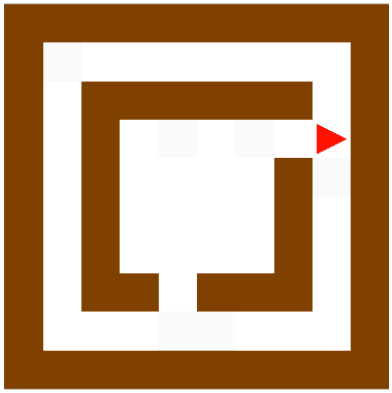
1. Descripción del Algoritmo

La práctica nos propone dos retos principales a la hora de diseñar el algoritmo: Debe ser robusto a la hora de no desperdiciar movimientos en choques con obstáculos, debe evitar quedarse atascado en pasillos estrechos y debe evitar "peinar" demasiado una zona dejando una sin recoger trufas.

1.1 La Matriz

Mi primera decisión a la hora de desarrollar el algoritmo fue el desarrollar un mapa donde el agente pudiese ir guardando aquellas posiciones que son obstáculos para así, una vez choque con uno, no volver a chocar dos veces con el mismo. Como la práctica nos asegura que los mapas siempre serán de 10x10, una matriz parece una buena opción donde guardar estos datos.



La idea es que una vez choque el robot con una casilla, registre el obstáculo en la matriz y no vuelva a chocar.

El primer problema que se me planteó con esta forma de actuar era que de entrada no conocemos en qué posición del mapa comienza el robot, ni tampoco tenemos forma de conocerlo, por lo que se corría el riesgo de que la matriz desbordase si fuese de 10x10. Finalmente solucioné esto desarrollando una **matriz de 20x20** en la que coloco el robot en la posición central al inicio (la 9,9), por lo que independientemente de donde comience, sé que no desbordará.

1.2 Obstáculos

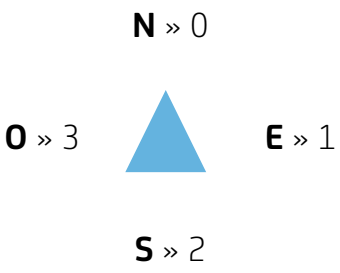
Mi matriz es una matriz de enteros que comienza con todas las casillas inicializadas a 0, en la que, cada vez que detecto que una casilla es detectada como obstáculo, pongo su valor a -1.

Para hacer esta asignación he creado una función llamada **AsignarObstaculo()**, la cual llamo en el caso de que se detecte que el robot chocó en su último movimiento (Es decir, que se activó el flag "*bump_*").

1.3 El Movimiento

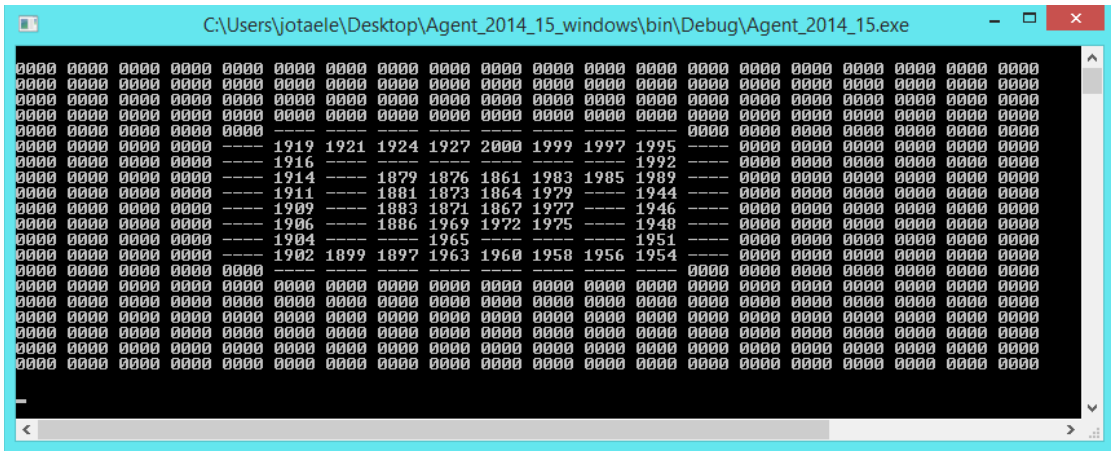
Una vez salvado el primer reto que suponen los obstáculos del mapa, tocaba desarrollar una forma de recorrer el mapa que intentase que ninguna casilla acabase en tierra baldía. Para ello, mi algoritmo finalmente usa de nuevo la matriz: En cada movimiento he creado una especie de *timestamp*, en la que el agente cambia en la matriz el valor de su posición actual por el del turno en el que pasa por esa casilla.

Una vez tengo en una matriz los valores con el turno en el que el robot pasó por cada casilla, puedo desarrollar un sistema, encapsulado en la función **MejorCasilla()**, que comprueba los valores en la matriz de las 4 casillas colindantes a la posición actual del robot (Norte, Sur, Este y Oeste). Para ello, busca el valor más pequeño de estos (es decir, la casilla por la que hace más tiempo que se pasó) garantizando así el que el robot visite todas las zonas del mapa de forma equitativa.



La función modula mediante enteros los puntos cardinales (Norte = 0, Este = 1, Sur = 2, Oeste =3) y devuelve el punto cardinal más adecuado en forma de entero.

Cabe destacar que esta función sufrió un pequeño retoque para contemplar el caso en que dos casillas empaten en valor. En ese caso, la función siempre considera mejor que el robot continúe recto antes de que gire si una de las casillas que empatan coincide con la orientación en ese momento del agente.

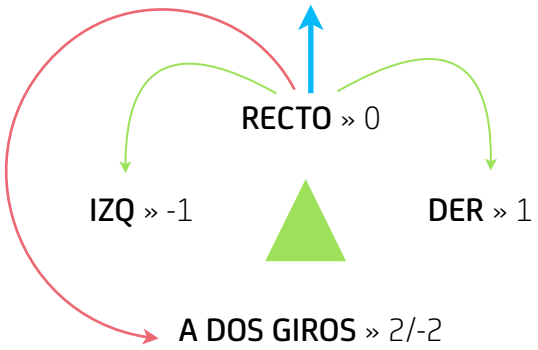


Representación de la matriz que crea el robot del primer mapa. Los valores representan en qué turno pasó por última vez el robot por esa casilla y los "-" representan los muros detectados.

1.4 La Orientación

Una vez tengo definida en cada posición del robot el mejor punto cardinal al que debe dirigirse en cada momento, sólo falta transformar eso en una orden al robot sobre si debe girar (y en tal caso en qué dirección) o si debe seguir recto.

La función **ElegirOrientacion(int ori_deseada)**, mediante la diferencia (Salvando el caso del 3) del punto cardinal actual junto con el deseado, permite determinar si para llegar al punto cardinal deseado el robot debe girar, o si por el contrario el punto cardinal deseado coincide con el actual y por tanto el robot debe seguir recto. Responde a esto mediante una asignación a enteros. El 0 representa continuar recto, el 1 representa girar a la derecha, el -1 girar a la izquierda, y si el resultado es 2 la casilla deseada está a más de un giro de la orientación actual, por lo que en ese caso he convenido que el robot siempre gire a la izquierda.



1.5 Decisión de la acción final

El valor de la función ElegirOrientacion() se le pasa finalmente al robot, determinando la acción final de éste, el cual determina su acción final a través de un switch. Cabe destacar que para mejorar la eficiencia final del robot, no recoge trufas de las casillas siempre que detecte que las hay, si no que espera a que haya al menos 5 trufas. En general, el esquema definitivo de selección de acción sería:

- Si la cantidad de trufas en cada casilla es desconocida (-1), la acción es OLFATEAR.
- Si las trufas detectadas son más de 5, entonces la acción será RECOGER.
- En cualquier otro caso, el robot GIRARÁ o AVANZARÁ según la función ElegirOrientación().

2. Tabla de Resultados para 10 Repeticiones

Mapa	Media de Trufas Recolectadas
Agent.map	1373.100
Agent_rap.map	2884.200
Mapa1.map	1564.100
Mapa1_rap.map	3222.500
Mapa2.map	1514.200
Mapa2_rap.map	3167.800
Mapa3.map	1617.100
Mapa3_rap.map	3199.900