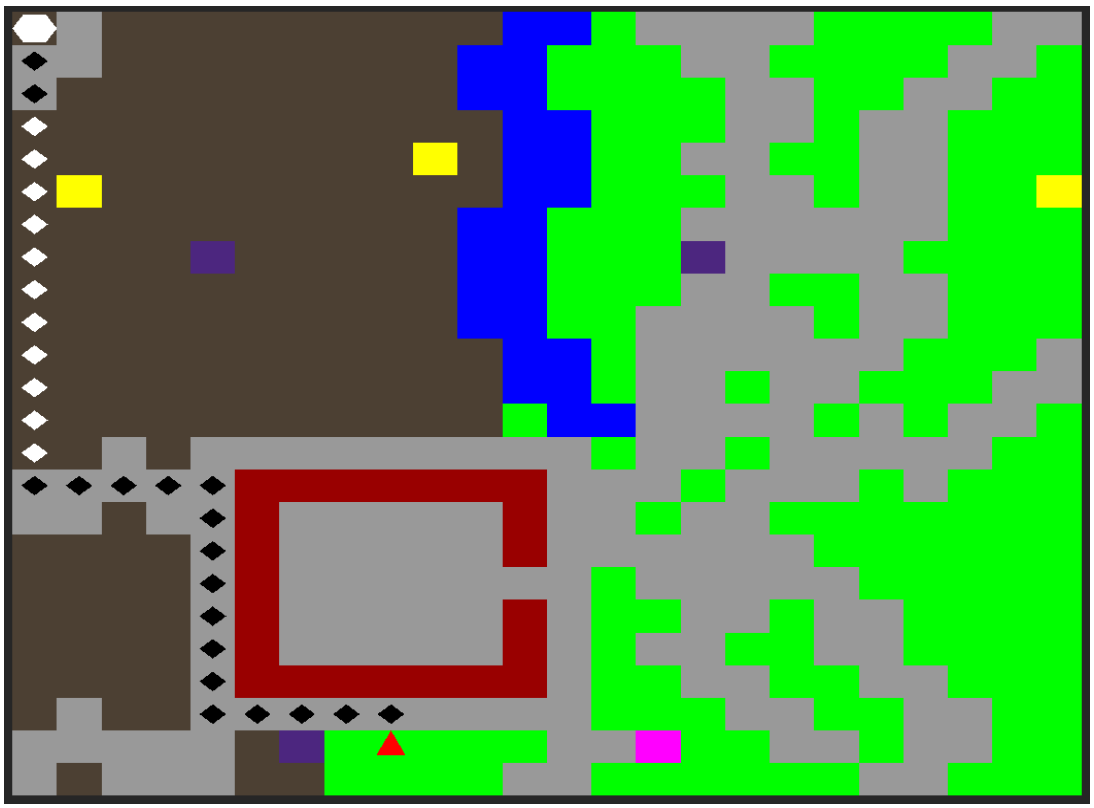
Memoria Práctica 2 Inteligencia Artificial: Agentes reactivos/deliberativos

David Jesús Ruiz de Valdivia Torres 2ºC



ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

UNIVERSIDAD DE GRANADA

Nivel 1:

En este nivel se encontraba la implementación de la búsqueda por profundidad ya hecha por lo que no nos centraremos en ella, en su lugar hablaremos de la búsqueda por anchura y la búsqueda de costo uniforme.

Vamos a empezar con la búsqueda en anchura.
Este algoritmo calcula el camino con menos movimientos. Su implementación fue muy sencilla porque solo hizo falta copiar el código de la búsqueda por profundidad y cambiar la pila que usaba para la lista de abiertos por una cola.

En profundidad:

```
bool ComportamientoJugador::pathFinding_Profundidad(const estado &origen, const estado &destino, list<Action> &plan)
{
    //Borro la lista
    cout << "Calculando plan\n";
    plan.clear();
    set<estado, ComparaEstados> generados; // Lista de Cerrados
    stack<nodo> pila;                      // Lista de Abiertos
```

En anchura:

```
bool ComportamientoJugador::pathFinding_Anchura(const estado &origen, const estado &destino, list<Action> &plan)
{
    //Borro la lista
    cout << "Calculando plan\n";
    plan.clear();
    set<estado, ComparaEstados> generados; // Lista de Cerrados
    queue<nodo> cola;                      // Lista de Abiertos
```

El cambio es solo ese porque profundidad trata los nodos expandidos de abiertos con una política LIFO y el de anchura, con FIFO.

Seguiremos con la búsqueda por costo uniforme, este método permite calcular el camino con menos coste en batería.
Nuevamente, esta búsqueda es muy similar a las anteriores: se expanden tres nodos hijos (hacia izquierda, derecha y hacia delante), y si no están ya en la lista de cerrados, se meten en la lista de abiertos para que posteriormente uno sea añadido a la lista de cerrados y sea expandido (qué nodo depende del método de búsqueda).

La diferencia aquí es que en las búsquedas de costo uniforme, la lista de abiertos debe estar ordenada según lo que queramos optimizar, en nuestro caso la batería.

Es entonces que usamos como estructura de datos para la lista de abiertos una cola con prioridad cuya función de ordenación es ComparaCoste.

```
bool ComportamientoJugador::pathFinding_PorCosto(const estado &origen, const estado &destino, list<Action> &plan)
{
    //Borro la lista
    cout << "Calculando plan\n";
    plan.clear();
    set<estado, ComparaEstados> generados;              // Lista de Cerrados
    priority_queue<nodo, vector<nodo>, ComparaCoste> q2; // Lista de Abiertos
```

Este functor ordena según los costes de cada casilla. Este coste se obtiene con devuelveCoste().

Esta función comprueba el tipo de casilla y devuelve el coste asociado según las reglas establecidas. Para esta parte se tuvieron que implementar nuevas variables estado, tieneBikini y tieneZapatillas. Esto son dos booleanos que indican si la mejora en cuestión ha sido obtenida o no.
Antes de insertar un nuevo nodo, se busca si este está repetido para eliminarlo y agilizar así la ejecución.

Con los nodos entonces ordenados mediante lo anterior, se va obteniendo al final del proceso un camino hacia el objetivo con el mínimo gasto de batería.

Nivel 2

En este segundo nivel se implementa el reto en donde se trata de encontrar el máximo número de objetivos en un mapa desconocido para el agente.

Para ello se ha usado un algoritmo basado en el A*. En dicho algoritmo se define la función heurística siguiente: f(n) de un nodo = G(n) + H(n) de ese mismo nodo n.
G(n) es el coste de haber llegado a esa casilla y H(n) es la distancia Manhattan del nodo al objetivo. La distancia Manhattan es la distancia absoluta entre dos puntos. Esta distancia Manhattan la devolveremos con devuelveHeuristica().

Nuestro nuevo método de pathFinding tendrá una cola con prioridad donde incluirá elementos ordenándolos mediante su valor F con ComparaDistancias.

```
bool ComportamientoJugador::pathFinding_Reto(const estado &origen, const estado &destino, list<Action> &plan)
{
    //Borro la lista
    cout << "Calculando plan\n";
    plan.clear();
    set<estado, ComparaEstados> generados;                    // Lista de Cerrados
    priority_queue<nodo, vector<nodo>, ComparaDistancias> q2; // Lista de Abiertos
```

Se calcularán G y H de los nodos hijos al crearlos antes de comprobar si ya están dentro de los cerrados.

```
// Generar descendiente de girar a la derecha
nodo hijoTurnR = current;
hijoTurnR.st.orientacion = (hijoTurnR.st.orientacion + 1) % 4;
hijoTurnR.st.tieneBikini = current.st.tieneBikini;
hijoTurnR.st.tieneZapatillas = current.st.tieneZapatillas;
hijoTurnR.H = devuelveHeuristica(hijoTurnR);
hijoTurnR.G = devuelveCoste(hijoTurnR.st) + costo_padre;

if (generados.find(hijoTurnR.st) == generados.end())
{
    hijoTurnR.secuencia.push_back(actTURN_R);
    q2.push(hijoTurnR);
}
```

Para ello, añadiremos G y H como nuevos atributos de los nodos, así como un puntero a su padre y el coste de haber llegado allí. El resto del procesamiento de los hijos es igual.

Nuestro agente empezará entonces a revelar mapa mediante sus sensores con descubrirMapa() que devolverá un booleano que nos informará acerca de si estamos descubriendo mapa que era desconocido antes o no (aparte de actualizar el mapa).

Si estamos descubriendo mapa nuevo, se recalcula un plan nuevo para ver si hay alguna alternativa mejor con el nuevo conocimiento, esto permite al agente adaptarse y calcular la mejor ruta en términos de batería en cada momento.

Esto nos lleva al método think. Este método no ha sido mencionado en el nivel 1 porque además de ser idéntico para las 3 búsquedas, venía implementado ya. En el nivel 2 se tuvo que realizar uno completamente nuevo, para separar el comportamiento del nivel 1 y el 2, se ha usado un if que separa dichas etapas.

Nuestro método think comienza asignando a un valor booleano el resultado de descubrirMapa(). Si ese valor booleano o nuevoObjetivo() (informa de si el objetivo ha cambiado) es true, entonces se actualiza el valor de destino con la información de los sensores y se indica que hay que rehacer el plan (estoy tiene sentido porque o bien hemos descubierto mapa nuevo y puede haber una ruta más óptima o bien hay que encontrar otro objetivo). Se rehace el plan con el

pathFinding descrito más arriba si es necesario (hayplan = false) y se ejecuta el nuevo plan creado mientras haya uno y este no esté vacío de acciones.

```
if (sensores.nivel == 4)
{

    Action accion = actIDLE;
    bool mapaNuevo = descubrirMapa(sensores);
    unsigned char contenidoCasilla;

    if (mapaNuevo or nuevoObjetivo(sensores))
    {
        hayplan = false;
        destino.fila = sensores.destinoF;
        destino.columna = sensores.destinoC;
    }

    if (!hayplan)
    {
        actualizaPlan(sensores);
    }

    if (hayplan and plan.size() > 0)
    {

        if (sensores.superficie[2] == 'a' || (sensores.terreno[0] == 'X' and sensores.bateria < 2250))
        {
            accion = actIDLE;
        }
        else
        {
            accion = plan.front();
            plan.erase(plan.begin());
        }
    }

    return accion;
}
```

Finalmente, he recopilado el resultado de las ejecuciones en los distintos mapas con los datos proporcionados tanto en la versión visual como en la textual, aquí los resultados:

Versión textual	
Mapa	N.º objetivos
mapa30.map	70
mapa50.map	17
mapa75.map	18
mapa100.map	8
islas.map	4
medieval.map	12

Versión visual	
Mapa	N.º objetivos
mapa30.map	44
mapa50.map	64
mapa75.map	24
mapa100.map	8
islas.map	1
medieval.map	12