
UNIVERSIDAD
DE GRANADA

2ºC

Grado en Ingeniería Informática

---

# Inteligencia Artificial:
# Práctica 2 - Los extraños mundos de Belkan.

---

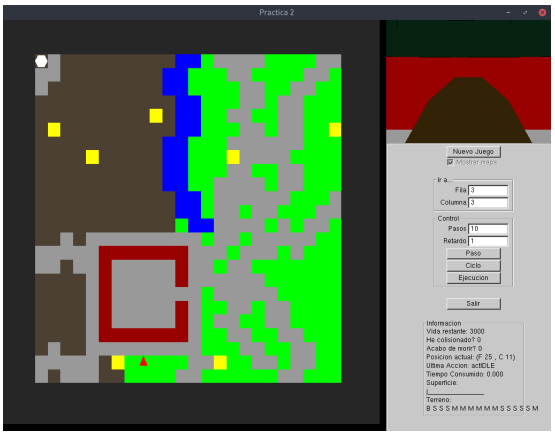
*Autor:*
Antonio David Villegas Yeguas

*Asignatura:*
*Inteligencia Artificial*

4 de mayo de 2019

# 1. Introducción.

En esta práctica se nos encargaba la tarea de implementar un agente reactivo/deliberativo usando un software que simula un videojuego, llamado "Los extraños mundos de Belkan".




Cada acción tiene un coste de vida, incluido el realizar un giro, y su misión será, para el nivel 1 que sea capaz de encontrar un camino hasta ese objetivo, y para el nivel 2, sobre el cual no tendrá conocimiento del mapa, ha de ser capaz de situarse en un punto (casillas amarillas), y una vez situado, encontrar el máximo número de objetivos en 300 segundos o en 3000 vidas.

Para encontrar esos objetivos, ha de buscar un plan usando algún algoritmo de búsqueda.

Para desarrollar esos algoritmos, usaremos distintas estructuras para tratar el mapa como un árbol de búsqueda.

La más importante, sera la estructura nodo:

```
//estructura nodo, trabajaremos con ella para consular los estados, la
// secuencia de acciones para llegar a ese estado, el coste de llegar a ese
// estado, la heuristica de ese nodo, y el coste total (coste + heuristica)
struct nodo{
   estado st;
   list<Action> secuencia;
   int coste;
   int heuristica;
   int coste_total;
};
```

En mi práctica presento cuatro algoritmos de búsqueda distintos:

1. Profundidad: En el que al explorar los nodos usaremos el estado (st) y una lista de acciones para llegar a ese estado (secuencia)
2. Anchura: Al igual que en profundidad, usaremos el estado y la lista de acciones para llegar a ese estado
3. Coste uniforme: En el que usaremos el estado, la secuencia de acciones, y además el coste, para encontrar una solución óptima
4. A*: En el que usaremos el estado, la secuencia de acciones, el coste, una heurística para estimar el coste de un camino

# 2. Nivel 1

En el nivel 1 nuestro personaje tendrá conocimiento del mapa, luego lo único que tendrá que hacer sera calcular un plan y seguirlo.

Esto lo podemos ver en el siguiente código:

Para empezar, actualizamos la fila y la columna del jugador, y establecemos la acción a realizar a IDLE.

Actualizamos la brújula y la fila y columna dependiendo de la ultima acción del jugador.

En caso de girar a la derecha, sumamos uno a la brújula, al girar a la izquierda sumamos tres a la brújula, y si estamos mirando al norte actualizamos la fila y la columna dependiendo de la orientación.

Si cambia el destino, volvemos a calcular el plan.

Si no hay plan, establecemos la fila, columna y orientación del estado actual a la fila y columna obtenida por los sensores y buscamos un plan.

Si hay plan, lo ejecutamos, teniendo en cuenta si hay obstáculos delante, y si no tenemos plan, realizamos un movimiento reactivo.

## 2.1. Comportamiento 1: Búsqueda en Profundidad

La búsqueda en profundidad se basa en expandir todos los nodos de un camino concreto, y cuando no quedan más nodos por explorar, vuelve al último nodo con hijos que dejó sin explorar.

Esta implementación la podemos conseguir usando una pila, ya que si siempre abrimos los nodos de la misma forma, en la cima de la pila siempre tendremos el resultado de explorar por un camino concreto, y cuando exploremos totalmente ese camino, en la cima quedara el último nodo que se dejó sin explorar, y de esta forma repetir el proceso hasta explorar todos los nodos, o encontrar una solución.

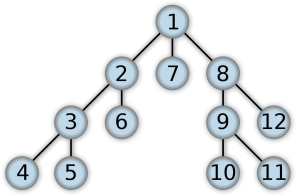
Ejemplo de generar un nodo:

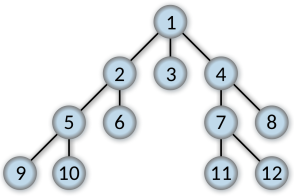
```
// Generar descendiente de girar a la derecha
nodo hijoTurnR = current;
hijoTurnR.st.orientacion = (hijoTurnR.st.orientacion+1)%4;
if (generados.find(hijoTurnR.st) == generados.end()){
   hijoTurnR.secuencia.push_back(actTURN_R);
   pila.push(hijoTurnR);
}
```

## 2.2. Comportamiento 2: Búsqueda en Anchura

La búsqueda en anchura se basa en expandir un nodo, y recorrer los nodos en el orden en el que se abrieron.

Recorrido en profundidad a la izquierda y en anchura a la derecha:




Esto lo podemos conseguir usando una cola, ya que exploramos los nodos en el orden que los introducimos.

Ejemplo de generar un nodo:

```
//Generamos nodo correspondiente a girar a la derecha
nodo rightSon = current;
rightSon.st.orientacion = (rightSon.st.orientacion+1) % 4;
if(generados.find(rightSon.st) == generados.end()){
   rightSon.secuencia.push_back(actTURN_R);
   cola_nodos.push(rightSon);
}
```

Un ejemplo de la ejecución del algoritmo:

## 2.3. Comportamiento 3: Búsqueda Coste Uniforme

La búsqueda de coste uniforme es una variación de la búsqueda en anchura, en la que en lugar de escoger el nodo en orden de apertura, se escoge el siguiente nodo por el costo de ir del nodo origen a ese nodo. Esto implica que si todos los nodos tienen el mismo coste, se realizará una búsqueda en anchura.

En este caso, para implementarlo en Los mundos de Belkan, usaremos un multiset de nodos, ordenados con un fuctor en el que únicamente se comparen los costes. Con esto conseguiremos que los nodos del multiset estén ordenados por coste.

Debido a que los ordenamos por coste, no podemos usar un set, ya que no permitiría almacenar dos nodos con el mismo coste, de ahí el usar el multiset.

Además en coste uniforme debemos tener en cuenta si podemos llegar a un estado de dos formas distintas, y seleccionar la de menor coste. Para ello uso la siguiente función, en la que considero que dos estados son iguales cuando la fila, la columna y la orientación son iguales, sin importar el coste:

```
// funcion para comparar si dos estados son iguales
bool ComparaEstado(const estado &a, const estado &n);
```

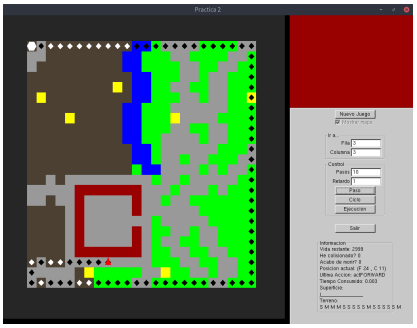
Esta función la usaremos para saber si el estado .ªza esta en la lista de abiertos, y tendremos que decidir si lo reemplazamos por el estado "n"

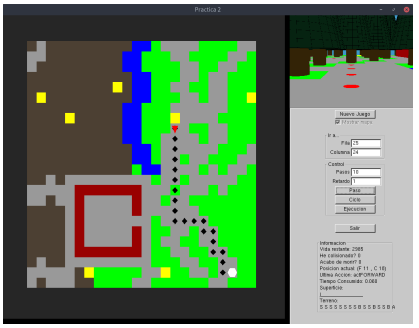
Para saber si un nodo esta contenido en el multiset, no podemos usar la función *find* que nos proporciona el multiset, debido a que deberíamos pasarle un coste por el que buscar, y queremos buscar nodos, luego usamos esta función, que nos devolverá un iterador apuntando al nodo, o a *abiertos.end()* en caso de que no encuentre el nodo:

```
multiset<nodo, OrdenaNodo>::const_iterator nodoEncontrado(const multiset<nodo, \
                                                           OrdenaNodo> & abiertos, \
                                                           const nodo & nodo);
```

Un ejemplo de la ejecución del algoritmo:

Comparación entre profundidad, anchura, y coste uniforme (de izquierda a derecha, en ese orden):





# 3. Nivel 2

En este nivel nuestro personaje no tiene conocimiento del mapa, es decir, solo sabe orientarse gracias a los sensores, los cuales no le dirán la fila y columna sobre la que se encuentra. El personaje ha de encontrar un punto de referencia (PK) en el que obtendrá la fila y la columna, y a partir de ahí sera capaz de saber que parte del mapa esta observando gracias que conoce su posición y al sensor *sensores.terreno*.

Para realizar esto, he dividido el método *think* en dos partes distintas gracias al sensor *sensores.nivel*.

Si se ejecuta en algún comportamiento del nivel 1, realizar las acciones expuestas en el apartado anterior, mientras que si el comportamiento es el 4, tendrá dos posibles comportamientos:

- Sí esta situado: Ha encontrado un PK, es capaz de saber donde se encuentra, y puede comenzar a buscar objetivos, usando el algoritmo de búsqueda
- No esta situado: Realizará un movimiento reactivo recordando por donde ha pasado para evitar repetir caminos y además si encuentra un PK en su rango de visión intentara ir a por el.

Estos dos comportamientos los conseguiremos con una variable *bool estoyBienSituado* que por defecto comenzará a *false* y cuando encuentre un PK se volverá *true*.

Para empezar, comprobará si aparece en una casilla PK, es decir, es capaz de leer la fila y la columna y actualizará la variable *estoyBienSituado*.

A partir de aquí dependerá del valor de *estoyBienSituado*:

Si ***estoyBienSituado == true***:

Actualizamos el valor de la fila y la columna, y la orientación actual.

Pintamos la parte del mapa que el personaje es capaz de observar usando *sensores.terreno* y escribiendo en *mapaResultado*.

Comprobamos si el destino se ha modificado, y en tal caso, invalidamos el plan estableciendo *hayPlan = false*.

Si no hay plan, actualizamos el estado actual, y buscamos un plan usando la función *pathFinding*.

Si encuentra un plan valido, lo ejecuta, siempre teniendo en cuenta posibles obstáculos, ejecutando la acción que este en el frente de *plan*.

Además, como no sabemos si el plan pasa por casillas obstaculo, o casillas que nos generaran demasiado coste, debido a que nuestro personaje no sabe el estado del mapa en una zona en la que no haya pasado, existe una variable *recalcular* inicializada a 3.

Cada acción que realicemos, esa variable se vera reducida en una unidad, y cuando llegue a 0, volveremos a calcular el plan, ya que tendremos nueva información del entorno.

Como también tenemos la limitación de 300 segundos de tiempo de búsqueda de un camino, y cada vez que recalculamos consume ese tiempo, el valor de *recalcular* varia según el tiempo consumido, que lo podemos saber gracias al sensor *sensores.tiempo*, de manera que cuanto más tiempo se consuma, podemos suponer que ha explorado más mapa y sera capaz de encontrar una buena ruta sin tener que recalcular.

Si ***estoyBienSituado == false***:

Si no estamos bien situados, usaremos una matriz auxiliar, que simulará el mapa, como no sabemos la posición de nuestro personaje y el mapa más grande sera de tamaño 100x100, la matriz auxiliar sera de tamaño 200x200, para nunca provocar un acceso ilegal de memoria.

Para movernos por esa matriz auxiliar usaremos dos variables *fil_img* y *col_img* que por defecto estarán en la posición 99x99, es decir, el centro del mapa.

La táctica a seguir, es ir almacenando en esa matriz auxiliar el numero de instantes que pasamos en una casilla, y de esa forma, evitar casillas en las que ya hemos estado.

Para empezar, actualizamos las posiciones relativas ( *fil_img* y *col_img*) dependiendo de la acción realizada anteriormente.

Cada instante que pasamos en la posición *[fil_img][col_img]* sumamos 1 a su valor en la matriz auxiliar con *mapaImg[fil_img][col_img]++;*

Ahora, de nuevo, tenemos dos posibles comportamientos:

1. Hay plan: Se ha encontrado un PK en el campo de visión e intentamos ir a por el
2. No hay plan: Estamos buscando el PK

Si ***hayPlan == false***:

Si no encontramos el PK, el primer paso es buscarlo en el campo de visión iterando sobre *sensores.terreno*

En caso de no encontrar el PK, comprobamos si lo que tenemos enfrente es un obstaculo, para asignarle un valor alto, y evitar pasar por ese punto, después, cogemos el coste de las casillas adyacentes de la matriz auxiliar, y decidimos cual tiene el menor valor, y por tanto, ha de ser explorada.

Si **posK != -1** quiere decir que hay un PK en el campo de visión, por lo que creamos un plan para llegar hasta el calculando el numero de avances y si tiene que girar

Por último, comprobamos si existe un plan para llegar al PK, y lo ejecutamos.

Con esto terminamos la definición del método *think*.

## 3.1. Comportamiento 4: Reto - Algoritmo de búsqueda A*

El algoritmo A* se trata de una modificación de la búsqueda de Coste Uniforme, en la que además de tener en cuenta el coste de pasar de un estado a otro, también tiene en cuenta una heurística que nos dará una estimación sobre si un nodo esta cerca del objetivo.

Para implementar este algoritmo he decidido usar como heurística la distancia manhattan entre un nodo y el objetivo.

La distancia manhattan sera: |nodo.fila - objetivo.fila| + |nodo.columna - objetivo.columna|

Para que una heurística sea admisible, siendo h(n) la estimación del coste para alcanzar el objetivo desde n, y C(n) el coste mínimo real para alcanzar el objetivo desde n, $h(n) \leq C(n)$.

Vemos entonces que usando la distancia manhattan esto se cumple, ya que la distancia manhattan estima que el coste entre dos casillas adyacentes es 1, y el mínimo coste en el que nos podemos mover de una casilla a otra, también es de 1.

En este caso ordenaremos la lista de abiertos por la suma del coste y la heurística, además de que en Coste Uniforme, si el nodo ya habia sido explorado, esa era la mejor opción, mientras que en A*, tenemos que volver a explorar ese nodo si tiene menos coste.