Inteligencia Artificial

Guía 3.1. Robótica

4° Licenciatura En Sistemas de Información

2020

Universidad: UADER FCyT Concepción del Uruguay

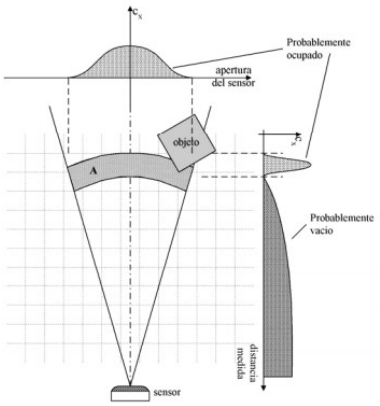
Profesor: Lopez De Luise Daniela, Bel Walter

Alumnos: Exequiel Gonzalez, Cepeda Leandro

1. Describa al menos cinco tipos de desplazamientos en robots móviles.

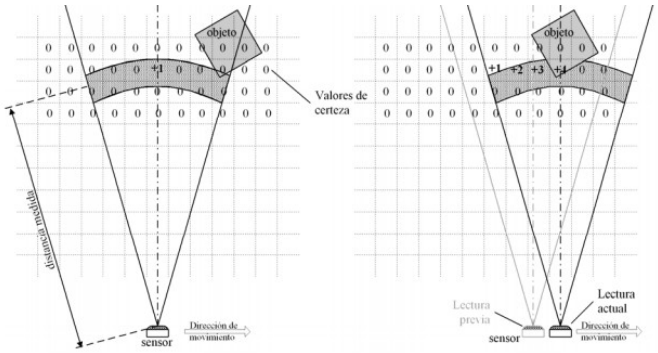
- Método de detección de bordes o esquinas.
Un método muy usado para eludir obstáculos con robots móviles está basado en la detección de bordes. En este método un algoritmo, a partir de la información dada por un sensor ultrasónico, trata de determinar la posición de los bordes verticales de los obstáculos y dirigir al robot alrededor de uno de los bordes "visibles". La línea que conecta dos bordes visibles es considerada para representar uno de los límites del obstáculo. La desventaja con la implementación de este método es que el vehículo debe detenerse frente al obstáculo para que los sensores tomen la información. Otra variante del método de "detección de bordes" es aquella en donde el robot permanece estacionario un lapso de tiempo tomando una visión panorámica del ambiente. Una desventaja común de ambas aplicaciones es la poca precisión debida al sistema sensorial empleado.

- La grilla de certeza para la representación de obstáculos
Este método, llamado "grilla de certeza" [21], es especialmente apto para la utilización de sensores imprecisos como los de ultrasonido. Es un método para representación probabilística de obstáculos en un modelo global sobre una grilla. En la grilla de certeza, el área de trabajo del robot es representada por un arreglo bidimensional de elementos cuadrados, llamados celdas. Cada celda contiene un valor de certeza (cv) que indica la confianza de que un obstáculo exista en una celda. Con este método, los cv son actualizados por una función de probabilidad que toma en cuenta las características del sensor utilizado. En la aplicación de este método, el robot móvil permanece estacionario un lapso tomando una imagen panorámica. Luego, la función probabilística Cx, es aplicada a cada una de las lecturas de los sensores, actualizando la grilla de certeza.

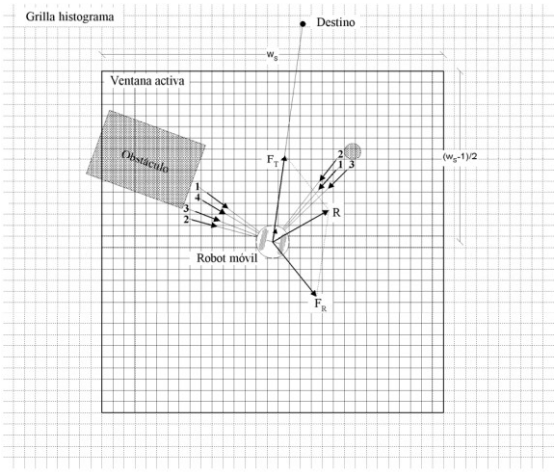


- El método del campo de potencial
La idea de fuerzas imaginarias actuando sobre el robot fueron sugeridas por Khatib. En este método, los obstáculos ejercen fuerzas repulsivas, mientras que el punto destino aplica una fuerza atractiva al robot. Una fuerza resultante “r”, que comprende la suma de las fuerzas atractivas y repulsivas, se calcula para una determinada posición del robot. Con “r” como la fuerza aceleradora actuando sobre el robot, se determina la nueva posición de éste para un intervalo de tiempo dado, y el algoritmo se repite nuevamente. Krogh fue mejorando este concepto tomando en consideración la velocidad del robot en la vecindad de los obstáculos. Thorpe aplicó el método del campo de potencial para planificar el camino fuera de línea (off-line).

- Método del campo de fuerzas virtuales (VFF)
El método VFF (Virtual Force Field) es el primer método que permite evitar obstáculos en tiempo real para vehículos autónomos rápidos. El VFF, a diferencia de los otros métodos, permite un control de movimiento continuo y rápido del vehículo a través de obstáculos inesperados y no requiere que el vehículo se detenga frente al obstáculo.
El método VFF usa una grilla-histograma cartesiana bidimensional C, para la representación de obstáculos. Como en el concepto de la grilla de certeza, cada celda i,j mantiene en la grilla un valor de certeza cij, que representa la confianza del algoritmo en la existencia de un obstáculo en esa ubicación.
El método de la grilla de certeza para la representación de obstáculos proyecta un contorno de probabilidad sobre las celdas que están afectadas por un juego de lecturas (es decir se ha detectado un obstáculo); este procedimiento es computacionalmente intensivo y podría imponerse un tiempo de penalización alto si la ejecución en tiempo real fuera intentada. En el método VFF se incrementa solo una celda en la grilla-histograma para cada juego de lecturas, creando una distribución de probabilidad con solo un pequeño gasto computacional. Para un sensor ultrasónico, esta celda corresponde a la distancia medida



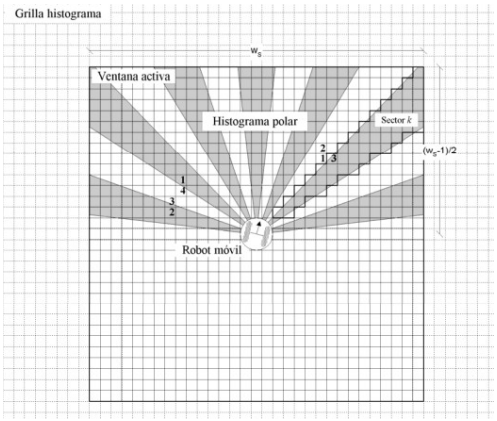
Luego se aplica la idea del campo de potencial a la grilla-histograma, así la información del sensor (información probabilística) puede usarse eficientemente para controlar el vehículo. Como el vehículo se mueve, una ventana de ws x ws celdas acompaña a éste, ocupando una región cuadrada de C. Se llama a esta región la "región activa". La ventana está siempre centrada en la posición del robot.



Cada celda activa ejerce una "fuerza repulsiva virtual" fij, contra el robot. Para cada iteración todas las fuerzas repulsivas virtuales se totalizan para producir la fuerza repulsiva resultante fr. Simultáneamente una "fuerza atractiva virtual" ft, de magnitud constante es aplicada al vehículo, "tirándolo" hacia su destino. La suma de fr y ft produce la fuerza resultante r.
En la práctica cada rango de lecturas es almacenado en la grilla-histograma tan pronto como sea posible y el siguiente cálculo de r toma estos datos en consideración. Esta característica otorga al vehículo una respuesta rápida ante los obstáculos que aparecen repentinamente, resultando en una conducta más reactiva, necesaria para altas velocidades.

- **Método del histograma de campo vectorial (VFH)**

Los inconvenientes del método VFF son que una reducción excesivamente drástica de datos ocurre cuando las fuerzas repulsivas individuales de las celdas de la grilla-histograma son totalizadas, para calcular la fuerza resultante fr. Como consecuencia, la información detallada acerca de la distribución local de obstáculos es muy pobre. Para remediar el problema se desarrolla un nuevo método denominado "Histograma de Campo Vectorial" (VFH). Este método emplea una técnica de reducción de datos en dos estados mucho más simple que la técnica de un solo paso usada por el método VFF. Existen tres niveles de representación de datos:
1. El nivel más alto retiene la descripción detallada del ambiente del robot. En este nivel, la grilla-histograma cartesiana bidimensional C, es continuamente actualizada en el tiempo con el juego de datos muestreados por el conjunto de sensores.
2. Para el nivel intermedio, un histograma polar unidimensional h, se constituye alrededor de la ubicación momentánea del robot. h comprende n sectores angulares de ancho γ. Una transformación mapea la región activa C* sobre h, resultando en que cada sector k retiene un valor hk que representa la "densidad polar de obstáculos" en la dirección que
corresponde al sector k.



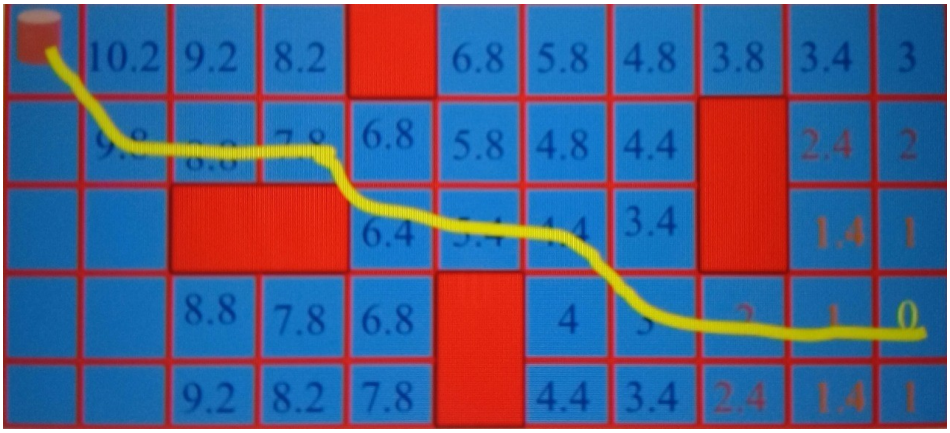
3. El nivel más bajo de representación de datos es la salida del algoritmo VFH y son los valores de referencia para el controlador de dirección y velocidad del vehículo.

Este método presenta algunas desventajas. Por tratarse de un método para evitar obstáculos en tiempo real la velocidad de procesamiento de la información está en relación con la máxima velocidad que puede desarrollar el robot. Por otro lado el volumen de información con que trabaja este método requiere una gran capacidad de almacenamiento de información en el robot.

https://www.aadeca.org/pdf/CP_monografias/monografia_robot_movil.pdf

2. Uno de los métodos de desplazamiento consiste en la proclamación dinámica, donde el espacio de trabajo se conoce y se cuadricula para mapearlo en el espacio de configuraciones. Cada configuración se asigna con un costo típicamente relacionado con el consumo de recursos y bondad del desplazamiento requerido. La traza se calcula conforme al costo mínimo final cuando con el costo global se maximiza. De ser necesario se recalcula el costo desde el inicio, cuando debe cambiarse la traza. El costo debe ser monótono decreciente. Esto es que cada nuevo costo siempre es menor o igual al actual.

El siguiente grafico muestra un ejemplo de traza y su espacio de trabajo



a- Realice un diseño de cómo representaría vectorialmente este world map.

Una manera de representar vectorialmente el Worldmap podría ser la siguiente, donde:
· X representa un obstáculo.
· 0 representa los espacios libres.
· * representa el avatar.
· # representa el destino.

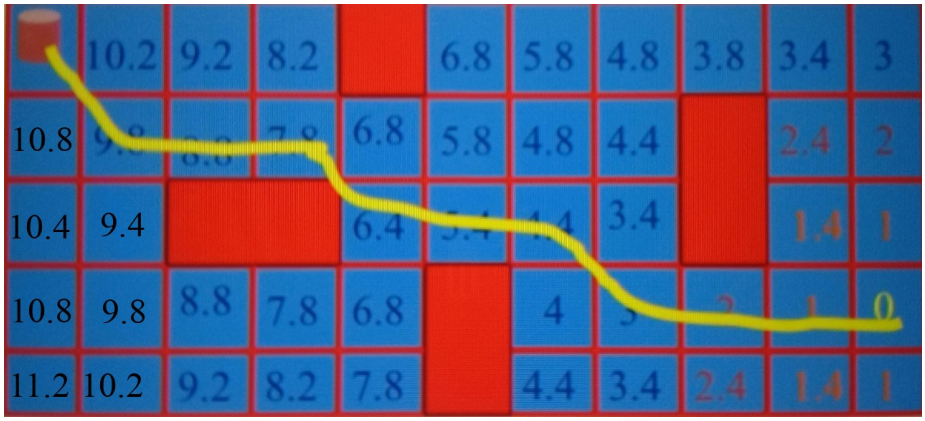
```
[[*,0,0,0,X,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0,X,0,0],
 [0,0,X,X,0,0,0,0,X,0,0],
 [0,0,0,0,0,X,0,0,0,0,#],
 [0,0,0,0,0,X,0,0,0,0,0]]
```

Otra manera de representar vectorialmente el Worldmap podría ser la siguiente, donde:
· ∞ representa un obstáculo.
· -1 representa el avatar.
· 0 representa el destino.

```
[[-1  , 10.2, 9.2, 8.2,   ∞, 6.8, 5.8, 4.8, 3.8, 3.4, 3],
 [10.8,  9.8, 8.8, 7.8, 6.8, 5.8, 4.8, 4.4,   ∞, 2.4, 2],
 [10.4,  9.4,   ∞,   ∞, 6.4, 5.4, 4.4, 3.4,   ∞, 1.4, 1],
 [10.8,  9.8, 8.8, 7.8, 6.8,   ∞,   4,   3,   2,   1, 0],
 [11.2, 10.2, 9.2, 8.2, 7.8,   ∞, 4.4, 3.4, 2.4, 1.4, 1]]
```

b- Genere la bd del gráfico en Prolog. Procure completar las casillas azules que aún no tienen costeo de modo que la traza resultante no sea alterada. Justifique.

Para completar las casillas azules que aun no tenían costeo, procuramos seguir el patrón ya establecido, teniendo en cuenta su representación simétrica en cuanto a los valores que ya se encontraban establecidos en las filas superiores. De esta manera si moviéramos el avatar hacia la esquina inferior izquierda, el costo seria exactamente el mismo que el actual.



c- Diseñe el programa e implemente en Prolog para la lógica de control de desplazamiento. TIP: recuerde que el robot se desplaza hacia un objetivo (marcado en la fig con 0)

```
:- module(mazeSolver, [solve/3]).
:- use_module('maze-printer.pl', [print_maze/0, print_maze/1]).

%% True if moving from [X0, Y0] to [X1, Y1] is valid
available_move([X0,Y0], [X1,Y1]) :-
       adj_tile([X0,Y0], [X1,Y1]),
       %% available_tile(X0,Y0),
       available_tile(X1,Y1).

%% True if tile is inside the maze
inside_maze(X1,Y1) :-
       maze_size(A,B),
       X1>=1,
       X1=<A,
       Y1>=1,
       Y1=<B.

%% True tile can be used in a path
available_tile([X,Y]) :-
       available_tile(X,Y).
available_tile(X0,Y0) :-
       inside_maze(X0,Y0),
       \+ barrier(X0,Y0).

%% True if [X0, Y0] is adajacent to [X1, Y1]
adj_tile([X0,Y0], [X0,Y1]) :-
       (Y1 is Y0-1).
adj_tile([X0,Y0], [X1,Y0]) :-
       (X1 is X0-1).
adj_tile([X0,Y0], [X0,Y1]) :-
       (Y1 is Y0+1).
adj_tile([X0,Y0], [X1,Y0]) :-
       (X1 is X0+1).


%%%%%% New %%%%%%

%% Checks endpoints first, then gets list of paths, then choses shortest from that list.
%% Cut after getting path list prevents back tracking after displaying all shortest lists
solve(ST, ET, Path) :-
       available_tile(ST),
       available_tile(ET),
       maze_size(N,M),
       get_paths_list(ST, ET, N*M, [], PathList),
       !,
       get_shortest_path(Path, PathList),
       print_maze(Path).

%% Plucks a path from list, if its length is minimal then true
get_shortest_path(Path, PathList) :-
       get_shortest_path_length(PathList, Min),
       member(Path, PathList),
       length(Path, Min).

%% Gets the minimum of the lengths of paths in the list (True if Min is the minimum length)
get_shortest_path_length([H|T], Min) :-
       length(H, MinStart),
       get_shortest_path_length(T, MinStart, Min).

%% Recurses through the list, taking the minimum length that it finds as it goes
get_shortest_path_length([], Min, Min).
get_shortest_path_length([NewPath|T], LatestMin, Min) :-
       length(NewPath, N),
       NewMin is min(N, LatestMin),
       get_shortest_path_length(T, NewMin, Min).

%% recursively finds paths that satisify, when all are found
%% this returns false, and will move to next rule.
get_paths_list(ST, ET, MaxLength, Cumu, PathList) :-
       path_solver_plus(ST, ET, MaxLength, [ET], Path),
       length(Path, N),
       N =< MaxLength, %% redundant check assuming it works in the path solver
       \+ memberchk(Path, Cumu),
       get_paths_list(ST, ET, N, [Path|Cumu], PathList).
%% Order means this is only called when above rule fails (all paths are found)
%% acts as boundary condition for recursion.
get_paths_list(_, _, _, PathList, PathList).

%% Solves for path backwards (as it's easy to build list that way)
%% When the current tile is the start tile then end.
path_solver_plus(ST, ST, _, Path, Path):-
       !.
%% Sink effect on the start tile, if we are currently one tile away then move straight there.
path_solver_plus(ST, CurrentT, MaxLength, Cumu, Path) :-
       available_move(CurrentT, ST),
       length(Cumu, N),
       N < MaxLength,
       Path = [ST|Cumu],
       !.
%% Finds an available move, if we haven't been there, add to the path and recurse
%% If the current path being built gets bigger than max allowed length then not allowed
path_solver_plus(ST, CurrentT, MaxLength, Cumu, Path) :-
       length(Cumu, N),
       N < MaxLength,
       available_move(CurrentT, D),
       \+ memberchk(D, Cumu),
       path_solver_plus(ST, D, MaxLength, [D|Cumu], Path).
```

```
?- consult(load).
true.

?- solve([1,1],[4,11],Path).

      1 2 3 4 5 6 7 8 9 1011
    +-----------------------+
  1 | o . . . X . . . . . . |
  2 | o o o o o . . . X . . |
  3 | . . X X o o o . X . . |
  4 | . . . . . . X o o o o o |
  5 | . . . . . . X . . . . . |
    +-----------------------+
 Path = [[1, 1], [2, 1], [2, 2], [2, 3], [2, 4], [2, 5], [3, 5], [3|...], [...|...]|...].
```

d- Genere en Prolog y muestre el world map.

El siguiente script permite visualizar el world map:

```
:- module(mazePrinter, [print_maze/0, print_maze/1]).

print_maze :-
       print_maze([]).
print_maze(Path) :-
       maze_size(Vert, Hori), nl,
       print_header(1,Hori), nl,
       print_maze_horiz_edge(1,Hori), nl,
       print_inner_maze(1, Vert, Hori, Path), nl,
       print_maze_horiz_edge(1,Hori).

print_spacer_top :-
       write('     '). %%5

print_spacer_corner :-
       write('   '). %%3

print_spacer_row :-
       write(' '). %%1

print_header(From, To) :-
       print_spacer_top,
       print_column_header(From, To).

%% Prints column header until false, hence the need for the cut
%% which prevents back tracking after header is printed
print_column_header(To, To) :-
       write(To).
print_column_header(From, To) :-
       From>=To.
print_column_header(From, To) :-
       write(From),
       print_heading_inner_spacer(From),
       Next is From+1,
       print_column_header(Next, To),
       !.

print_maze_horiz_edge(From, To) :-
       print_spacer_corner,
       write('+'),
       print_maze_horiz_border(From, To),
       write('+').

%% Prints top/bottom border until false, hence the need for the cut
%% which prevents back tracking after border is printed
print_maze_horiz_border(To, To) :-
       write('---').
print_maze_horiz_border(From, To) :-
       From>=To.
print_maze_horiz_border(From, To) :-
       write('--'),
       Next is From+1,
       print_maze_horiz_border(Next, To),
       !.

%% Prints the inside of the maze (below top border, above bottom border)
%% Cut prevents back tracking when mze is printed
print_inner_maze(Row, Row, Columns, Path) :-
       print_maze_row(Row, 1, Columns, Path).
print_inner_maze(Row, RowTo, _, _) :-
       Row>=RowTo.
print_inner_maze(Row, RowTo, Columns, Path) :-
       print_maze_row(Row, 1, Columns, Path),
       nl,
       RowNext is Row+1,
       print_inner_maze(RowNext, RowTo, Columns, Path),
       !.

print_heading_inner_spacer(Row) :-
       Row > 9 ; write(' '),
       true.

print_maze_row(Row, ColumnFrom, ColumnTo, Path) :-
       print_spacer_row,
```

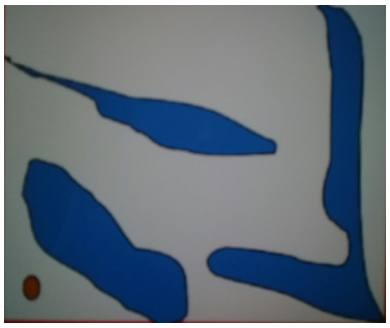
```
       write(Row),
       print_heading_inner_spacer(Row),
       write('|'),
       print_maze_row_inner(Row, ColumnFrom, ColumnTo, Path).

%% Prints a row of the inner maze, stops when it reachs maze size
%% Cute prevents backtracking after row is printed
print_maze_row_inner(Row, Column, Column, Path):-
       write(' '),
       print_symbol(Row, Column, Path),
       write(' |').
print_maze_row_inner(_, Column, ColumnTo, _) :-
       Column>=ColumnTo.
print_maze_row_inner(Row, Column, ColumnTo, Path) :-
       write(' '),
       print_symbol(Row, Column, Path),
       NextColumn is Column+1,
       print_maze_row_inner(Row, NextColumn, ColumnTo, Path),
       !.

%% Prints 'o' for tile in the path
%% Prints 'x' for a barrier
%% Prints '.' otherwise
print_symbol(Row, Column, Path) :-
       memberchk([Row,Column], Path), write('o').
print_symbol(Row, Column, _) :-
       barrier(Row, Column), write('x').
print_symbol(_, _, _) :-
       write('.').
```

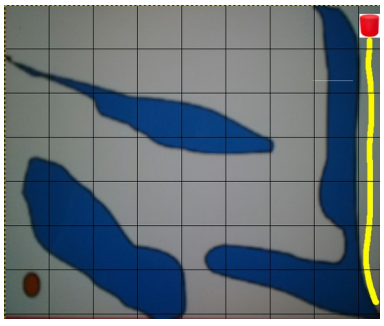
El siguiente código permite definir el world map:

```
:- module(maze, [maze_size/2, barrier/2]).

maze_size(5, 11).

barrier(1, 5).
barrier(2, 9).
barrier(3, 3).
barrier(3, 4).
barrier(3, 9).
barrier(4, 6).
barrier(5, 6).
```

```
?- consult(load).
true.

?- print_maze.

    1 2 3 4 5 6 7 8 9 1011
  +----------------------+
1 | . . . . x . . . . . . |
2 | . . . . . . . . x . . |
3 | . . x x . . . . x . . |
4 | . . . . . x . . . . . |
5 | . . . . . x . . . . . |
  +----------------------+
true.
```

3. Considere el programa del ejercicio anterior.
a- Genere un world map conforme el siguiente espacio de trabajo.





```
?- consult(load).
true.

?- print_maze.

    1 2 3 4 5 6 7 8 9
  +------------------+
1 | . . . . . . . x . |
2 | x x . . . . . x . |
3 | . . x x x x . x . |
4 | . . . x x . . x . |
5 | . x x . . . . x . |
6 | . x x x . x x x . |
7 | . . x x . . x x . |
  +------------------+
true.
```

b- Considere el punto rojo como la posición destino, suponiendo su robot sale del vértice superior derecho. Genere el costeo y muéstrelo en el espacio de configuraciones.

Una manera de representar vectorialmente el Worldmap sin los valores costeados podría ser el siguiente, donde:
· X representa un obstáculo.
· 0 representa camino libre.
· * representa el avatar.
· # representa el destino.

```
[[0,0,0,0,0,0,0,X,*],
 [X,X,0,0,0,0,0,X,0],
 [0,0,X,X,X,X,0,X,0],
 [0,0,0,X,X,0,0,X,0],
 [0,X,X,0,0,0,0,X,0],
 [0,X,X,X,0,X,X,X,0],
 [#,0,X,X,0,0,X,X,0]]
```

Una manera de representar vectorialmente el Worldmap con los valores costeados podría ser el siguiente, donde:

· ∞ representa un obstáculo.
· * representa el avatar.
· 0 representa el destino.

```
[[ 6, 7, 8, 9, 10, 11, 12, ∞,  *],
 [ ∞, ∞, 7, 8,  9, 10, 11, ∞, 13],
 [ 4, 5, ∞, ∞,  ∞,  ∞, 10, ∞, 12],
 [ 3, 4, 5, ∞,  ∞,  8,  9, ∞, 11],
 [ 2, ∞, ∞, 5,  6,  7,  8, ∞, 10],
 [ 1, ∞, ∞, ∞,  5,  ∞,  ∞, ∞,  9],
 [ 0, 1, ∞, ∞,  4,  5,  ∞, ∞,  8]]
```

c- Corra su controlador y muestre la nueva traza.

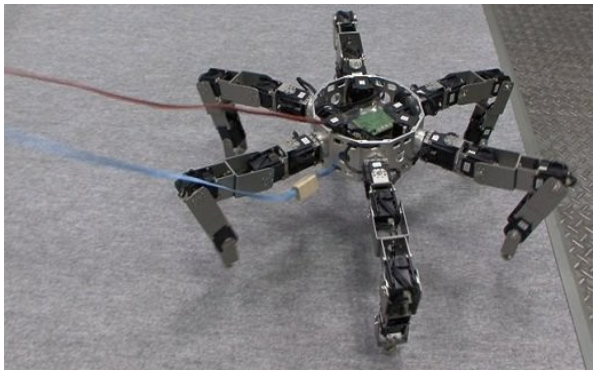
El siguiente código permite definir el world map:

```
:- module(maze, [maze_size/2, barrier/2]).

maze_size(7, 9).

barrier(1, 8).
barrier(2, 1).
barrier(2, 2).
barrier(2, 8).
barrier(3, 3).
barrier(3, 4).
barrier(3, 5).
barrier(3, 6).
barrier(3, 8).
barrier(4, 4).
barrier(4, 5).
barrier(4, 8).
barrier(5, 2).
barrier(5, 3).
barrier(5, 8).
barrier(6, 2).
barrier(6, 3).
barrier(6, 4).
barrier(6, 6).
barrier(6, 7).
barrier(6, 8).
barrier(7, 3).
barrier(7, 4).
barrier(7, 7).
barrier(7, 8).
```

```
?- consult(load).
true.

?- print_maze.

    1 2 3 4 5 6 7 8 9
  +------------------+
1 | . . . . . . . x . |
2 | x x . . . . . x . |
3 | . . x x x x . x . |
4 | . . . x x . . x . |
5 | . x x . . . . x . |
6 | . x x x . x x x . |
7 | . . x x . . x x . |
  +------------------+
true.

?- solve([1,9],[7,1],Path).
false.
```

4. Determine los grados de libertad de los siguientes efectores.

a. Grados de libertad: 18 grados rotatorios en las patas.



b. Grados de libertad: 4 grados en las ruedas, 2 en las puertas.

c. Grados de libertad: 2 grados en las ruedas.

d. Grados de libertad: 7 grados en total; 2 grados rotatorios, 1 grado cilíndrico, 4 grados en las pinzas.