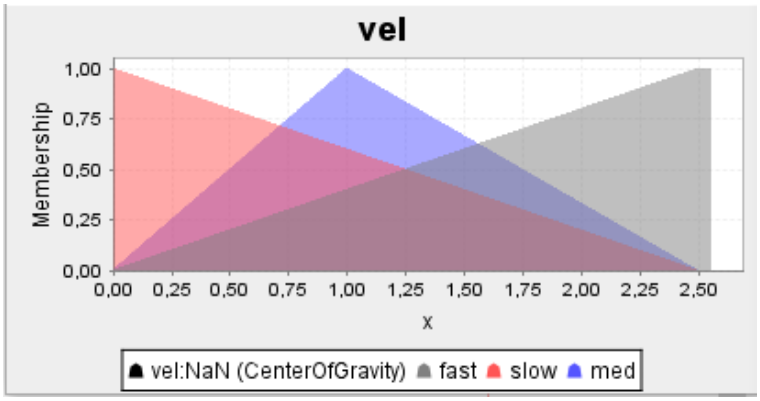
```
FUZZIFY sig
//                (gradosGiro, pertenencia)
    TERM izqm   := (0,0) (180,1);
    TERM izq    := (-15,0) (45,1) (90,1) (135,1) (135,0); //
    TERM cen    := (-1,0) (0,1) (1,0);
    TERM der    := (-135,0) (-135,1) (-90,1) (-45,1) (15,0);
    TERM derm   := (-180,1) (0,0);
END FUZZIFY
```

Gráfica de evolución de la variable "sig":



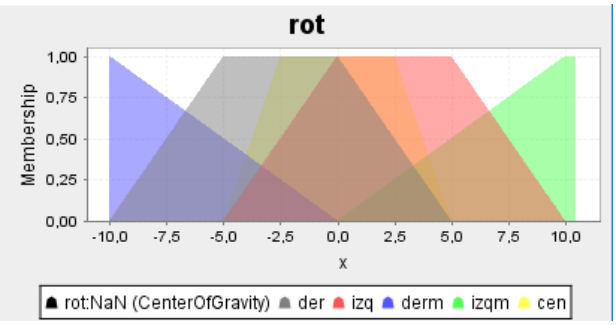
Variable "vel": velocidad a la que va el robot. En esta no he ajustado los valores de las tuplas dadas como ejemplo porque me han parecido correctos para el desarrollo de la prática.

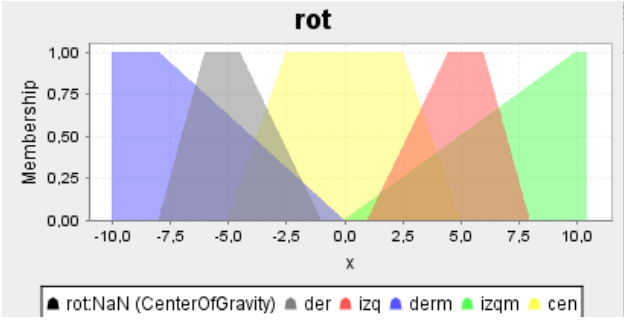
```
DEFUZZIFY vel
    TERM fast   := (0,0) (2.5,1);
    TERM med    := (0,0) (1,1) (2.5,0);
    TERM slow   := (0,1) (2.5,0);
    // Use 'Center Of Gravity' defuzzification method
    METHOD : COG;
    // Default value is 1 (if no rule activates defuzzifier)
    DEFAULT := 1;
END_DEFUZZIFY
```



Variable "rot": velocidad rotacional en radianes por segundo. A esta variable he modificado los valores de las tuplas para adecuarlo al resto de variables ("sig").

A continuación se pueden ver en la izquierda la gráfica dada como ejemplo y a la derecha la gráfica usada en la entrega de la prática.





El código de la variable es el siguiente:

```
DEFUZZIFY rot
    TERM izqm   :=    (-8,0) (0,0) (10,1);
    TERM izq    :=    (1,0)(4.5,1) (6,1) (8,0);
    TERM cen    :=    (-10,0) (-5,0) (-2.5,1) (2.5,1) (5,0);
    TERM der    :=    (-8,0) (-6,1) (-4.5,1) (-1,0);
    TERM derm   :=    (-8,1) (0,0) (10,0);

    // Use 'Center Of Gravity' defuzzification method
    METHOD : COG;
    // Default value is 1 (if no rule activates defuzzifier)
    DEFAULT := 0;
END_DEFUZZIFY
```

7.2 EXPLICACIÓN DE LAS REGLAS ESPECIFICADAS

Ahora que ya están hechas las variables, hay que usarlas para establecer reglas con las que el robot se va a mover por el camino óptimo calculado en el A*.

Las primeras que he definido **son las de la velocidad del robot** en función de las distancia a la que se encuentran los objetos.

```
// velocidad
 RULE 1: IF s0 IS far THEN vel IS fast;

 RULE 1: IF s0 IS med THEN vel IS med;

 RULE 1: IF s0 IS near THEN vel IS slow;
```

Cuando el "S0" se acerca a los muros y obstáculos reduce la velocidad para poder girar a tiempo y cuanto no detecta nada va a la melocidad más alta. En estas reglas he ido probando distintas combinaciones pero la mejor que me ha funcionado es la mostrada. La siguiente regla hace que el robot de vueltas sobre si mismo a mitad de camino sin llegar a la salida.

```
// RULE 1: IF s0 IS far THEN vel IS med; ## no funciona ##
```

A continuación las **reglas para las rotaciones,** en este caso he asignado la rotación correspondiente de "sig" con cada variable de "rot", El resultado de las reglas es el siguiente:

```
// rotaciones
RULE 2: IF sig IS izqm THEN rot IS izqm;
RULE 2: IF sig IS izq THEN rot IS izq;

RULE 2: IF sig IS cen THEN rot IS cen;

RULE 2: IF sig IS der THEN rot IS der;
RULE 2: IF sig IS derm THEN rot IS derm;
```

Para terminar el último grupo de reglas son las usadas para **guiar al robot sin chocarse** con los objetos, para ello yo me he usado "S0", "S1", "S2, "S5" y "S8":

RULE 3: IF s1 IS near THEN rot IS der;
En el momento que tenga un objeto cerca del lado izquierdo del robot, este va a rotar a la derecha. Esto se puede dar en esquinas.

RULE 3: IF s8 IS far and s1 IS far THEN rot IS cen;
Cuando los sensores de los lados no detectan ningún objeto el robot va a "girar" al centro.

RULE 3: IF s2 IS near THEN rot IS derm;
Como en algunos casos el robot apuraba demasiado en algunas esquinas y se chocaba decidí usar un sensor más del lado izquierdo para que cuando gire lo haga teniendo en cuenta que aun tiene un objeto al lado.

RULE 3: IF s0 is far and s1 is far and s8 is near and s5 is near THEN rot is derm;
La última regla, si esta el robot siempre va recto y choca contra la pared

Algunas de las reglas que he ido usando pero no funcionan son las siguientes:

// R3: IF s8 IS far and s1 IS far and s0 IS far THEN rot IS cen; ## no funciona
// R3: IF s0 is far and s1 is far and s8 is near and s5 is near THEN rot is derm;## no funciona
// R3: IF s0 IS near and s1 IS med and s8 IS med and s5 is far THEN rot IS derm; ## no funcina
// R3: IF s8 IS near THEN rot IS izqm; ## no funciona
// RULE 1: IF s0 IS far THEN vel IS med; ## no funciona ##