

Práctica de Sistemas de Búsqueda

Enunciado	2
Especificación formal del problema	3
Definición del espacio de estados	3
Estado inicial	5
Estado final	6
Conjunto de reglas de producción	7
Implementación del problema con una estrategia de backtracking	11
Versión backtrack original	15
Versión backtrack iterativa	22
Implementación de problema utilizando un algoritmo A*	25
Definición de la función heurística	27
Obtención de la solución y análisis de los resultados	28

Especificación formal del problema

Realizar la especificación formal del problema. Generar un documento pdf con la definición.

Como se puede observar en el enunciado tenemos que desarrollar el sistema de producción de un robot para encontrar la secuencia de posiciones y movimientos que permitan al robot salir del laberinto que se muestra en la imagen.

Este robot cuenta con sensores de posición por lo que en todo momento el robot sabe en qué posición está y en qué orientación está dispuesto. El robot como dice el enunciado cuenta con dos tipos de operaciones, una de movimiento en la que el robot se desplaza una casilla en la dirección a la que esté orientado, y operaciones de giro de 90 grados tanto a la izquierda como a la derecha, manteniéndose el robot en la misma casilla.

Una vez tenemos claro cual es el entorno del problema que tenemos que resolver, lo primero que vamos a hacer es realizar una especificación formal del problema. Para ello, tenemos que definir los 4 componentes básicos que tienen todas las especificaciones formales de problemas. Estos componentes son el espacio de estados, el estado inicial, el estado final y el conjunto de reglas de producción.

Definición del espacio de estados

Como se ha dicho en el enunciado, nuestro robot tiene un encoder que le permite realizar cálculos de odometría. Por lo tanto, nuestro robot tendrá información sobre en qué posición se encuentra dentro del laberinto además de la orientación que tiene el robot con respecto a la entrada del laberinto.

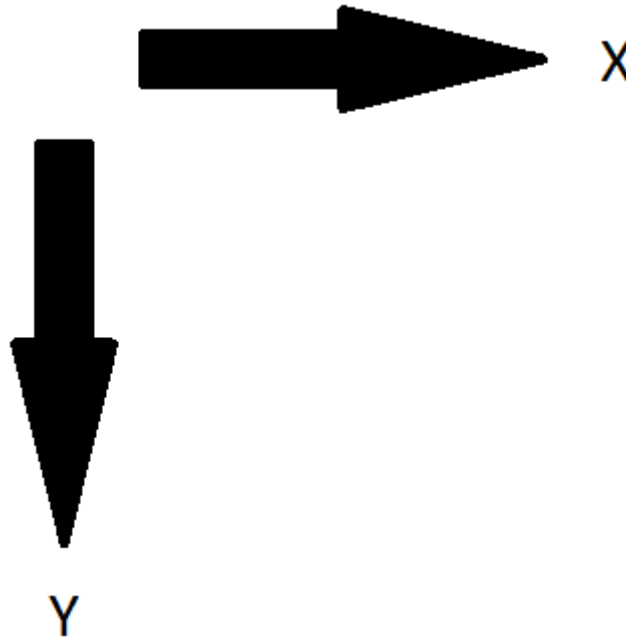
Es por eso por lo que nuestro espacio de estados va a estar formado por:

- Un valor llamado x el cual hace referencia a la coordenada horizontal de la casilla en la que se encuentra el robot con respecto a la esquina superior izquierda que es el eje de coordenadas.
- Un valor llamado y el cual hace referencia a la coordenada vertical de la casilla en la que se encuentra el robot con respecto a la esquina superior izquierda que es el eje de coordenadas.
- Un valor llamado θ el cual hace referencia a la orientación del robot con respecto a la entrada del laberinto, es decir, se considerará que la orientación del robot en la entrada del laberinto será de 0 y partir de este valor de referencia, se obtiene la orientación relativa del robot.

De esta manera, nuestro espacio de estados va a ser el siguiente:

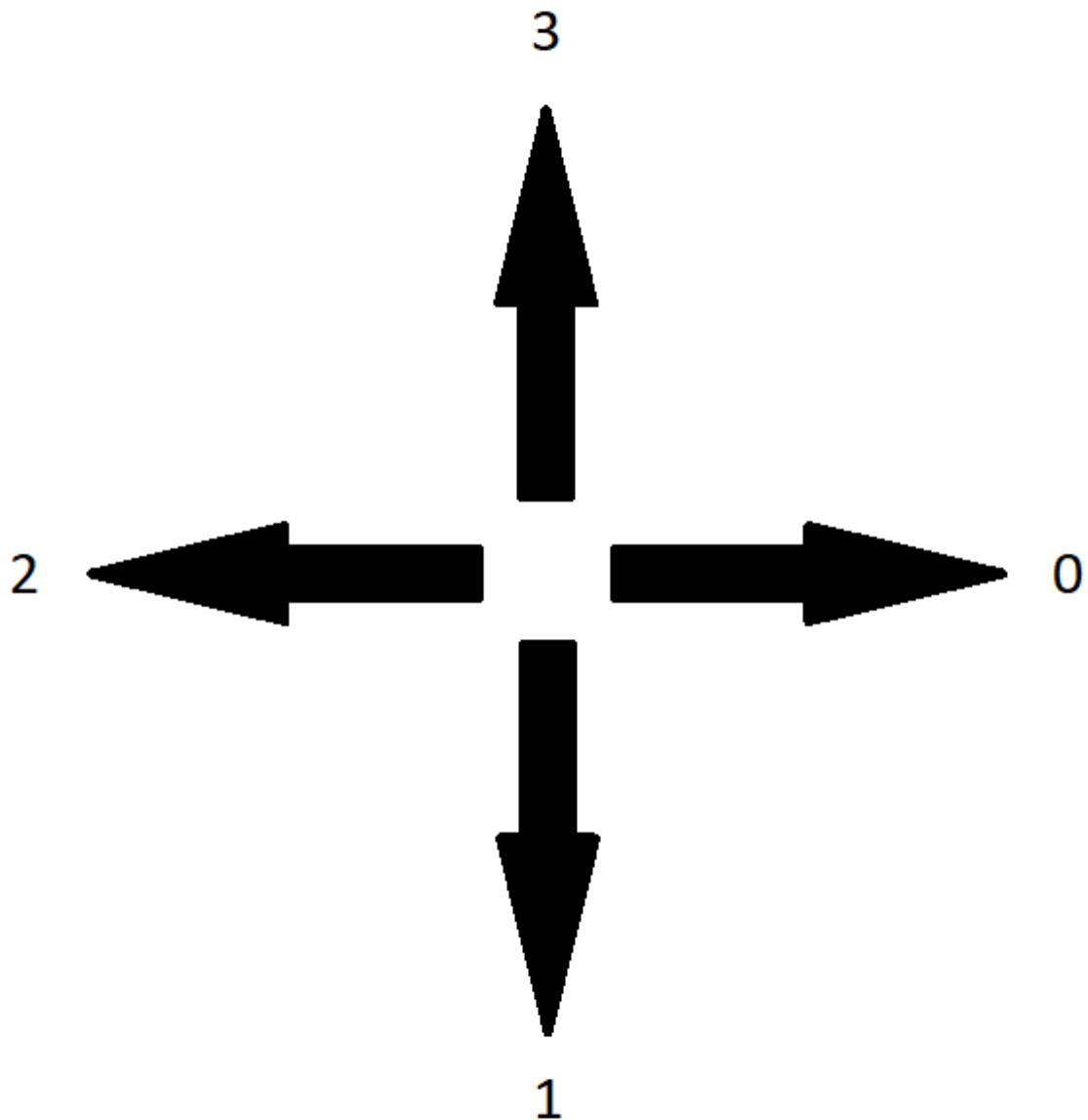
(x , y , θ)

Ojo: En este problema, el valor de la coordenada y aumenta a medida que se va bajando en este eje vertical y para que así todas las coordenadas de las casillas del laberinto sean positivas.



Antes de continuar con la especificación formal del problema quería comentar un aspecto muy a tener en cuenta sobre cómo hemos definido la orientación del robot en el problema. Como se indica en el enunciado, el robot puede realizar giros de 90 grados tanto a la derecha como a la izquierda. De esta manera, tenemos 4 valores discretos posibles para la orientación del robot. En vez de expresar estos valores de orientación en radianes (0 , $\pi/2$, π , $3\pi/2$) vamos a expresar esta orientación con números del 0 al 3 (0, 1, 2, 3) ya que esto nos facilita expresar más adelante la rotación del robot en PROLOG.

De esta manera, para cada tipo de orientación del robot se le va asignar un número del 0 al 3 siguiendo el siguiente esquema:

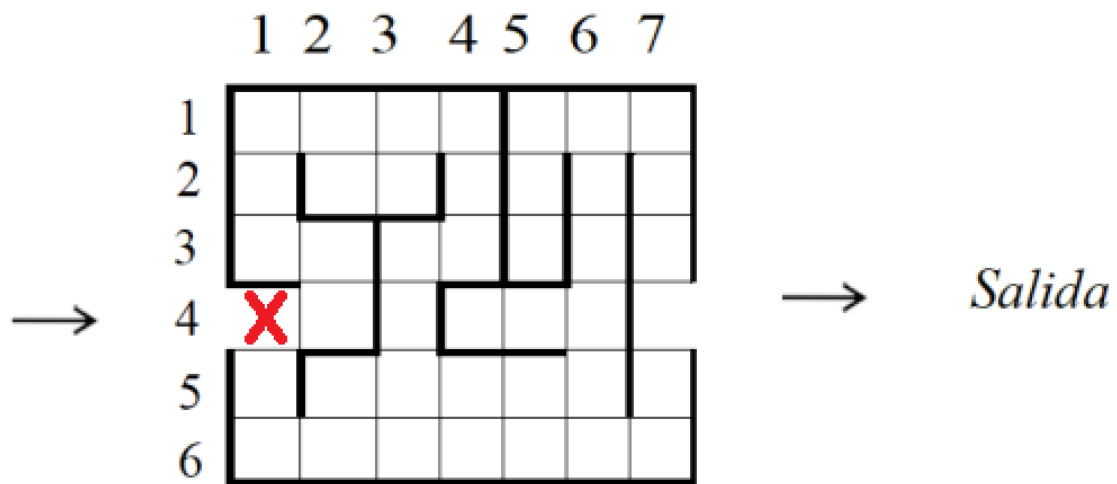


Estado inicial

Una vez hemos definido cuál es el espacio de estados de nuestro problema, el siguiente paso es definir el estado inicial del problema. Este estado inicial nos va a marcar la localización y la rotación de nuestro robot en el inicio del problema.

En este problema podemos observar que la casilla de entrada se encuentra en las coordenadas $x = 1$ e $y = 4$, mientras que la orientación del robot será de 0 ya que como hemos comentado anteriormente, el valor de la orientación θ se tomaba a partir de la orientación inicial del robot que dijimos que sería de 0.

De esta manera tenemos que el estado inicial del robot será el estado **(1,4,0)**.

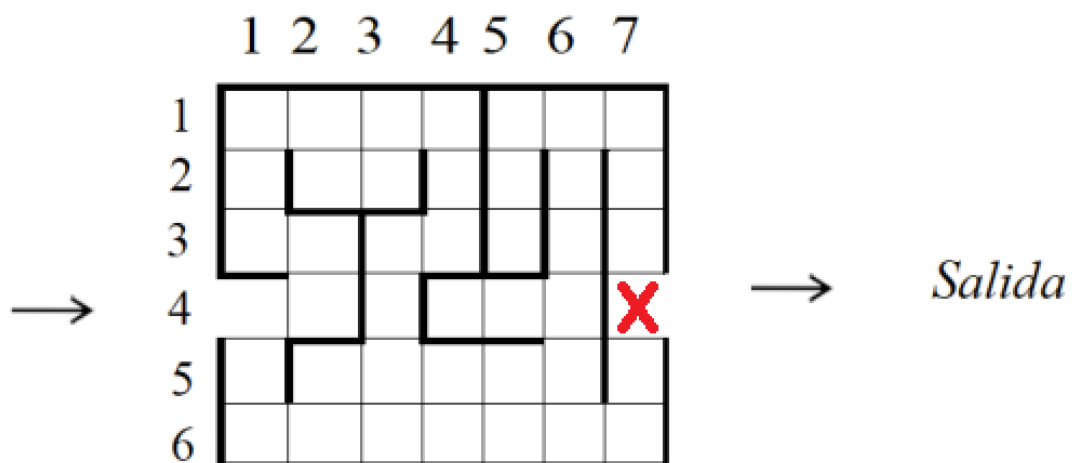


Estado final

Una vez hemos definido cuál es el espacio de estados de nuestro problema y el estado inicial del mismo, el siguiente paso es definir el estado final del problema. Este estado final nos va a marcar la localización y la rotación que debe tener nuestro robot para conseguir solucionar el problema.

En este problema podemos observar que la casilla de salida se encuentra en las coordenadas $x = 7$ e $y = 4$, mientras que la orientación del robot será de 0 ya que el robot en esta casilla de salida necesita tener la misma orientación que en el estado inicial para poder salir del laberinto.

De esta manera tenemos que el estado inicial del robot será el estado **(7,4,0)**.



Conjunto de reglas de producción

Una vez hemos definido cuál es el espacio de estados de nuestro problema y el estado inicial y final del mismo, el siguiente y último paso en la especificación formal del problema es definir el conjunto de reglas de producción. Este conjunto de reglas de producción operan sobre la base de datos global. Esta base de datos global describe de forma única el estado del sistema objetivo del problema a través del espacio de estados.

Estás reglas de producción van a tener una serie de precondiciones que se deben cumplir para que la aplicación de las reglas sea efectiva. En caso de que una de las precondiciones de una regla de producción no se cumpla, automáticamente la regla no podrá ser ejecutada. En caso de que se cumplan todas las precondiciones de una regla, se producirá la aplicación de la misma y esto supondrá una modificación en la base de datos global.

En el enunciado del problema, se ha indicado que el robot puede realizar operaciones de avance y de giro tanto a la izquierda como a la derecha. De esta manera cabría pensar que nuestro conjunto de reglas de producción estará formado por 3 reglas, una que permita avanzar al robot, otra que permita al robot girar a la derecha 90 grados y otra que permita al robot girar a la izquierda 90 grados.

Sin embargo, la regla de avanzar debe descomponerse en varias reglas de avanzar ya que esta operación de avanzar depende de la orientación del robot. Básicamente, las modificaciones en la base de datos global al realizar la operación de avance son distintas en función de la orientación del robot. De esta manera, vamos a tener que descomponer la regla de avanzar en 4 reglas de avance, una para cada orientación posible del robot.

Una vez hemos identificado las 6 reglas de producción que va a contener nuestra especificación formal del problema vamos a definir de forma más detallada cada una de estas reglas de producción indicando las precondiciones y las modificaciones sobre la base de datos una vez se cumplan las precondiciones.

Regla de avance hacia delante (Orientación 0)

En esta primera regla, vamos a definir el avance del robot cuando se encuentra en la orientación 0. Cuando el robot se encuentra en la **orientación 0**, vamos a considerar que avanza hacia **delante**. Para que esta regla pueda aplicarse, deben cumplirse 2 precondiciones:

- **El valor de la coordenada x debe ser menor que 7:** Con esta precondición nos aseguramos que el robot no se salga del laberinto y se choque con una de las paredes que delimita el laberinto, ya que el laberinto abarca las casillas con coordenadas x de la 1 a la 7.
- **No debe existir una pared vertical delante:** Con esta precondición nos aseguramos que el robot no se choque con una de las paredes interiores del laberinto con orientación vertical.

Si se cumplen estas 2 precondiciones, esta regla de avance modificará la coordenada x de la posición del robot ya que esta se incrementará en una unidad al avanzar a la casilla siguiente con coordenada $x + 1$ (manteniéndose θ e y constantes).

Regla de avance hacia la derecha (Orientación 1)

En esta segunda regla, vamos a definir el avance del robot cuando se encuentra en la orientación 1. Cuando el robot se encuentra en la **orientación 1**, vamos a considerar que avanza hacia la **derecha**. Para que esta regla pueda aplicarse, deben cumplirse 2 precondiciones:

- **El valor de la coordenada y debe ser menor que 6:** Con esta precondición nos aseguramos que el robot no se salga del laberinto y se choque con una de las paredes que delimita el laberinto, ya que el laberinto abarca las casillas con coordenadas y de la 1 a la 6.
- **No debe existir una pared horizontal delante:** Con esta precondición nos aseguramos que el robot no se choque con una de las paredes interiores del laberinto con orientación horizontal.

Si se cumplen estas 2 precondiciones, esta regla de avance modificará la coordenada y de la posición del robot ya que esta se incrementará en una unidad al avanzar a la casilla siguiente con coordenada $y + 1$ (manteniéndose x y θ constantes).

Regla de avance hacia atrás (Orientación 2)

En esta tercera regla, vamos a definir el avance del robot cuando se encuentra en la orientación 2. Cuando el robot se encuentra en la **orientación 2**, vamos a considerar que avanza hacia **atrás**. Para que esta regla pueda aplicarse, deben cumplirse 2 precondiciones:

- **El valor de la coordenada x debe ser mayor que 1:** Con esta precondición nos aseguramos que el robot no se salga del laberinto y se choque con una de las paredes que delimita el laberinto, ya que el laberinto abarca las casillas con coordenadas x de la 1 a la 7.
- **No debe existir una pared vertical delante:** Con esta precondición nos aseguramos que el robot no se choque con una de las paredes interiores del laberinto con orientación vertical.

Si se cumplen estas 2 precondiciones, esta regla de avance modificará la coordenada x de la posición del robot ya que esta se decrementará en una unidad al avanzar a la casilla siguiente con coordenada $x - 1$ (manteniéndose θ e y constantes).

Regla de avance hacia la izquierda (Orientación 3)

En esta cuarta regla, vamos a definir el avance del robot cuando se encuentra en la orientación 3. Cuando el robot se encuentra en la **orientación 3**, vamos a considerar que

avanza hacia la **izquierda**. Para que esta regla pueda aplicarse, deben cumplirse 2 precondiciones:

- **El valor de la coordenada y debe ser mayor que 1:** Con esta precondición nos aseguramos que el robot no se salga del laberinto y se choque con una de las paredes que delimita el laberinto, ya que el laberinto abarca las casillas con coordenadas y de la 1 a la 6.
- **No debe existir una pared horizontal delante:** Con esta precondición nos aseguramos que el robot no se choque con una de las paredes interiores del laberinto con orientación horizontal.

Si se cumplen estas 2 precondiciones, esta regla de avance modificará la coordenada y de la posición del robot ya que esta se decrementará en una unidad al avanzar a la casilla siguiente con coordenada y - 1 (manteniéndose x y θ constantes).

Regla de giro a la derecha

En esta quinta regla, vamos a definir el giro de robot 90 grados hacia la derecha. Para que esta regla pueda aplicarse no se debe cumplir ningún tipo de precondición, pues el robot realiza esta giro a la derecha en la misma posición por lo que no hay posibilidad de que se choque con una pared.

La acción de esta regla es incrementar el valor de la orientación en una unidad siguiendo el esquema de orientación explicado en el espacio de estados. Sin embargo, al disponer de 4 valores discretos (del 0 al 3) para especificar la orientación del robot, no siempre hay que incrementar en una unidad el valor de la orientación del robot.

En el caso de que nuestro robot esté en la orientación 3 y queramos aplicar la regla girar a la derecha, en vez de incrementar en una unidad el valor de esta orientación, daremos el valor de 0 para así cerrar el círculo.

De esta manera, vamos a utilizar la siguiente función matemática para definir el valor de la orientación cuando se aplica la regla de girar a la derecha:

$$f(x) = \begin{cases} \theta = \theta + 1 & : 0 \leq \theta \leq 2 \\ 0 & : \theta = 3 \end{cases}$$

Regla de giro a la izquierda

En esta sexta regla, vamos a definir el giro de robot 90 grados hacia la izquierda. Para que esta regla pueda aplicarse no se debe cumplir ningún tipo de precondición, pues el robot realiza esta giro a la izquierda en la misma posición por lo que no hay posibilidad de que se choque con una pared.

La acción de esta regla es decrementar el valor de la orientación en una unidad siguiendo el esquema de orientación explicado en el espacio de estados. Sin embargo, al disponer de 4 valores discretos (del 0 al 3) para especificar la orientación del robot, no siempre hay que decrementar en una unidad el valor de la orientación del robot.

En el caso de que nuestro robot esté en la orientación 0 y queramos aplicar la regla girar a la izquierda, en vez de decrementar en una unidad el valor de esta orientación, daremos el valor de 3 para así cerrar el círculo.

De esta manera, vamos a utilizar la siguiente función matemática para definir el valor de la orientación cuando se aplica la regla de girar a la izquierda:

$$f(x) = \begin{cases} \theta = \theta - 1 & : 1 \leq \theta \leq 3 \\ 3 & : \theta = 0 \end{cases}$$

Con esto ya tendremos la especificación formal del problema perfectamente realizada. Ahora el siguiente paso es implementar en PROLOG un programa el cual sea capaz de resolver este problema que hemos especificado utilizando el algoritmo de backtrack.

Implementación del problema con una estrategia de backtracking

Implementar el problema con una estrategia de backtracking utilizando el lenguaje Prolog. Se deben plantear dos versiones: una básica y otra que incremente de forma iterativa el límite de profundidad. ¿Qué diferencias se observan en el comportamiento del algoritmo y en la calidad de la solución obtenida?

Ahora, a partir de la especificación formal del problema que hemos realizado, vamos a implementar la resolución de este problema en PROLOG utilizando la estrategia de backtracking. Para realizar esta implementación, vamos a partir de la implementación del problema del Pastor, el Lobo, la Oveja y la Berza que vimos en clase, ya que hay aspectos sobre el uso del backtracking que son comunes para todos los problemas.

Para adaptar esta implementación del problema del Pastor, vamos a redefinir el concepto de estado, ya que en nuestro problema, nuestro espacio de estados está formado por los componentes x e y de la posición del robot y su orientación siendo estos 3 valores enteros.

Después definiremos en la sección de cláusulas las reglas de producción que hemos definido en la especificación formal del problema con sus respectivas precondiciones y acciones sobre la base de datos global.

Obsérvese cómo para definir las paredes interiores del laberinto, se ha utilizado una regla individual para definir cada pared, pues no siguen ningún tipo de patrón numérico y no hay otra manera de especificar la existencia de estas paredes.

En las siguientes páginas se adjunta, el contenido del código utilizado para la implementación de este problema aunque también se incluirá este código fuente en la carpeta de la entrega, este código fuente está incluido dentro del proyecto de Prolog del pastor (Practica Sistemas de Búsqueda.pro):

```
/******
```

Copyright (c) My Company

Project: PASTOR

FileName: PASTOR.PRO

Purpose: No description

Written by: Visual Prolog

Comments:

```
*****/
```

```
include "pastor.inc"
```

```
domains
```

```
    posicion=integer
```

```
angulo=integer
est=estado(posicion,posicion,angulo)
lista=est*
limite=integer
```

predicates

```
mueve(est,est)
hayParedVertical(posicion,posicion,posicion,posicion)
hayParedHorizontal(posicion,posicion,posicion,posicion)

miembro(est,lista)
resuelve(lista,est,limite,limite) /* Lista de estados, Estado destino */
escribe(lista)
mejorsol(integer)
```

clauses

```
mueve(estado(I,Y,0),estado(F,Y,0)):-
    F = I + 1,
    I < 7,
    not(hayParedVertical(I,Y,F,Y)),
    not(hayParedVertical(F,Y,I,Y)).
/*write("Ad \n"),
write('(', F, '-', Y, '-', 0, ');\n').*/
```

```
mueve(estado(X,I,1),estado(X,F,1)):-
    F = I + 1,
    I < 6,
    not(hayParedHorizontal(X,I,X,F)),
    not(hayParedHorizontal(X,F,X,I)).
/*write("Der \n"),
write('(', X, '-', F, '-', 1, ');\n').*/
```

```
mueve(estado(I,Y,2),estado(F,Y,2)):-
    F = I - 1,
    I > 1,
    not(hayParedVertical(I,Y,F,Y)),
    not(hayParedVertical(F,Y,I,Y)).
/*write("At \n"),
write('(', F, '-', Y, '-', 2, ');\n').*/
```

```
mueve(estado(X,I,3),estado(X,F,3)):-
    F = I - 1,
    I > 1,
```

```
not(hayParedHorizontal(X,I,X,F)),
not(hayParedHorizontal(X,F,X,I)).
/*write("Izq \n"),
write('(', X, '-', F, '-', 3, ');\n').*/
```

```
mueve(estado(X,Y,AI),estado(X,Y,AF)):-
    AI < 3,
    AF = AI + 1.
/*write("Giramos a la derecha\n"),
write('(', X, '-', Y, '-', AF, ');\n').*/
```

```
mueve(estado(X,Y,3),estado(X,Y,AF)):-
    AF = 0.
/*write("Giramos a la derecha\n"),
write('(', X, '-', Y, '-', AF, ');\n').*/
```

```
mueve(estado(X,Y,AI),estado(X,Y,AF)):-
    AI > 0,
    AF = AI - 1.
/*write("Giramos a la izquierda\n"),
write('(', X, '-', Y, '-', AF, ');\n').*/
```

```
mueve(estado(X,Y,0),estado(X,Y,AF)):-
    AF = 3.
/*write("Giramos a la izquierda\n"),
write('(', X, '-', Y, '-', AF, ');\n').*/
```

```
hayParedVertical(1,2,2,2).
hayParedVertical(1,5,2,5).
hayParedVertical(2,3,3,3).
hayParedVertical(2,4,3,4).
hayParedVertical(3,2,4,2).
hayParedVertical(3,4,4,4).
hayParedVertical(4,1,5,1).
hayParedVertical(4,2,5,2).
hayParedVertical(4,3,5,3).
hayParedVertical(5,2,6,2).
hayParedVertical(5,3,6,3).
hayParedVertical(6,2,7,2).
hayParedVertical(6,3,7,3).
hayParedVertical(6,4,7,4).
hayParedVertical(6,5,7,5).
```

```
hayParedHorizontal(1,4,1,3).
hayParedHorizontal(2,5,2,4).
```

```
hayParedHorizontal(2,3,2,2).
hayParedHorizontal(3,3,3,2).
hayParedHorizontal(4,5,4,4).
hayParedHorizontal(4,4,4,3).
hayParedHorizontal(5,5,5,4).
hayParedHorizontal(5,4,5,3).
```

```
/*Estados repetidos */
```

```
miembro(E,[E|_]).
miembro(E,[_|T]):-
    miembro(E,T).
```

```
/*Resolucion de algoritmo */
```

```
resuelve(Lista,Destino,_,_):-
    Lista=[H|T],
    Destino=H,
    escribe(Lista).
```

```
resuelve(Lista,Destino,Lim_ant,Limite):-
    Lista=[H|T],
    not(miembro(H,T)),
    mueve(H,Hfinal),
    Nlista=[Hfinal|Lista],
    Nue_Lim=Lim_ant+1,
    Nue_Lim<=Limite,
    resuelve(Nlista,Destino,Nue_Lim,Limite).
```

```
/*Escritura de la lista */
```

```
escribe([]).
escribe([H|T]):-
    escribe(T),
    write(H,'\n').
```

```
mejorsol(Lim_ini):-
    resuelve([estado(1,4,0)],estado(7,4,0),1,Lim_ini).
```

```
mejorsol(Lim_ini):-
    Nue_lim=Lim_ini+1,
    mejoresol(Nue_lim).
```

goal

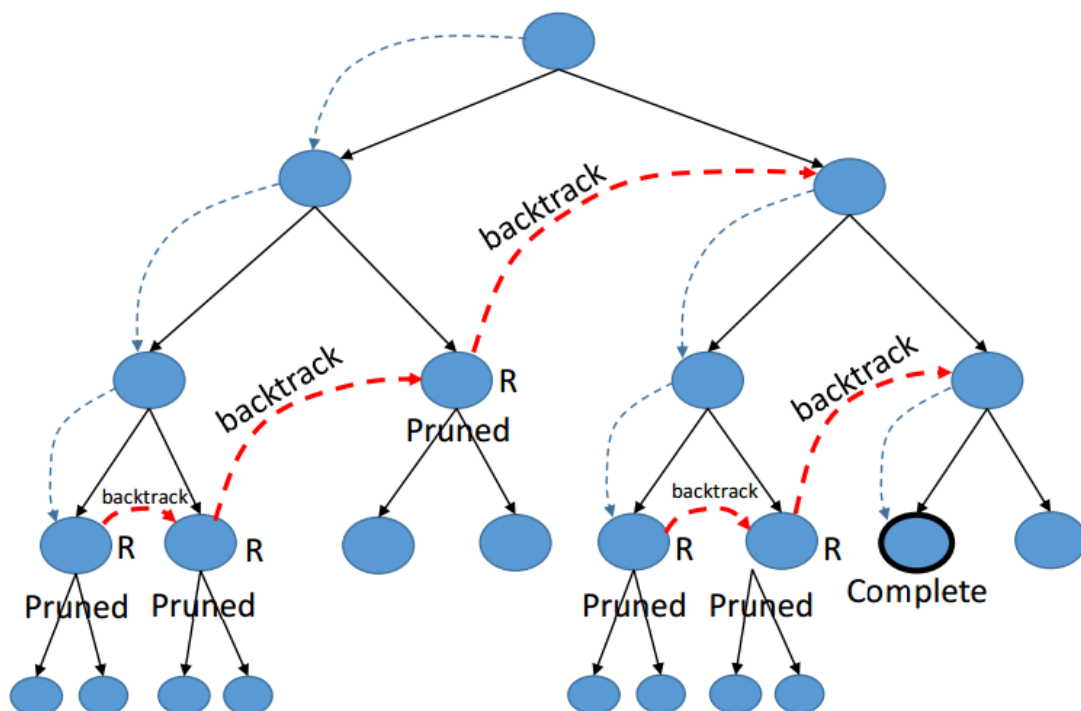
```
resuelve([estado(1,4,0)],estado(7,4,0),1,50).
/*mejorsol(1).*/
```

En el enunciado se nos pide implementar 2 versiones para la resolución de este problema. En la primera versión se utilizará la estrategia de backtrack original utilizando un límite estático definido en tiempo de compilación mientras que en la segunda versión se utilizará una estrategia de backtrack que vaya incrementando de manera dinámica el valor de este límite.

Versión backtrack original

En primer lugar, vamos a implementar la solución a este problema utilizando el algoritmo de backtrack original. En este algoritmo de backtrack original, se van a ir explorando distintos conjuntos de reglas aplicables hasta llegar a un estado repetido o llegar a una estado sin salida en la que no tengamos más reglas aplicables. Además, en esta estrategia vamos a incluir un límite máximo al conjunto de reglas aplicables necesarias para alcanzar la solución ya que no nos interesan las soluciones que requieran aplicar un conjunto de reglas excesivamente grandes.

De esta manera, con esta estrategia original de backtracking vamos a realizar una búsqueda por profundidad ya que vamos a explorar en profundidad una parte del árbol de búsqueda. La búsqueda en profundidad se caracteriza por ser rápida ya que el objetivo es encontrar una solución en el menor tiempo aunque la calidad de la solución obtenida no sea óptima.



Una vez hemos explicado las bases del funcionamiento de esta versión original del backtracking, vamos a buscar una solución para el laberinto utilizando esta versión original o básica del backtracking utilizando la implantación realizada en PROLOG. Para esta implementación de la solución vamos a utilizar un límite estático de 50 para indicar que no

deseamos buscar en el árbol de búsqueda a una profundidad mayor de 50 y la solución obtenida cuente con menos de 50 (o 50) reglas aplicables.

Tras ejecutar este algoritmo de backtracking en su versión básica con un límite de 50 reglas, obtenemos el siguiente resultado:

```
[Inactive C:\Users\Raul\Desktop\4º\Fundamentos de Sistemas Inteligentes\Ejemplos clase\p]
estado[1,4,0]
estado[2,4,0]
estado[2,4,1]
estado[2,4,2]
estado[1,4,2]
estado[1,4,1]
estado[1,5,1]
estado[1,6,1]
estado[1,6,2]
estado[1,6,3]
estado[1,6,0]
estado[2,6,0]
estado[3,6,0]
estado[4,6,0]
estado[5,6,0]
estado[6,6,0]
estado[7,6,0]
estado[7,6,1]
estado[7,6,2]
estado[6,6,2]
estado[5,6,2]
estado[4,6,2]
estado[3,6,2]
estado[2,6,2]
estado[2,6,3]
estado[2,5,3]
estado[2,5,0]
estado[3,5,0]
estado[4,5,0]
estado[5,5,0]
estado[6,5,0]
estado[6,5,1]
estado[6,5,2]
estado[6,5,3]
estado[6,4,3]
estado[6,3,3]
estado[6,2,3]
estado[6,1,3]
estado[6,1,0]
estado[7,1,0]
estado[7,1,1]
estado[7,2,1]
estado[7,3,1]
estado[7,4,1]
estado[7,5,1]
estado[7,5,2]
estado[7,5,3]
estado[7,4,3]
estado[7,4,0]
yes
```

Como se puede observar, la calidad de la solución es bastante “pobre” ya que se ha necesitado aplicar un gran número de reglas para llegar a la solución.

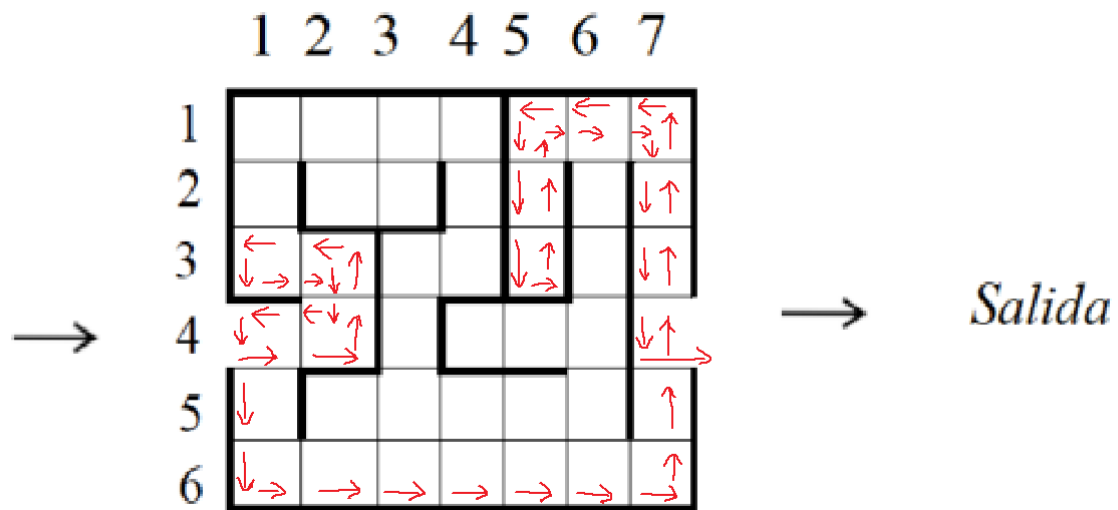
De manera gráfica, el recorrido el robot hasta alcanzar el estado final es el siguiente:

*Salida*

Ahora vamos a probar a modificar el orden en el que están colocadas las reglas de producción de manera que la regla de girar a la izquierda aparezca antes que la regla de girar a la derecha y por tanto se prioriza la ejecución del giro a la izquierda sobre el giro a la derecha.

El resultado de la ejecución en este caso es el siguiente:

estado[1,4,0]
estado[2,4,0]
estado[2,4,3]
estado[2,3,3]
estado[2,3,2]
estado[1,3,2]
estado[1,3,1]
estado[1,3,0]
estado[2,3,0]
estado[2,3,1]
estado[2,4,1]
estado[2,4,2]
estado[1,4,2]
estado[1,4,1]
estado[1,5,1]
estado[1,6,1]
estado[1,6,0]
estado[2,6,0]
estado[3,6,0]
estado[4,6,0]
estado[5,6,0]
estado[6,6,0]
estado[7,6,0]
estado[7,6,3]
estado[7,5,3]
estado[7,4,3]
estado[7,3,3]
estado[7,2,3]
estado[7,1,3]
estado[7,1,2]
estado[6,1,2]
estado[5,1,2]
estado[5,1,1]
estado[5,2,1]
estado[5,3,1]
estado[5,3,0]
estado[5,3,3]
estado[5,2,3]
estado[5,1,3]
estado[5,1,0]
estado[6,1,0]
estado[7,1,0]
estado[7,1,1]
estado[7,2,1]
estado[7,3,1]
estado[7,4,1]
estado[7,4,0]
yes



Como se puede observar en la imagen, ahora al priorizar el giro a la izquierda sobre el giro a la derecha, el camino tomado para encontrar la solución es totalmente diferente. Obsérvese como hay varios giros a la derecha pero estos solo se producen cuando las reglas de avance y de giro a la izquierda ya se han aplicado y no quedan más reglas aplicables para esos estados.

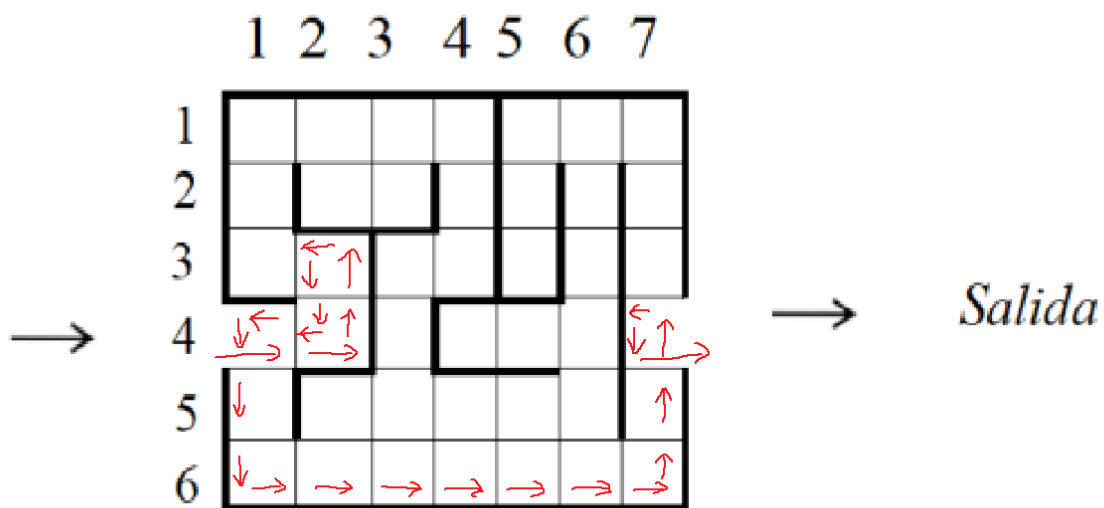
Ahora, vamos a mantener esta prioridad de la regla de giro a la izquierda sobre la regla de giro a la derecha y vamos a reducir a la mitad el valor del límite estático con el objetivo de realizar una búsqueda de una solución que contenga 25 o menos reglas.

El resultado es el siguiente:

```

estado[1,4,0]
estado[2,4,0]
estado[2,4,3]
estado[2,3,3]
estado[2,3,2]
estado[2,3,1]
estado[2,4,1]
estado[2,4,2]
estado[1,4,2]
estado[1,4,1]
estado[1,5,1]
estado[1,6,1]
estado[1,6,0]
estado[2,6,0]
estado[3,6,0]
estado[4,6,0]
estado[5,6,0]
estado[6,6,0]
estado[7,6,0]
estado[7,6,3]
estado[7,5,3]
estado[7,4,3]
estado[7,4,2]
estado[7,4,1]
estado[7,4,0]
yes


```



Como se puede observar en la imagen, se sigue priorizando el giro a la izquierda sobre el giro a la derecha, pero al imponer un límite estático más bajo la solución es bastante mejor que las obtenidas en los casos anteriores. No es la mejor solución pero está bastante mejor que las encontradas hasta ahora. En este caso la solución cuenta con 25 reglas.

Por último, vamos a cambiar de nuevo el orden de las reglas para que la regla de giro a la derecha vuelva a tener prioridad sobre la regla de giro a la izquierda y vamos a mantener este valor del límite estático a 25 con el objetivo de realizar una búsqueda de una solución que contenga 25 o menos reglas.

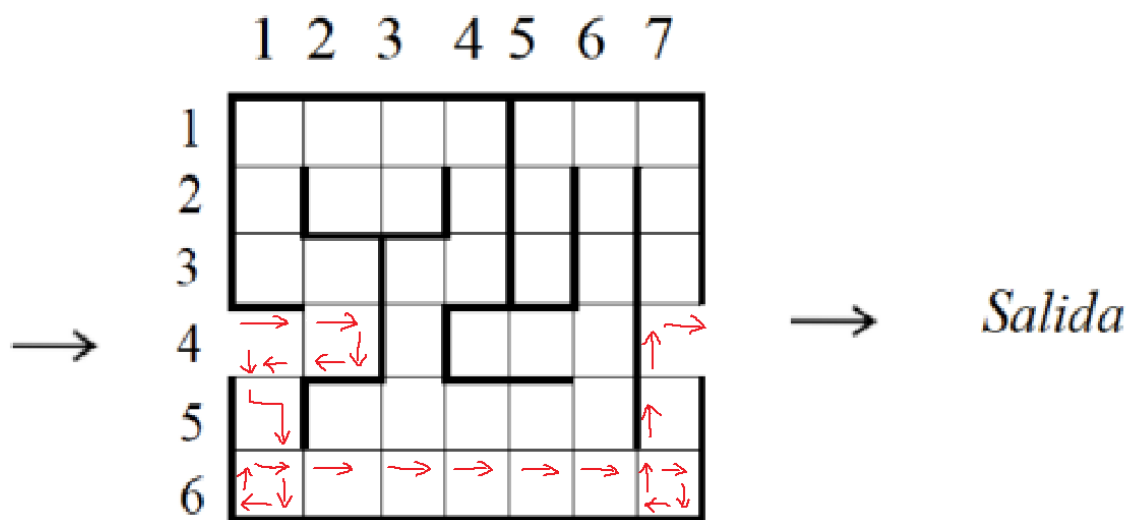
El resultado es el siguiente:

 [Inactive C:\Users\Raul\Desktop\4º\Fundamentos de Sistemas Inteligentes\Ejemplos clase\p]

```

estado[1,4,0]
estado[2,4,0]
estado[2,4,1]
estado[2,4,2]
estado[1,4,2]
estado[1,4,1]
estado[1,5,1]
estado[1,6,1]
estado[1,6,2]
estado[1,6,3]
estado[1,6,0]
estado[2,6,0]
estado[3,6,0]
estado[4,6,0]
estado[5,6,0]
estado[6,6,0]
estado[7,6,0]
estado[7,6,1]
estado[7,6,2]
estado[7,6,3]
estado[7,5,3]
estado[7,4,3]
estado[7,4,0]
yes

```



Como se puede observar en la imagen, ahora se vuelve a priorizar el giro a la derecha sobre el giro a la izquierda, pero al imponer un límite estático más bajo la solución es bastante mejor que la obtenidas con límite 50. No es la mejor solución pero está bastante mejor que las encontradas hasta ahora con límite 50. En este caso la solución cuenta con 23 reglas (frente a las 25 obtenidas priorizando el giro a la izquierda sobre el giro a la derecha).

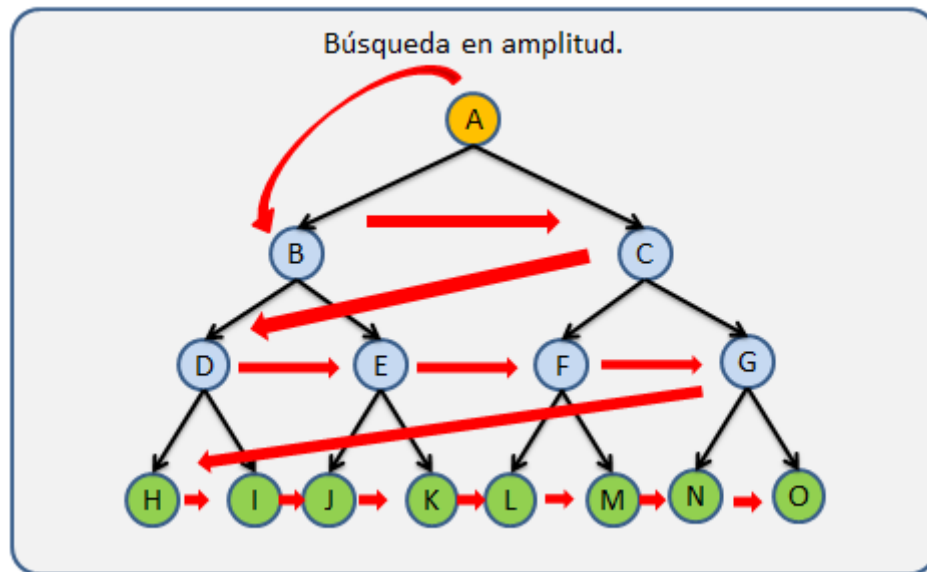
Con esto hemos comprobado experimentalmente como la solución obtenida utilizando el algoritmo de backtrack básico se obtiene de manera rápida pero no nos garantiza en absoluto que sea la solución óptima, ya que hay muchas ramas del árbol de búsqueda que no se han explorado y que pueden tener una mejor solución que la obtenida.

También se puede observar que la calidad de esta solución mejora a medida que se va disminuyendo el límite estático introducido. Es muy importante elegir un valor adecuado para este límite ya que si introducimos un valor demasiado bajo no podemos obtener ninguna solución. También hemos visto que el orden en el que se colocan las reglas aplicables tiene influencia en la solución obtenida ya que se exploran distintas ramas del árbol de exploración.

Versión backtrack iterativa

En segundo lugar, vamos a implementar la solución a este problema utilizando el algoritmo de backtrack con una profundidad iterativa. En este algoritmo de backtrack iterativo, se van a ir explorando todos los conjuntos o sucesiones de reglas aplicables posibles utilizando un límite a este conjunto de reglas el cual va a ir aumentando poco a poco de manera iterativa.

De esta manera, con esta estrategia iterativa de backtracking vamos a realizar una búsqueda por amplitud ya que vamos a explorar todos los nodos del árbol de búsqueda de una determinada profundidad y una vez se hayan explorado todos los nodos de una determinada profundidad, se explorarán todos los nodos del siguiente nivel de profundidad. La búsqueda en amplitud se caracteriza por ser lenta ya que el objetivo es explorar todas las posibles combinaciones de reglas a una determinada profundidad hasta encontrar la solución con la profundidad más baja (óptima).



Una vez hemos explicado las bases del funcionamiento de esta versión de límite iterativo del backtracking, vamos a buscar una solución para el laberinto utilizando esta versión iterativa del backtracking utilizando la implantación realizada en PROLOG. Obsérvese cómo empezamos explorando soluciones con profundidad 1 y a partir de aquí se va aumentando el valor de este límite hasta encontrar la solución (óptima).

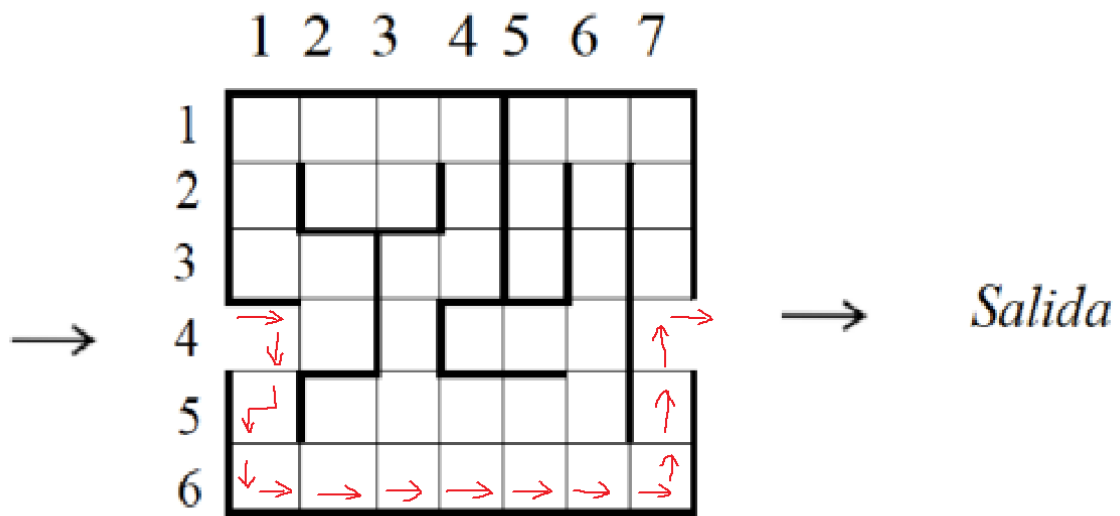
Tras ejecutar este algoritmo de backtracking en su versión iterativa, obtenemos el siguiente resultado:

[Inactive C:\Users\Raul\Desktop\4º\Fundamentos de Sistemas Inteligentes\Ejemplos clase\p]

```
estado[1,4,0]
estado[1,4,1]
estado[1,5,1]
estado[1,6,1]
estado[1,6,0]
estado[2,6,0]
estado[3,6,0]
estado[4,6,0]
estado[5,6,0]
estado[6,6,0]
estado[7,6,0]
estado[7,6,3]
estado[7,5,3]
estado[7,4,3]
estado[7,4,0]
yes
```

Como se puede observar, la calidad de la solución es bastante buena (es más, es la solución óptima) ya que se ha necesitado aplicar un pequeño número de reglas para llegar a la solución.

De manera gráfica el recorrido el robot hasta alcanzar el estado final es el siguiente:



En este caso, podemos observar como la solución encontrada para resolver el laberinto es la mejor solución ya que implica el menor número de movimientos. Es la solución que hubiéramos utilizado cualquier humano que disponga del mapa del laberinto para salir del laberinto.

En este caso, el orden en el que se dispongan las reglas aplicables es poco relevante, pues no van a tener ninguna influencia sobre el resultado final ya que no existen más soluciones de la misma longitud que la solución óptima. En todo caso tendría alguna leve influencia sobre el tiempo de cálculo de la solución pero como Prolog realiza la implementación del backtrack de manera tan rápida y eficiente esta diferencia de tiempos resultante del cambio en el orden de las reglas aplicables es inapreciable a simple vista.

Con esto hemos comprobado experimentalmente como la solución obtenida utilizando el algoritmo de backtrack iterativo se obtiene de manera bastante lenta pero en este caso nos garantiza que la solución encontrada sea la solución óptima, ya que se han explorado todas ramas del árbol de búsqueda a una determinada profundidad hasta encontrar la solución óptima.

Implementación de problema utilizando un algoritmo A*

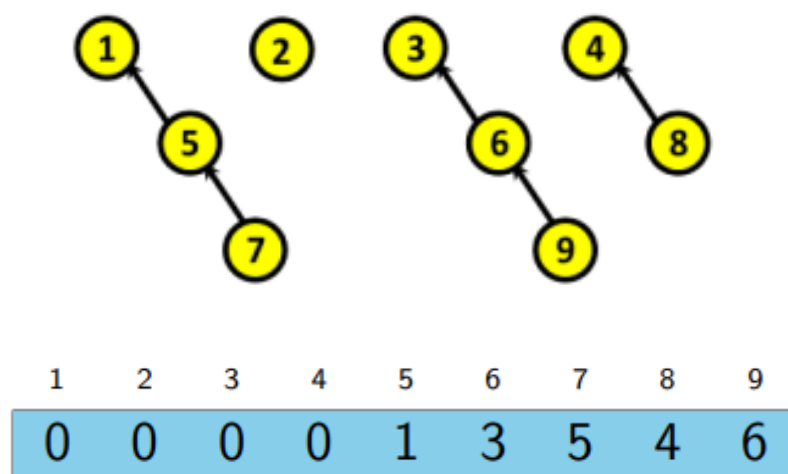
Proponer una heurística para desarrollar un algoritmo A* que resuelva el problema de resolución del laberinto propuesto. Discutir la calidad de las soluciones obtenidas, así como el tiempo de cálculo implicado (si es útil considerar el número de nodos explorados)

Una vez hemos obtenido la solución a este problema del laberinto a través de una implementación del problema con la estrategia de backtracking, ahora vamos a realizar la implantación de este problema utilizando la estrategia del algoritmo A*. A diferencia del caso anterior, ahora tenemos la libertad total de elegir el entorno y el lenguaje que más nos guste para implementar este algoritmo A* de la solución.

En nuestro caso, hemos decidido implementar este algoritmo A* que resuelva el problema en el lenguaje c, pues se ha utilizado mucho a lo largo de la carrera y los alumnos tenemos bastante experiencia en este lenguaje.

Por un lado, para la implementación de este algoritmo A* se necesita un conjunto de estados abiertos y un conjunto de estados cerrados. En nuestro caso, vamos a implementar este conjunto de estados abiertos y cerrados utilizando listas enlazadas. Por otro lado, para la implementación de este algoritmo A* se necesita un árbol de búsqueda cuyos nodos van a ser los distintos estados explorados. En nuestro caso, vamos a implementar este árbol de búsqueda a través de una representación de árboles utilizando una matriz como hemos visto en la asignatura de Estructuras de Datos y Algoritmos II donde cada nodo almacenará la información necesaria sobre el estado y sobre el apuntador correspondiente (entre otras cosas).

En la siguiente imagen se muestra un ejemplo de la lógica para representar este árbol de búsqueda:



Un algoritmo A es un algoritmo de exploración de grafos el cual utiliza una función heurística para ordenar el conjunto de estados abiertos. De esta manera, el algoritmo A es una estrategia de búsqueda informada ya que hace uso de una información o heurística para determinar el orden en el que se va a explorar el árbol de exploración.

Para determinar el orden de los estados abiertos, se va a utilizar una función f la cual va a depender de la profundidad del nodo g y del valor de esta heurística que hayamos definido h .

$$f(n) = g(n) + h(n)$$

Esta heurística hace referencia al coste estimado para alcanzar el nodo o estado final desde el nodo o estado actual.

Para que este algoritmo A se “convierta” en un algoritmo A* debe cumplirse la siguiente propiedad.

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

Básicamente, para que un algoritmo A sea un algoritmo A*, el valor de esta estimación de la distancia al nodo o estado final debe ser inferior o igual al verdadero valor de la distancia entre el nodo actual y el nodo final. Dicho de otra manera, la estimación sobre el número de reglas restantes para alcanzar el estado final, debe ser menor o igual que el valor real. De esta manera, si queremos que nuestro Algoritmo A pase a ser un Algoritmo A* tenemos que realizar una sobreestimación (estimación digamos demasiado optimista) sobre el coste de alcanzar el objetivo.

Esta condición que debe cumplir un algoritmo A para ser un algoritmo A* se conoce como característica de admisibilidad. Es por esto por lo que un algoritmo A* se dice que es admisible. Mientras que esta característica de admisibilidad se cumpla, el algoritmo A* nos garantiza encontrar la mejor solución.

En el momento en el que se realice esta estimación por debajo del valor óptimo, esta característica de admisibilidad se perderá y no podremos garantizar que la solución encontrada sea la mejor ya que realizamos la estimación por debajo del valor óptimo.

Definición de la función heurística

Ahora, vamos a definir una función heurística para nuestro problema en concreto teniendo en cuenta esta característica de admisibilidad que debe cumplir nuestra heurística para que nuestro algoritmo A se convierta en un algoritmo A* y así poder encontrar la solución óptima.

Como ya comentamos en el enunciado del problema, en cada regla de avance se avanza una única casilla en un única coordenada. De esta manera, podemos afirmar que para llegar al estado final desde el estado actual en el que nos encontremos, habiendo una distancia x entre estos 2 puntos, se necesitaría de al menos x reglas para alcanzar este estado final.

Como tenemos dos coordenadas espaciales x e y la distancia entre estos dos puntos se calcularía con la fórmula:

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

Sin embargo, esta distancia entre dos puntos tendría decimales y no nos interesa (por sencillez y comprensión) que la función heurística tenga decimales. Es por esto por lo que vamos a redefinir este concepto de distancia. Para ello, vamos a considerar la distancia entre estos dos puntos, la diferencia en valor absoluto de componente x del estado actual y el final más la diferencia en valor absoluto de componente y del estado actual y el final.

$$|(x_{final} - x_{actual})| + |(y_{final} - y_{actual})|$$

El resultado de esta “distancia” será mayor que la verdadera distancia entre el estado actual y final, pero esta función heurística sigue siendo admisible ya que los movimientos que realiza el robot sólo varían el valor de una de las dos componentes de la posición (x, y) en una unidad por lo que en el “peor” de los casos, el valor de esta distancia es igual al número real de reglas aplicadas para llegar al estado final desde el actual.

De esta manera, hemos conseguido definir una función heurística admisible ya que cumple con la característica de admisibilidad y sobreestima el esfuerzo para alcanzar el estado final (en el peor de los casos, si se diera, igualaría esa estimación al valor real). Además, hemos conseguido simplificar el cálculo de esta función heurística para obtener un valor entero para así hacer más sencillo el entendimiento de esta heurística.

Obtención de la solución y análisis de los resultados

Ahora, una vez hemos definido la heurística a utilizar para nuestra implementación del algoritmo A*, vamos a realizar dicha implementación del algoritmo A* para que pueda resolver el problema del laberinto. Para la implementación de este algoritmo A* partiremos del pseudocódigo necesario para implementar un algoritmo de exploración de grafos.

1	Crear un <i>grafo de exploración</i> <i>G</i> que consista exclusivamente en el nodo inicial <i>s</i> . Iniciar con <i>s</i> una lista llamada <i>ABIERTOS</i> .
2	Crear una lista llamada <i>CERRADOS</i> que inicialmente estará vacía.
3	CICLO: si <i>ABIERTOS</i> está vacía, salida con fallo .
4	Seleccionar el primer nodo de la lista <i>ABIERTOS</i> , suprimirlo de ella e incluirlo en <i>CERRADOS</i> . Llamar <i>n</i> a este nodo.
5	Si <i>n</i> es un nodo objetivo, salida con éxito , dando la solución obtenida construyendo un camino, por medio de los apuntadores, desde <i>n</i> hasta <i>s</i> en <i>G</i> . (Los apuntadores se establecen en el paso 7).
6	Expandir el nodo <i>n</i> , generando el conjunto <i>M</i> de sus sucesores que no sean a la vez ascendientes de <i>n</i> . Incorporar estos miembros de <i>M</i> , como sucesores de <i>n</i> , en <i>G</i> .
7	Establecer un apuntador a <i>n</i> desde aquellos miembros de <i>M</i> que no estaban ya incluidos en <i>ABIERTOS</i> o <i>CERRADOS</i> . Añadir estos miembros de <i>M</i> a <i>ABIERTOS</i> . Para cada miembro de <i>M</i> que ya figurase en <i>ABIERTOS</i> o <i>CERRADOS</i> , decidir si se modifican o no sus apuntadores, dirigiéndolos a <i>n</i> (como se explica posteriormente). Para cada miembro de <i>M</i> que estuviese ya en <i>CERRADOS</i> , decidir, para cada uno de sus descendientes en <i>G</i> , si se modifican o no sus apuntadores (Ver texto).
8	Reordenar la lista <i>ABIERTOS</i> con arreglo a cualquier esquema arbitrariamente adoptado o de acuerdo con su mérito heurístico.
9	Ir a CICLO

Este algoritmo de exploración de grafos se convertirá en un algoritmo A a través de la ordenación de los estados del conjunto de estados abierto de acuerdo a algún mérito heurístico y a través de la heurística admisible que hemos diseñado y explicado antes, este algoritmo A se convertirá en un algoritmo A*.

El código fuente de esta implementación se adjuntará en la carpeta de la entrega pero no se mostrará en el informe debido a su longitud.

```

rmelgo@melgo:~/Escritorio/FSI/Algortimo A*$ gcc listaEnlazadaSimple.c main.c -o listaEnlazadaSimple
rmelgo@melgo:~/Escritorio/FSI/Algortimo A*$ ./listaEnlazadaSimple

Solucion encontrada con exito!!!

Se han explorado 97 estados

86 (7, 4, 0)
77 (7, 4, 3)
65 (7, 5, 3)
56 (7, 6, 3)
44 (7, 6, 0)
38 (6, 6, 0)
33 (5, 6, 0)
27 (4, 6, 0)
25 (3, 6, 0)
22 (2, 6, 0)
13 (1, 6, 0)
10 (1, 6, 1)
3 (1, 5, 1)
1 (1, 4, 1)
0 (1, 4, 0)

```

The diagram shows a 6x7 grid with obstacles represented by thick black lines. The start cell is at (4,1) and the end cell is at (6,7). Red arrows indicate the path: (4,1) → (4,2) → (5,2) → (5,3) → (4,3) → (4,4) → (5,4) → (5,5) → (5,6) → (5,7) → (6,7).

De esta manera, hemos obtenido el mismo resultado que el obtenido en la versión iterativa del backtracking en la que se hacía una búsqueda en amplitud. Sin embargo, esta solución es mucho más eficiente debido al número de estados que se han tenido que visitar ya que en la estrategia iterativa del backtracking se exploraban todas las combinaciones posibles a todas las profundidades hasta encontrar la mejor solución (de menor profundidad). En

cambio con el algoritmo A* cada estado solo se va a expandir una vez. De esta manera, como nuestro laberinto es de 6x7 y para cada casilla se tienen 4 orientaciones diferentes se tiene un total de $4*6*7=168$ estados por lo que sabemos que no se van a expandir más de 168 estados lo cual es un límite superior bastante bueno en comparación con el backtracking.

En el caso particular de nuestra solución, se han explorado 97 estados lo que supone que casi la mitad de los estados no han sido explorados pues se ha visto que en ningún caso conducen a la solución óptima. Además la mayoría de estos estados visitados corresponde a la parte inferior del laberinto, pues estas casillas inferiores son las que forman parte de la solución óptima y tienen un mejor valor para la heurística, siendo ordenadas primero en el conjunto de abiertos.

Ahora, como curiosidad, vamos a modificar el cálculo de esta heurística para que nuestro algoritmo pierda esta característica de admisibilidad y no se pueda garantizar que la solución obtenida es óptima. Para ello, vamos a modificar esta heurística para "priorizar" la búsqueda por la parte superior del mapa donde claramente el camino para llegar a la solución es mucho más largo y enrevesado.

$$|(x_{final} - x_{actual})| + (((y_{actual} - y_{final}) + 4) * 20)$$

Con esta modificación conseguimos que los estados con un valor para y más bajo tengan más prioridad y se ordenen primero en el conjunto de abiertos que los estados con coordenada y más alta (que son en general los que llevan a la solución óptima). De esta manera, vamos a conseguir desviar la exploración para que la solución obtenida no sea óptima.

Si ahora ejecutamos nuestra implementación del problema con el uso de esta función heurística modificada obtenemos el siguiente resultado:

```

rnelgo@melgo:~/Escritorio/FSI/Algortimo A*$ gcc listaEnlazadaSimple.c main.c -o listaEnlazadaSimple
rnelgo@melgo:~/Escritorio/FSI/Algortimo A*$ ./listaEnlazadaSimple

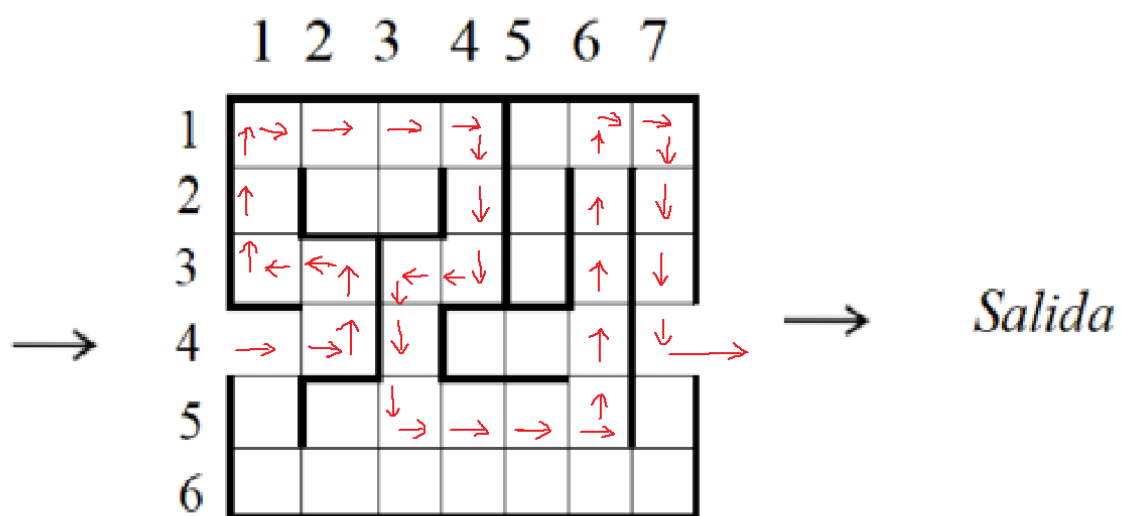
Solucion encontrada con exito!!!

Se han explorado 133 estados

114 (7, 4, 0)
106 (7, 4, 1)
100 (7, 3, 1)
96 (7, 2, 1)
95 (7, 1, 1)
91 (7, 1, 0)
88 (6, 1, 0)
85 (6, 1, 3)
82 (6, 2, 3)
80 (6, 3, 3)
78 (6, 4, 3)
74 (6, 5, 3)
71 (6, 5, 0)
69 (5, 5, 0)
68 (4, 5, 0)
59 (3, 5, 0)
57 (3, 5, 1)
56 (3, 4, 1)
54 (3, 3, 1)
51 (3, 3, 2)
39 (4, 3, 2)
34 (4, 3, 1)
30 (4, 2, 1)
27 (4, 1, 1)
24 (4, 1, 0)
23 (3, 1, 0)
20 (2, 1, 0)
17 (1, 1, 0)
16 (1, 1, 3)
13 (1, 2, 3)
11 (1, 3, 3)
10 (1, 3, 2)
7 (2, 3, 2)
6 (2, 3, 3)
2 (2, 4, 3)
1 (2, 4, 0)
0 (1, 4, 0)

```

De manera gráfica el recorrido el robot hasta alcanzar el estado final es el siguiente:



Como se puede observar, efectivamente al cambiar la forma de calcular la heurística se ha desviado en medida de lo posible la exploración por la parte superior del mapa y se ha obtenido una solución bastante mala pues este algoritmo A ha perdido esta característica de admisibilidad y el resultado obtenido no se puede garantizar que sea el óptimo.

Como curiosidad obsérvese cómo en este caso el número de estados expandidos ha sido mayor que en el caso anterior (133 estados expandidos) ya que se ha explorado una porción del mapa más grande que en el caso anterior.

Con esto, hemos podido observar como la implementación del algoritmo A* es capaz de encontrar la solución óptima siempre y cuando hagamos uso de una heurística admisible que sobreestime el coste real de alcanzar la solución. También se ha podido ver cómo al perder esta característica de admisibilidad, la solución obtenida no ha sido para nada óptima.

También hay que tener en cuenta que hemos modificado esta heurística de manera deliberada para forzar un resultado distinto y no óptimo. También es posible que el uso de una heurística no admisible proporcione el resultado óptimo, pero en estos casos no podemos garantizar que esta solución obtenida sea la mejor.