DECSAI
Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Inteligencia Artificial en Investigación Operativa

Curso académico 2012/2013

Práctica 2

Planificación para vehículos autónomos

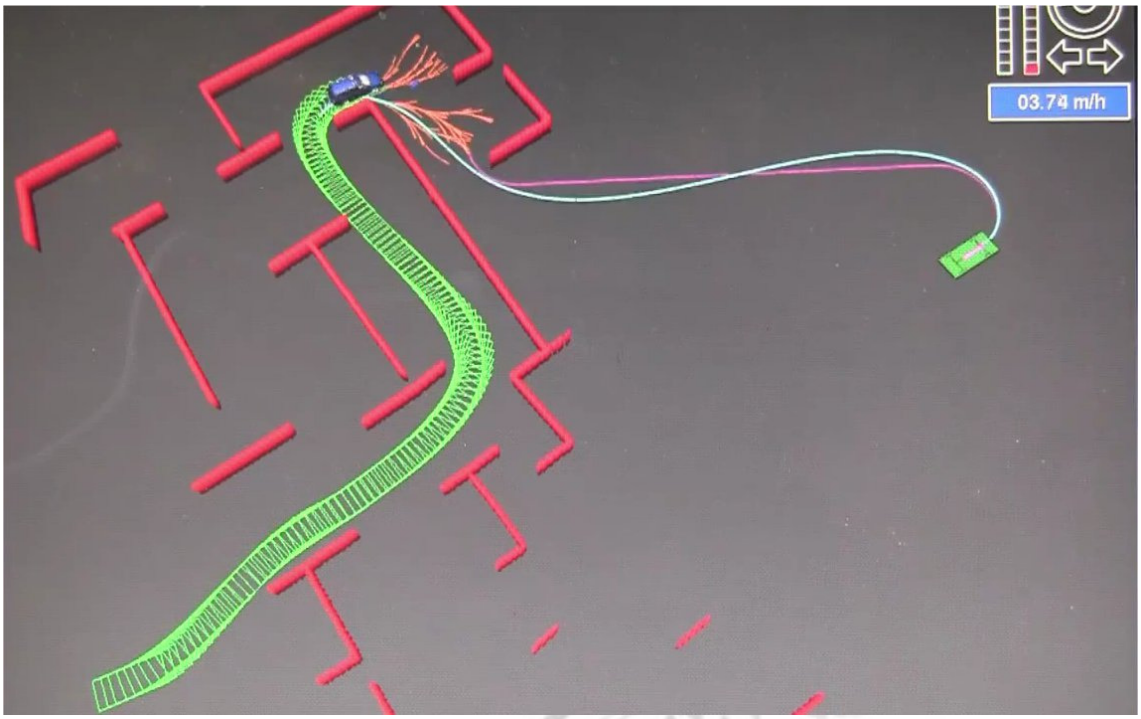
© *Fernando Berzal*

ENTREGA DE LA PRÁCTICA
(a través del acceso identificado de DECSAI)
`https://decsai.ugr.es/`

```
resultados.m
initialState.m
nextState.m
sameState.m
finalState.m
heuristic.m
cost.m
stochasticCost.m
```

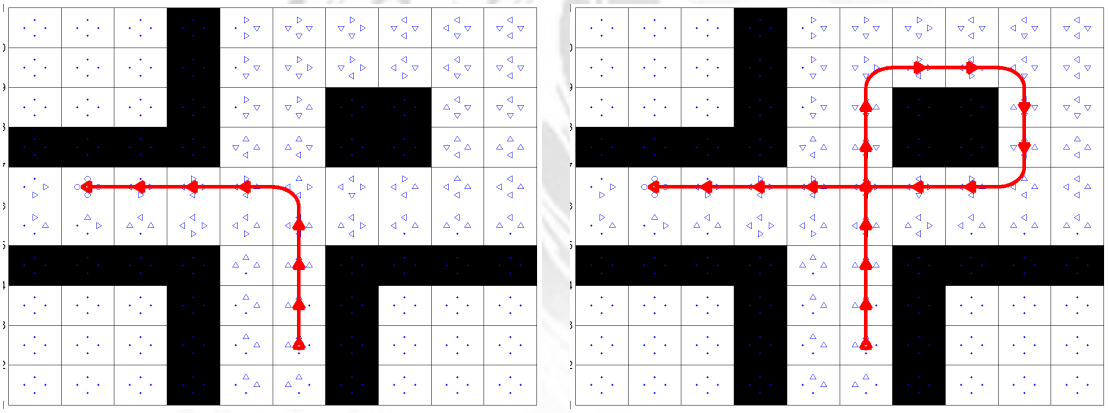
Planificación para vehículos autónomos

En esta práctica, comprobaremos cómo podemos utilizar distintas técnicas para planificar la trayectoria de un vehículo autónomo fijando como objetivo el destino del trayecto que el vehículo autónomo ha de realizar.



Uso del algoritmo A* en la planificación de trayectorias del coche autónomo de Google.

En la primera parte de la práctica, utilizaremos técnicas de búsqueda para encontrar la ruta más adecuada para llegar a nuestro destino, mientras que en la segunda parte utilizaremos programación dinámica para determinar la "política" óptima que ha de gobernar el movimiento del vehículo autónomo, incluso en entornos estocásticos.



Ficheros de MATLAB

Junto con el guión de la práctica, tiene a sus disposición una serie de ficheros en los que ya están implementadas algunas estrategias de planificación y un conjunto de plantillas que deberá rellenar para conseguir que su implementación funcione correctamente. Además de estos ficheros, también ha de rellenar el fichero `resultados.m` con sus respuestas a los distintos ejercicios propuestos en este guión.

A continuación se describe la función de los distintos ficheros de los que dispone:

- Los ficheros `*Board.m` crean distintas configuraciones de tableros sobre los que se moverá nuestro vehículo autónomo.

- El fichero `actions.m` define las distintas acciones que puede realizar nuestro vehículo autónomo: seguir en línea recta [`G`], girar a la izquierda [`L`] o girar a la derecha [`R`]. De forma análoga, el fichero `orientations.m` define las distintas orientaciones que puede tener el vehículo conforme se va moviendo: Norte [`N`], Sur [`S`], Este [`E`] y Oeste [`W`].

- Los ficheros `newState.m`, `initialState.m`, `nextState.m`, `finalState.m` y `sameState.m` son los ficheros que se utilizan para representar nuestro problema como un problema de búsqueda.

- El fichero `heuristic.m` es el fichero que implementa las distintas heurísticas que se pueden utilizar con el algoritmo A*, cuya implementación se proporciona en el fichero `astar.m`.

- Los ficheros `mdp*.m` y `stochastic*.m` tendrá que utilizarlos en la segunda parte de la práctica, en la que verá cómo se puede utilizar programación dinámica para resolver problemas de planificación de trayectorias.

- Los ficheros `actionCost.m`, `cost.m` y `stochasticCost.m` deberá utilizarlos para calcular adecuadamente los costes asociados a las distintas acciones que puede realizar el vehículo.

- El fichero `collision.m` nos permite saber si el vehículo chocaría con algún obstáculo en caso de que intentase realizar una acción determinada a partir del estado indicado (puede utilizar la implementación de esta función como base para la resolución de la práctica).

- Los ficheros `path.m`, `interpolatedPath.m` y `smoothedPath.m` se utilizan para reconstruir el camino que físicamente recorre el vehículo al moverse sobre el tablero.

- Los ficheros `createFigure.m` y `show*.m` incluyen funciones utilizadas para la visualización de resultados (tablero, políticas y rutas seguidas por el vehículo).

- El fichero `demo.m` sirve de demostración del uso de las distintas funciones implementadas en esta práctica.

- El fichero `test.m` incluye una batería de casos de prueba que una implementación correcta de la práctica debería superar con éxito (su calificación final dependerá [parcialmente] de su grado de éxito a la hora de superar esta batería de prueba).

- El fichero `resultados.m` deberá rellenarlo con sus respuestas a las distintas cuestiones y ejercicios propuestos en este guión (junto con los casos de prueba de `test.m`, determinará su calificación de la práctica).

Configuraciones del problema

Las funciones definidas en los ficheros `*Board.m` se utilizan para crear distintas configuraciones de tableros sobre los que luego se efectuará la planificación:

```
board = frontierBoard(n);
```
crea un tablero sencillo de tamaño `nxn`, sin obstáculos de ningún tipo.

```
board = clearBoard(n);
```
crea otra configuración de un tablero de tamaño `nxn` sin obstáculos de ningún tipo (con las casillas inicial y final en esquinas opuestas del tablero).

```
board = pacmanBoard(n);
```
crea un tablero de tamaño `nxn` con una configuración similar al de un comecocos.

```
board = zigzagBoard(n);
```
crea un tablero de tamaño `nxn` con paredes que nos obligan a ir haciendo zigzag.

```
board = costBoard(n);
```
crea un tablero de tamaño `nxn` en el que el coste de recorrer las casillas inferiores es superior al coste de las casillas superiores, lo que nos empujará siempre a ir hacia arriba si queremos encontrar la ruta óptima.

Además de los tableros anteriores, también dispone de 3 tableros que nos permitirán observar el comportamiento de nuestro vehículo en una intersección de tráfico:

`board = loopBoard(n);`
crea un tablero de tamaño nxn con un cruce.

`board = laneBoard(n);`
crea un tablero de tamaño nxn con un cruce y calles de dos carriles con costes asociados diferentes (en el que resulta más arriesgado circular por el lado izquierdo de la calzada).

`board = threeLaneBoard(n);`
crea un tablero de tamaño nxn con un cruce y calles de tres carriles.

La implementación de todos estos tableros tiene una estructura similar e incluye información que le resultará necesaria para su implementación de la práctica:

- `board.n` nos indica el tamaño del lado del tablero
- `board.start.x`, `board.start.y` y `board.start.orientation` nos dan las coordenadas de la casilla y la orientación inicial de nuestro vehículo.
- `board.goal.x` y `board.goal.y` nos proporcionan las coordenadas de la posición en la que debe quedar finalmente nuestro vehículo.
- `board.cost(x,y)` nos indica el coste que supone atravesar la casilla (x,y).
- `board.wall` nos indica el coste que supone chocar contra un obstáculo (o salirnos del tablero).

Representación de los estados del problema

Su primera tarea consiste en definir adecuadamente la forma de representar los estados de nuestro problema, que consiste en conseguir que nuestro vehículo llegue hasta su objetivo (marcado por `board.goal.x` y `board.goal.y`) desde su posición inicial (que viene dada por sus coordenadas, `board.start.x` y `board.start.y`, y por su orientación inicial, `board.start.orientation`).

En este problema, deberá mantener tanto la posición actual de nuestro vehículo (sus coordenadas x e y, que guardaremos en `state.x` y `state.y`) como su orientación (almacenada en `state.orientation`), junto con el coste que supone haber llegado hasta el estado actual (valor que almacenaremos en `state.cost`). Internamente, para reconstruir el camino hasta la solución, también se utiliza una quinta variable [`state.prev`] que **NO** deberá modificar bajo ningún concepto si desea que su implementación funcione correctamente.

Para representar el problema adecuadamente y poder resolver nuestro problema de planificación como un problema de búsqueda, deberá proporcionar una implementación válida de las siguientes funciones en MATLAB:

- `initialState(board)` crea el estado inicial a partir de los datos proporcionados por el tablero sobre el que se moverá nuestro vehículo.

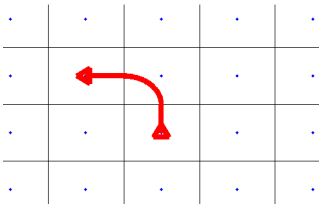
- `sameState(this, other)` nos indica si dos estados, `this` y `other`, representan el mismo estado de búsqueda o no (algo que resulta necesario para evitar ciclos al realizar la búsqueda, p.ej. cuando se llega a estados ya visitados).

- `finalState (board, state)` nos indica si el estado `state` corresponde a uno de los posibles estados finales para el problema de búsqueda definido por el tablero `board`.

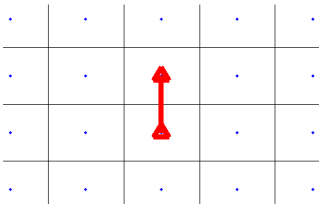
- `nextState (board, state, action)` es la función que utilizaremos para obtener el estado al que se llega cuando, estando en el estado `state`, aplicamos la acción `action` en el problema de búsqueda definido por el tablero `board`. En esta función deberá calcular correctamente las coordenadas de nuestro robot (sin salirse del tablero) y su orientación, además calcular el coste que supone llegar al nuevo estado a partir del actual (para lo cual utilizaremos la función auxiliar `cost`.

En nuestro problema, las acciones disponibles son las definidas por la función `actions`: `G` (moverse en línea recta, en función de la orientación actual del vehículo), `R` (avanzar girando a la derecha) y `L` (avanzar girando a la izquierda).

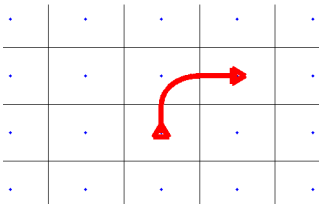
Asegúrese de reflejar correctamente los resultados de las distintas acciones en su implementación. Gráficamente, esto es lo que sucede con cada una de las acciones:



Girar a la izquierda
(acción `L`)



Avanzar
(acción `G`)



Girar a la derecha
(acción `R`)

No olvide implementar la función `cost(board,state,action,next)`, que nos permite calcular el coste de, partiendo del estado inicial `state` en el tablero `board`, aplicar la acción `action` y llegar al estado `next`.

En la implementación de la función `cost` deberá tener en cuenta lo siguiente:

- Cada acción tiene un coste diferente, dado por la función `actionCost()`, que asumimos que siempre será mayor o igual que cero.

- El coste asociado a atravesar la casilla a la que nos movemos viene dado por `board.cost(next.x, next.y)`.

- Cuando nos encontramos en un estado determinado y, al intentar avanzar, chocamos con algún obstáculo, aunque no consigamos movernos, incurrimos en un coste que debemos tener en cuenta. Utilice la función auxiliar `collision(board,state,action)` para determinar si se produce una colisión y, cuando esto suceda, añada al coste de su acción el coste del choque (dado por `board.wall`).

El coste total de la acción, por tanto, es la suma del coste de ejecutar la acción, el coste que supone atravesar la casilla del tablero a la que nos movemos y la posible penalización que se produciría en caso de colisión.

Antes de pasar al siguiente apartado, asegúrese de que su implementación de las funciones anteriores funciona correctamente diseñando distintos casos de prueba.

Planificación mediante el algoritmo A*

Para utilizar el algoritmo A*, lo primero que tiene que hacer es implementar las distintas funciones heurísticas que utilizaremos en esta práctica en el fichero `heuristic.m`. En particular, debe implementar las siguientes heurísticas:

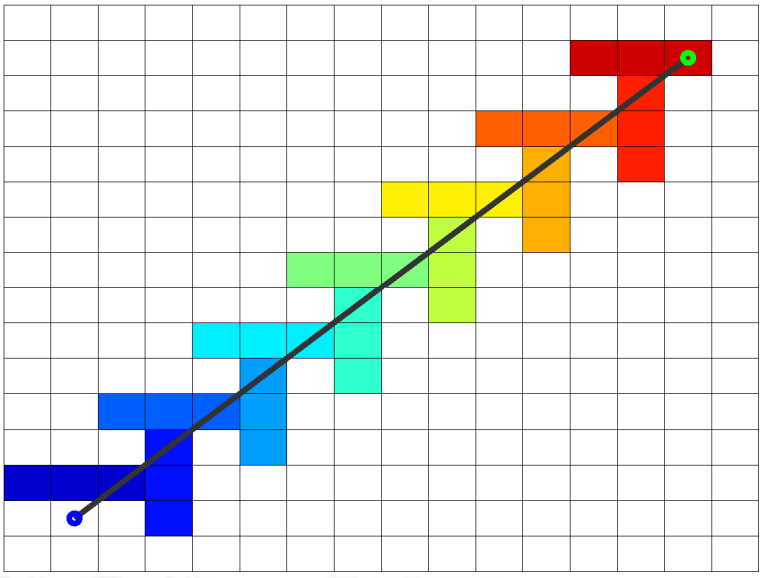
- `'ucs'` para el algoritmo de búsqueda de coste uniforme (que se diferencia del algoritmo A* en que no utiliza el valor heurístico de un nodo para determinar el siguiente nodo que se expandirá, sólo el coste para llegar al nodo en cuestión).

- `'manhattan'` para una heurística basada en la distancia de Manhattan (de la posición correspondiente al estado evaluado con respecto a la posición del objetivo).

- `'chessboard'` para emplear la distancia de Chebyshev, también conocida como distancia del tablero de ajedrez (de la posición correspondiente al estado evaluado con respecto a la posición del objetivo).

- `'euclidean'` para emplear una heurística basada en la distancia euclídea (de la posición correspondiente al estado evaluado con respecto a la posición del objetivo).

¡OJO! La heurísticas resultantes deben ser siempre admisibles para cualesquiera valores devueltos por la función `actionCost`. Para ello, deberá tener en cuenta el caso límite en el que la función `actionCost` siempre devuelve `0` como resultado. En este caso, el algoritmo A* debe comportarse como muestra la siguiente figura para un tablero de tipo `clearBoard` utilizando cualquiera de las 3 heurísticas definidas:



EJERCICIO 1: EJECUCIÓN DEL ALGORITMO A*

La función definida en `astar.m` implementa el algoritmo de búsqueda A* utilizando la heurística que se le indique como segundo parámetro (el mismo parámetro que recibe la función que ya debe tener implementada en `heuristic.m`).

Ejecute el algoritmo A* sobre distintas configuraciones de tableros y anote sus resultados en el fichero `resultados.m` utilizando las heurísticas `ucs`, `manhattan`, `euclidean` y `chessboard`:

1. `frontierBoard`, de tamaño 16x16.
2. `clearBoard`, de tamaño 16x16.
3. `pacmanBoard`, de tamaño 16x16.
4. `zigzagBoard`, de tamaño 16x16.
5. `costBoard`, de tamaño 16x16.
6. `loopBoard`, de tamaño 16x16.
7. `laneBoard`, de tamaño 16x16.
8. `threeLaneBoard`, de tamaño 16x16.

Al realizar los experimentos indicados, asegúrese de que el coste individual de ejecutar las acciones es `0` cuando seguimos en línea recta [acción `G`] y `2` cuando giramos, tanto a la izquierda como a la derecha [acciones `L` y `R`].

Indique, en `resultados.m`, el número de nodos expandidos por el algoritmo A*, la longitud del camino encontrado hasta la solución y su coste total, que debería ser el mismo para las distintas heurísticas, ya que todas serán admisibles si las ha implementado correctamente.

Observe cómo, dependiendo del problema de búsqueda concreto, la heurística más adecuada puede ser diferente. En este caso, ¿con cuál de las tres heurísticas se quedaría?

EJERCICIO 2: EL COSTE DE LAS ACCIONES

Las pruebas anteriores se han realizado asumiendo que las acciones que realiza nuestro vehículo tienen el mismo coste, cuando esto no es siempre así. En la práctica, en el caso de un automóvil, girar a la izquierda suele tener un "coste" mayor que girar a la derecha, por lo que es algo que deberíamos tener en cuenta.

En este ejercicio, nos centraremos en dos de las configuraciones que ya hemos utilizado: `loopBoard`, de tamaño 16x16, y `laneBoard`, también de tamaño 16x16.

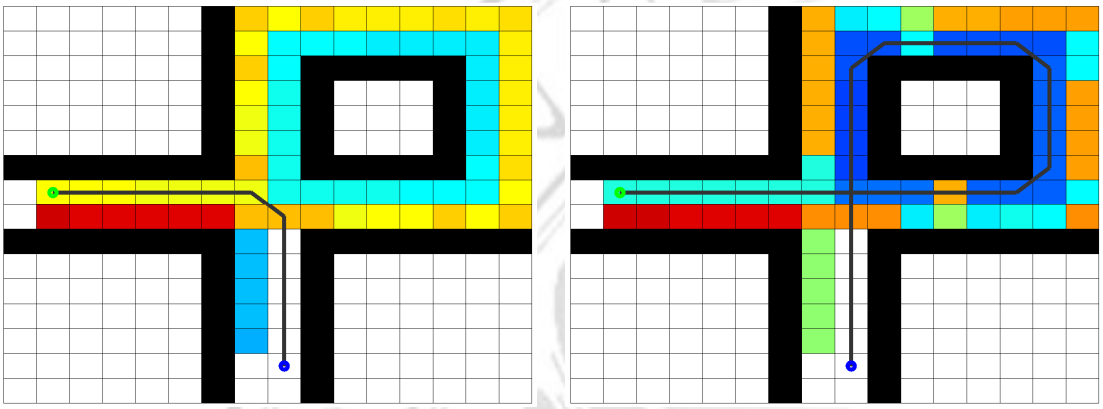
Compruebe lo que sucede si modificamos el fichero `actionCost.m` para hacer que el coste de girar a la izquierda (`L`) sea superior al de girar a la derecha (`R`) para los siguiente valores de coste:

1. `coste('R')=2, coste('L')=20`.
2. `coste('R')=2, coste('L')=200`.

Indique, en `resultados.m`, el número de nodos expandidos por el algoritmo A* utilizando la distancia de Manhattan, la longitud del camino encontrado hasta la solución y su coste total.

EJERCICIO 3: VALORES CRÍTICOS

Como habrá podido observar, al crecer el coste de girar a la izquierda, llega un momento en el que, en vez de arriesgarnos en el cruce girando a la izquierda, nuestro planificador basado en el algoritmo A* decide que es mejor darle la vuelta a la manzana realizando 3 giros de 90º a la derecha, evitando así el riesgo de una posible colisión.



Si asumimos que el coste de girar a la derecha es 2, ¿cuál sería el valor que debería tener el coste de girar a la izquierda para que nuestro planificador cambie de estrategia en un tablero `loopBoard` de tamaño 16x16 y decida no girar a la izquierda? ¿Y si el tablero fuese de 20x20? ¿De 24x24? ¿De 32x32? ¿De 100x100? ¿De 1000x1000?

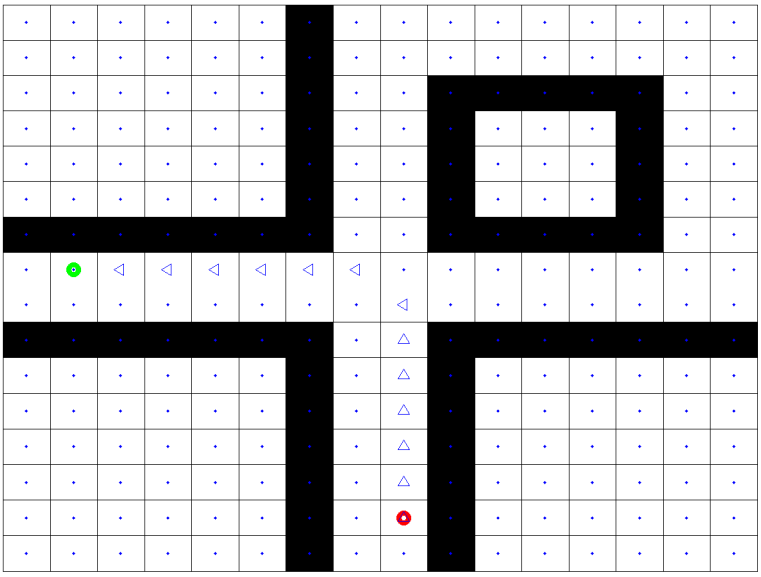
Indique sus respuestas en el fichero de resultados `resultados.m`.

Planificación mediante programación dinámica

Implícitamente, cuando estamos calculando la ruta óptima con el algoritmo A*, estamos definiendo cuál es la acción más adecuada para nuestro vehículo autónomo en cada uno de los estados intermedios en los que se encuentra a lo largo de su trayectoria. Cualquier otra ruta que pase por esos mismos estados intermedios seguirá los mismos pasos que la ruta escogida.

Gráficamente, podemos ver las decisiones tomadas en cada uno de los estados intermedios si utilizamos el siguiente fragmento de código:

```
board = laneBoard(16);
[path, nodes, policy] = astar (board,'manhattan');
showPolicyMap (board, policy, 'A* policy');
```



Las decisiones tomadas a lo largo de la trayectoria del vehículo, tal como se han determinado mediante la aplicación del algoritmo A*, son las óptimas para el problema y definen una política [de actuación].

La programación dinámica proporciona un método alternativo para la planificación de trayectorias en situaciones como las planteadas en esta práctica. Igual que el algoritmo A*, encontrará el camino de menor coste desde la posición inicial hasta nuestro objetivo. Es más, nos proporcionará directamente el camino de menor coste desde cualquier posición inicial posible.

En la situación de arriba, imaginemos que, al girar a la izquierda, nos encontramos un obstáculo que nos impide seguir por el camino previsto, ¿qué hacemos entonces? Tendremos que buscar una ruta alternativa... o consultar la política que nos proporciona la acción óptima en cualquier situación.

Para determinar la política óptima que controlará el movimiento de nuestro vehículo en cualquier estado posible, utilizaremos la función `mdpPolicy()`, que recibe como parámetro el tablero con la configuración de nuestro problema.

Para cada uno de los estados `(x,y,o)` de nuestro problema, donde `(x,y)` son las coordenadas del vehículo y `o` es su orientación (`{N,S,E,W}`), podemos definir el valor asociado a las distintas acciones tal como se describe a continuación:

Sean S el conjunto de estados de nuestro problema y A el conjunto de acciones posibles:

- Definimos una función de transición *next(s,a)* que nos proporciona el estado resultante s' de, partiendo del estado s, ejecutar la acción a.

- Definimos una función de coste *cost(s,a,s')* que nos proporciona el coste de, partiendo del estado s, ejecutar la acción a y llegar al estado s'.

- La política óptima $\pi^*: S \rightarrow A$ es la política que minimiza la función de coste para cada estado $s \in S$.

¿Cómo calculamos la política óptima?

La función de transición ya la hemos definido antes: nuestra función `nextState`.

La función de coste también la hemos creado ya teniendo en cuenta el coste de atravesar una casilla concreta del tablero, el coste asociado a la acción escogida y la posibilidad de que acabemos chocando con un obstáculo (o saliéndonos del tablero).

Una vez definidas estas funciones, podemos partir de la siguiente expresión recursiva para calcular la política óptima que controlará los movimientos de nuestro vehículo autónomo:

$$v(s) = \min_{a \in A} \{ cost\big(s, a, next(s,a)\big) + v\big(next(s,a)\big) \}$$

Dada la expresión anterior, el valor asociado a cada estado lo podemos calcular de forma iterativa utilizando programación dinámica.

De las distintas acciones posibles, elegiremos siempre aquélla que minimiza la suma del coste asociado a la acción y del valor del estado al que nos lleva. De esa forma, sabremos cuál es la acción que debe controlar el movimiento de nuestro vehículo en cualquier situación posible.

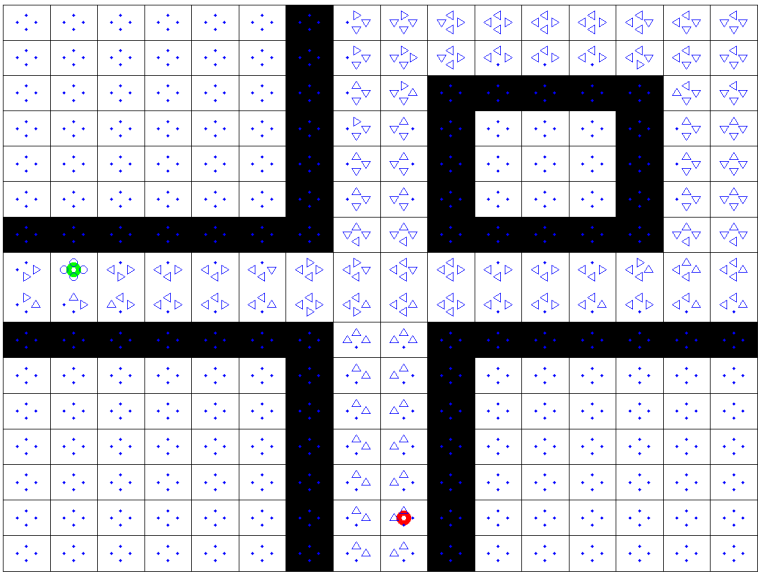
Para determinar la política óptima asociada a un tablero dado basta con realizar una llamada a la función `mdpPolicy()`:

```
policy = mdpPolicy(board);
```

Si quiere visualizar la política obtenida, puede recurrir a la función `showPolicy()`:

```
showPolicy (board, policy, 'MDP policy');
```

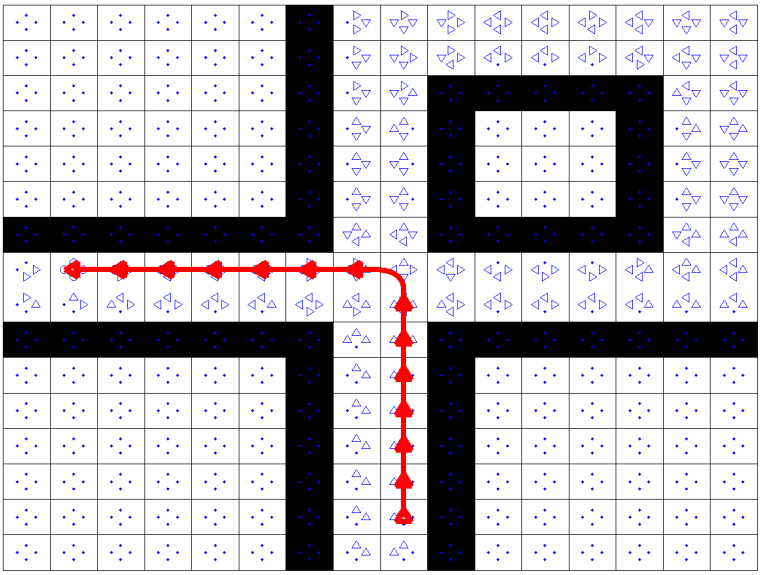
Dicha función muestra, en cada casilla del tablero, un triángulo que indica cuál es la acción más adecuada en función de la orientación en la que se encuentre nuestro vehículo:



Observe que, en la figura, se muestran algunos estados para los que no existe una política definida (aquellos estados desde los que no se puede llegar al objetivo).

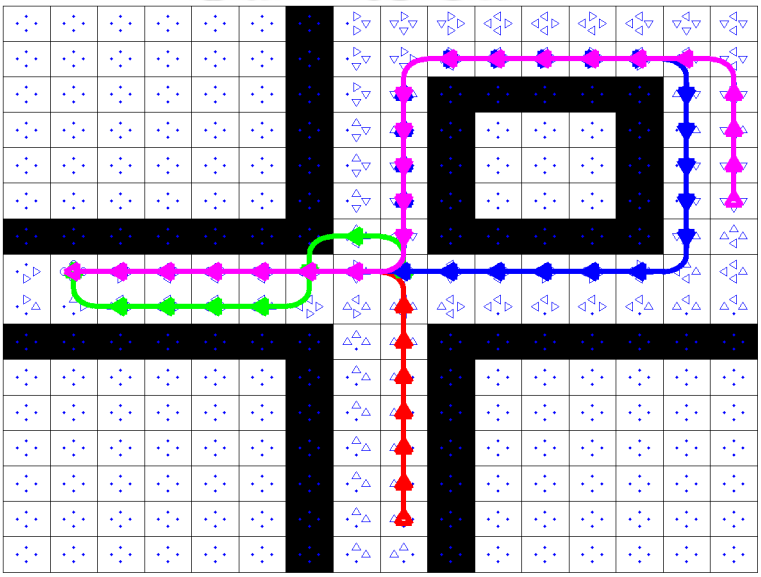
A partir de la política óptima, se puede reconstruir el camino que debe seguir nuestro vehículo desde cualquier posición en la que pueda encontrarse. Para obtener dicho camino y visualizarlo gráficamente, utilice las funciones `mdpPath()` y `showPath()`:

```
[path, orientation] = mdpPath(board, policy, x,y,o);
showPath(path,orientation);
```



Si lo desea, puede visualizar varios caminos simultáneamente indicando colores diferentes a través de un argumento opcional de la función showPath(), p.ej.

```
showPath(p1,o1,'g');
showPath(p2,o2,'r');
showPath(p3,o3,'b');
```



EJERCICIO 4: CAMINOS ÓPTIMOS

En un tablero como el mostrado en las figuras anteriores, de tipo `laneBoard` y de tamaño 16x16, ¿cuál es la longitud del camino óptimo que se obtiene desde las siguientes coordenadas iniciales para cada una de las orientaciones posibles?

(9,2) (9,9) (8,16) (16,8) (15,15)

Observe cómo, en ocasiones, dependiendo de nuestra orientación inicial, no existe ningún camino posible hasta la solución, por lo que no nos movemos de la casilla de partida y la función `mdpPath()` nos devuelve un camino de longitud 0.

Planificación estocástica

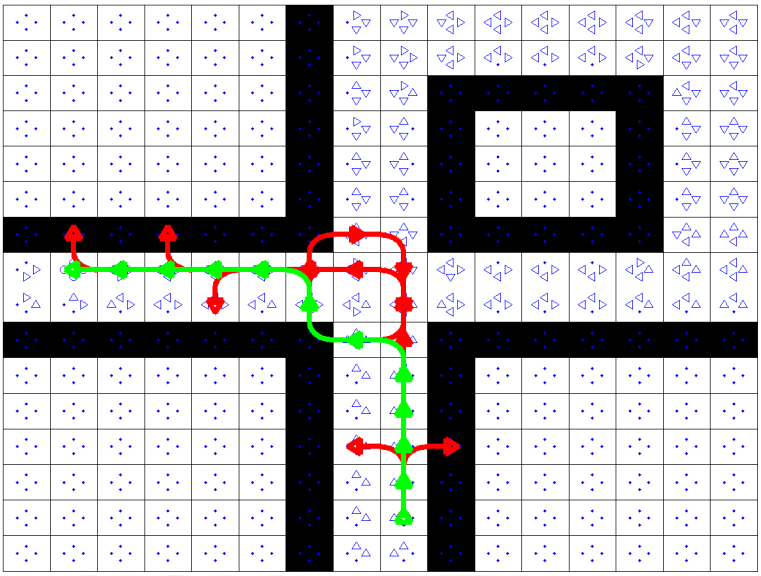
El uso de programación dinámica nos permite modelar situaciones en las que existe incertidumbre.

Por ejemplo, podemos trabajar cuando los actuadores (o la respuesta) de nuestro vehículo no son del todo fiables y conseguir que nuestro vehículo siga funcionando [la mayor parte del tiempo].

Formalmente, podemos modelar la fiabilidad de nuestro vehículo mediante una probabilidad de fallo que utilizaremos para evaluar las posibles consecuencias de que la acción que le indiquemos a nuestro vehículo no se ejecute correctamente.

Por ejemplo, si la probabilidad de fallo del control de la dirección del vehículo es del 1% (en la práctica será mucho inferior si queremos que el vehículo cumpla con su función), el 1% de las veces que intentemos seguir en línea recta, el vehículo se desviará (asumamos que la mitad de las veces se irá a la izquierda y la otra mitad, a la derecha). De forma análoga, el 1% de las veces que le indiquemos que curve, el vehículo se moverá en línea recta.

Estos fallos tienen consecuencias (podemos chocar con obstáculos) y la política óptima que seguirá el vehículo variará con respecto a la versión determinista (no estocástica) del problema, como se puede ver en la siguiente figura:



La figura recoge 100 ejecuciones y muestra en verde la política óptima que controla el movimiento de nuestro vehículo. No obstante, en ocasiones el vehículo se desvía de su ruta "óptima", tal como se muestra en rojo. A veces, eso nos impide lograr nuestro objetivo (p.ej. desvíos al comienzo y al final del trayecto) mientras que otras veces somos capaces de corregir el rumbo rectificando una vez que nos damos cuenta del desvío con respecto a la ruta prevista (giro a la izquierda y bucle al maniobrar en el cruce).

Como puede ver, la política óptima no es exactamente la misma que teníamos en el caso determinista. ¿Por qué?

Si somos observadores, nos daremos cuenta de que el extraño camino elegido por nuestro vehículo autónomo minimiza el riesgo de una colisión en el cruce y resulta más adecuado que la maniobra que realizaba antes (aunque esta parezca más razonable a simple vista).

EJERCICIO 6: PLANIFICACIÓN ESTOCÁSTICA

Implemente la función `stochasticCost (board, state, action, next, failureProbability)` que nos permite calcular el coste de, partiendo del estado inicial `state` en el tablero `board`, aplicar la acción `action` y llegar al estado `next`. cuando la probabilidad de que se produzca un fallo es `failureProbability`.

En su implementación, puede utilizar directamente la función `cost (board, state, action, next)` que ya creó con anterioridad. Sólo debe tener en cuenta que, para estimar el coste asociado a la acción, debe incluir el coste asociado al resultado que se obtendría si la acción no se ejecuta con éxito. En otras palabras:

- Si decidimos girar, a la izquierda o a la derecha, puede que el vehículo no nos responda con probabilidad P, por lo que seguirá su camino en línea recta. Por tanto, el coste asociado a la ejecución de la acción será (1-P)*C(ok) + P*C(error), donde C(ok) es el coste asociado a la ejecución correcta de la acción y C(error) es el coste asociado a seguir en línea recta.

- De forma análoga, cuando queramos seguir en línea recta (acción G), puede que nos desviemos de nuestro rumbo. Asumiendo que la probabilidad de desviarnos hacia ambos lados es la misma, el coste asociado a nuestra acción será (1-P)*C(ok) + (P/2)*C(L) + (P/2)*C(R).

Una vez implementada la función de coste en el fichero `stochasticCost.m`, puede determinar la política óptima asociada a un entorno estocástico llamando a la función `stochasticPolicy()`:

```
policy = stochasticPolicy (board, failureProbability);
```

Si quiere visualizar la política obtenida, puede usar la función `showPolicy()`, igual que para el caso no estocástico:

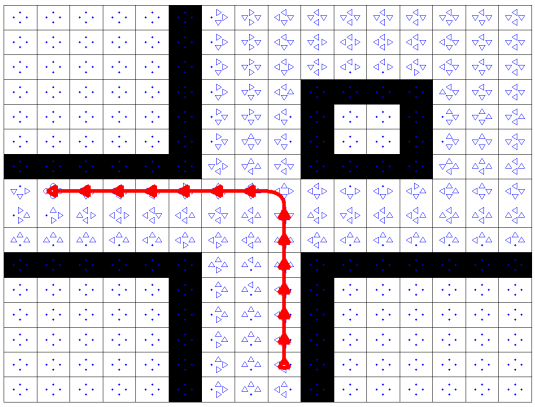
```
showPolicy (board, policy, 'Stochastic policy');
```

Ahora, para simular el comportamiento estocástico del vehículo, utilice `stochasticPath()` en vez de `mdpPath()`:

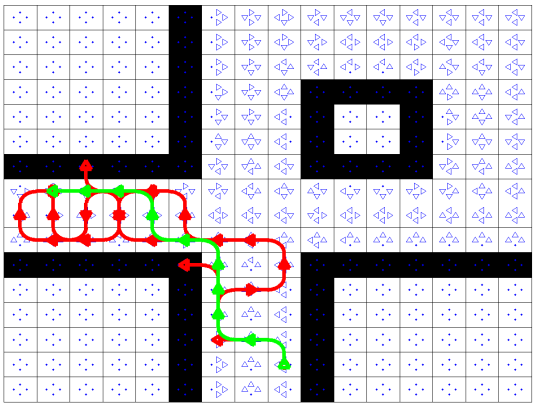
```
stochasticPath (board, policy, x, y, o, p);
```

`mdpPath()` en este caso, le puede servir para visualizar la política óptima que se intenta seguir desde un estado inicial dado, que puede diferir del camino real que se sigue.

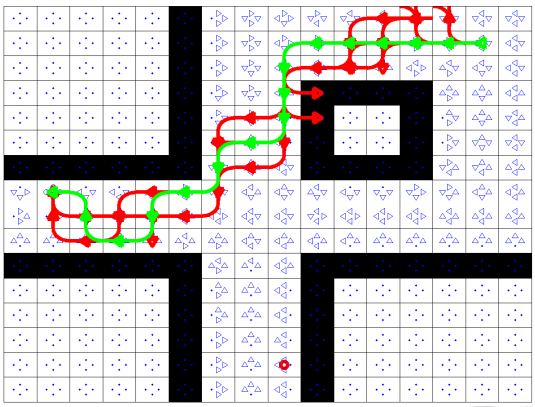
Compruebe los curiosos efectos que provoca el comportamiento estocástico de nuestro vehículo utilizando un tablero de tipo `threeLaneBoard`:

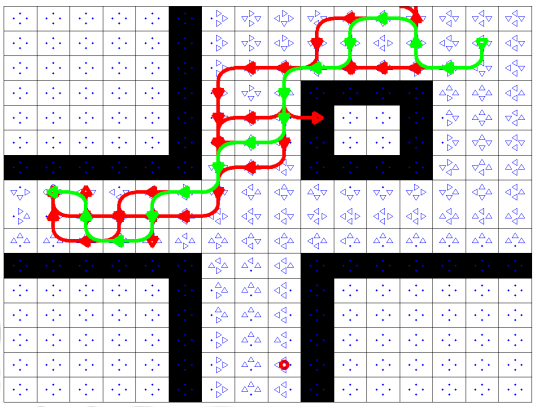


Planificación determinista



Planificación estocástica





Planificación estocástica desde una misma posición con distinta orientación inicial.

EVALUACIÓN DE LA PRÁCTICA

Para comprobar el funcionamiento de su implementación de la práctica, ejecute `test`, que lanza una batería automatizada de casos de prueba sobre las funciones que haya definido.

Tras remitir su fichero de resultados, se realizará una evaluación automática de los valores indicados en su fichero de resultados de la práctica, `resultados.m`.

El 50% de su calificación corresponderá a las respuestas que indique en `resultados.m`, mientras que el 50% restante provendrá de hasta qué punto satisfaga los casos de prueba definidos en `test.m`.