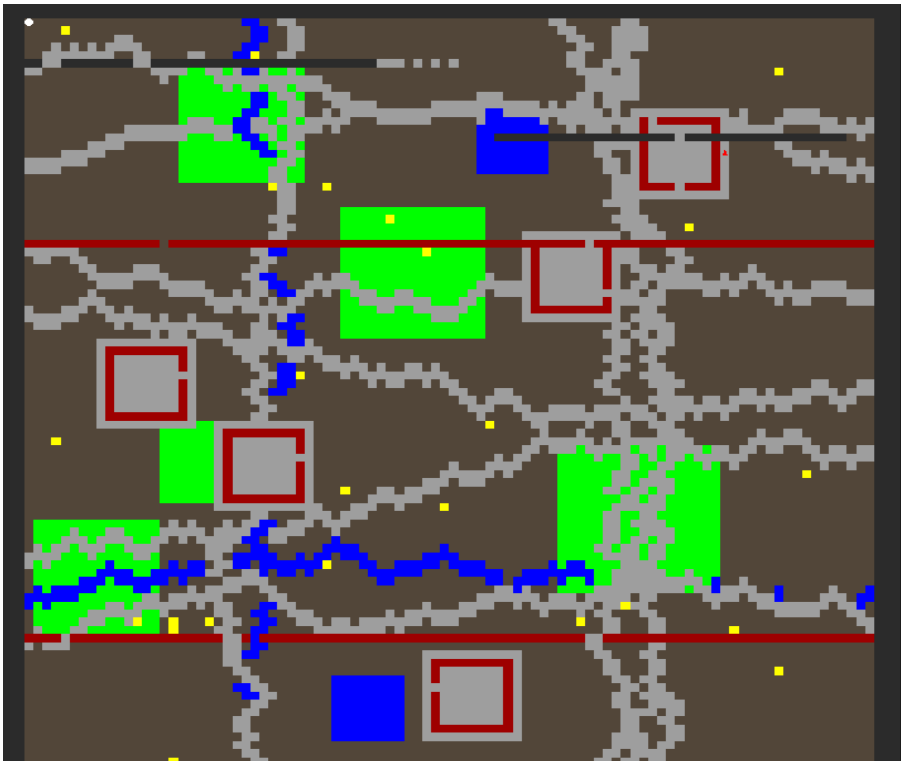
Alba Casillas Rodríguez
2ºA. Grupo de prácticas: A1
Curso 2018/2019

LOS EXTRAÑOS MUNDOS DE BELKAN



Comenzaré explicando el funcionamiento del método:

Action ComportamientoJugador::think(Sensores sensores);

implementado basándome en el código proporcionado por el tutorial, al cual le he hecho ciertas modificaciones para su funcionamiento en ambos niveles.

Inicialmente, compruebo si mi jugador conoce el mapa (deliberativo) o no (reactivo); para ello, utilizo la variable "sensores.mensajeF", ya que **mensajeF** devuelve la fila en la que se encuentra el jugador y **mensajeC** la columna; de forma que solo recibirá el valor "-1" cuando no conozca su posición en el mapa, es decir, en el reto.

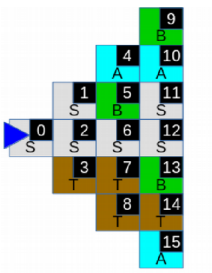
Inicialmente siempre comienzo en modo reactivo (utilizando una variable **booleana "reactivo"** en la clase ComportamientoJugador, inicializada en los constructores a "false"). Unicamente si entra en la comprobación inicial (de forma que el agente sabría su posición puesto que mensajeF sería distinto de -1), modifico el valor de reactivo a false.

Tras esto, como los sensores no nos devuelven la **orientación**, para saber cual ha sido el efecto de nuestra última acción, debemos actualizarla nosotros usando la variable "brújula", según la última acción realizada (usando un switch).

A continuación, he programado un método cual solamente será llamado si nos encontramos en el segundo nivel (en el código se correspondería a sensores.nivel == 4) y si no estamos en modo reactivo. El propósito de esta función es pintar el mapa una vez que el agente ha encontrado una casilla amarilla y se dirige al objetivo, descubriendo así que es lo que ve al agente a cada paso.

void ComportamientoJugador::PintaMapa(Sensores sensores);

El propósito de este método es, teniendo en cuenta los sensores y la orientación del agente, dale valor al tipo de terreno que se corresponde con los sensores. Para ayudarme a hacer este método, he tomado como referencia una de las imágenes proporcionadas en el seminario explicativo:



Si el jugador no conoce el mapa (reactivo == true), debe moverse por el mapa hasta encontrar una **casilla amarilla**. Para ello, me baso en un **contador** y **comportamiento aleatorio** (usando la función **rand()**) del jugador. Mediante el contador controlo que el agente avance 5 veces seguidas mientras no haya obstáculos, y a la quinta vez realiza un giro (a la izquierda o a la derecha) que se decide mediante un número aleatorio de entre 8 valores (los pares giran a la izquierda y los impares a la derecha).

A su vez, he implementado en la clase comportamientoJugador un método que calcule un plan para que el agente se acerque a una casilla amarilla si la ve con uno de los sensores; puesto que

de lo contrario, el agente comienza a dar vueltas por el mapa perdiendo mucho tiempo en busca de una casilla pK, saltándose algunas más próximas a su posición.

En el método, cuya cabecera es:

bool ComportamientoJugador::buscaPK(int posicion, list<Action> & plan);

creo un plan usando los tres movimientos disponibles del agente según el sensor con el que haya identificado a la casilla amarilla, por eso mismo paso un entero "posicion", el cual se corresponde al sensor usado.

También, si la variable "hayPlan" tiene valor false, se calcula un camino hasta el destino mediante uno de los algoritmos proporcionados en los niveles. Este plan se ejecutará cogiendo el primer elemento de una **lista de "Action"** (acciones) que conforman el plan.

Una vez que haya plan y este sea mayor que cero, el valor de la siguiente acción (variable que devuelve nuestro método think) será el primer valor de la lista que conforma el plan a ejecutar, destacando aquí que si en el camino el agente se choca con un obstáculo u aldeano, gire, poniendo también la variable "hayPlan" a false, para que pueda **recalcular** uno nuevo.

Por último, igualamos a una variable "**ultimaAccion**" el valor de la variable "**sigAcción**"; lo cual es necesario para conocer la orientación del personaje.

Nivel 1

Para superar este nivel, donde el jugador conoce el mapa del mundo donde se encuentra, se deben implementar 3 algoritmos: profundidad, anchura y costo uniforme.

Procederé a explicar cómo he resuelto la búsqueda por anchura y costo uniforme ya que la búsqueda por profundidad ya estaba implementada en los archivos proporcionados en clase.

Búsqueda por anchura

En esta búsqueda, se expande primero el nodo raíz, después sus sucesores… así sucesivamente; expandiéndose todos los nodos de una misma profundidad antes de expandirse a los nodos del siguiente nivel. Por eso mismo utilizamos una **cola (FIFO)**, de manera que esta pone a todos los nuevos sucesores generados (girar a la derecha, girar a la izquierda y avanzar) al final de la cola mientras no estén repetidos o sean un obstáculo por el que no se puede pasar, lo que significa que los nodos superficiales se expanden antes que los más profundos.

Búsqueda por costo uniforme

La gran diferencia con la búsqueda en anchura y por profundidad es que ahora tenemos en cuenta el coste de las casillas; por ello queremos conseguir el camino menos costoso hasta el objetivo. Para ello, he decidido modificar el **struct "nodo"**, el cual ya se encontraba proporcionado en el código de programa, para **añadirle un valor entero "coste"**.

Para calcular el coste de un nodo, he usado una función llamada **calculaCoste**, la cual hace uso de un **char "terreno"**, el cual toma el valor del tipo de terreno devuelto por **mapaResultado[nodo.st.fila][nodo.st.columna]** y que mediante un switch, da valor a una variable local coste, la cual será el resultado a obtener. A esta función la llamo al crear el nodo current, y al

generar el hijo para avanzar; puesto que el coste de girar es uno; por lo cual solo debo de hacer nodoHijo.coste++.

Para conseguir un buen funcionamiento del algoritmo, el coste de los sucesores se le suman al del padre, ya que en vez de expandir el nodo más superficial, se expande el nodo n con el **camino de costo más pequeño**. Los costes de los nodos se insertan en una **cola con prioridad**.

La cola con prioridad:

priority_queue<nodo, vector<nodo>, ComparaCoste> prioridad;

la he implementado haciendo uso de un **functor**:

```
struct ComparaCoste{
  bool operator()(const nodo & nodo1, const nodo & nodo2)const{
    return nodo1.coste > nodo2.coste;
  }
};
```

Nivel 2

En general, la dificultad del nivel se encuentra esencialmente en la función “think”, explicada con anterioridad. Sin embargo, y aunque se pudiese reutilizar alguno de los algoritmos de búsqueda anteriormente implementados, he resuelto este nivel con el **algoritmo A***.

Esta búsqueda evalúa los nodos combinando g(n), el **coste** para alcanzar el nodo, y h(n), el c**oste estimado** del camino más barato desde n hasta el nodo objetivo.

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

Para que nuestro algoritmo A* sea óptimo, he utilizado una **heurística admisible** explicada en clase, la **distancia de Manhattan**, cuyo valor es el valor absoluto de la distancia entre el nodo y el objetivo, contando únicamente las distancias horizontales y verticales (ya que no podemos movernos en diagonal).

```
int calcularDistanciaManhattan(const estado & origen, const estado & destino){
   return abs(origen.fila - destino.fila) + abs(origen.columna - destino.columna);
}
```

Por tanto, en mi struct nodo he añadido otra **variable entera “prioridad”**, valiendo la suma del coste del nodo y el valor proporcionado por la heurística.

Los valores de los nodos, al igual que en el coste uniforme, son guardados en una cola con prioridad que hace uso de un **fuctor** para comparar las prioridades de los nodos.

```
struct ComparaManhattan{
  bool operator()(const nodo & nodo1, const nodo & nodo2)const{
    return nodo1.prioridad > nodo2.prioridad;
  }
};
```