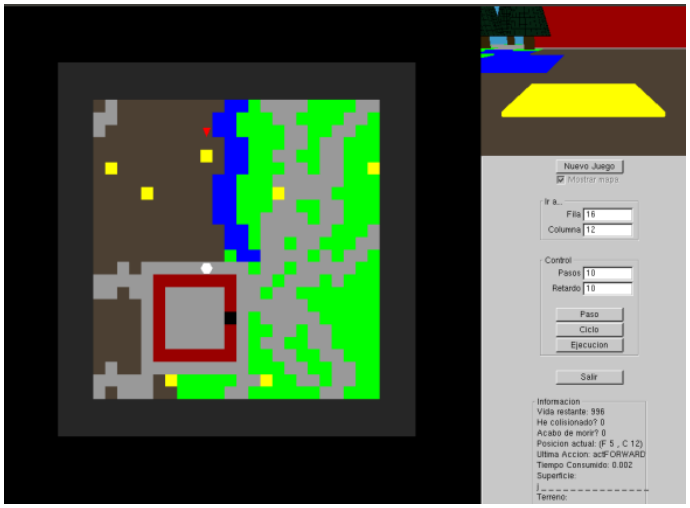
Universidad de Granada

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIONES

BELKAN



Melguizo Marcos, Iñaki

1 Nivel 1: Agente deliberativo

En el método $think$ de la clase jugador.cpp he tenido que crear una variable entera “contador_plan” para detectar cuando se ha llegado a un punto amarillo ya que cuando se llega a un punto amarillo se crea un nuevo plan vaciando la lista de acciones

```
bool hay_plan = pathFinding (sensores.nivel, actual, destino, plan);
if (contador_plan!=plan.size()){
    contador_aux=0;
    contador_plan=plan.size();

}
if (hay_plan  && contador_aux<plan.size()){
    list<Action> plan_aux=plan;
    for (int i=0; i < contador_aux; i++){
        plan_aux.pop_front();
    }
```

- Búsqueda en **proundidad**: Para el algoritmo de búsqueda he utilizado una **cola** en la que para cada nodo que extraía de la cola lo exploraba y, si ninguno de los estados resultantes de la exploración era el destino, volvía a meter en la cola el Hijo Izquierda, Hijo Derecha e Hijo Forward en este orden de manera que siempre que pudiera avanzar avanzaría y en caso de no poder giraría a la derecha y como última opción a la izquierda.

```
stack<nodo> pila;

nodo current;
current.st = origen;
current.secuencia.empty();

pila.push(current);
```

  También, cada estado explorado se introducía en un set, de manera que no se explorara dos veces el mismo estado.

```
set<estado,ComparaEstados> generados;
```

- Búsqueda en **anchura**: Para este algoritmo he creado una **lista de listas de nodos**, en la que cada lista de nodos se corresponde con los nodos que hay en cada nivel de profundidad. Mientras el nodo destino no ha sido encontrado, para cada nodo que hay en la lista, recorriendo dichos nodos mediante un bucle, exploro dichos nodos y veo si alguno de dichos nodos explorados se corresponde con el estado final y si no es así introduzco dichos nodos explorados en la lista (habiendo borrado previamente los nodos padre de la lista). Así consigo la búsqueda en anchura.

```
list<list<nodo>> ll;
list<list<nodo>>::iterator itr;
```

```
list<nodo> listaprimero;
listaprimero.push_back(nodo_aux);
ll.push_front(listaprimero);

itr=ll.begin();


list<nodo> lista_aux;

generados.insert(nodo_aux.st);
```

- Búsqueda por **coste uniforme**: Para este algoritmo he creado una lista de *costo* (nodo + coste hasta llegar a ese nodo).

```
struct costo{
    nodo n;
    int cost;
};
```

Con una búsqueda a través de la lista, buscaba el costo de menor *cost* y exploraba dicho nodo y de esa forma cuando lleguemos al nodo destino, lo habremos hecho por el recorrido de menor costo posible.

```
minimo_coste=1000;
contador=0;
std::list<costo>::iterator it;
std::list<costo>::iterator it2;
for (it=l.begin();it!=l.end(); ++it){
    if ((*it).cost<minimo_coste){
        minimo_coste=(*it).cost;
        it2=it;
        //contador_minimo=contador;
    }
    //contador++;
}
```

2 Nivel 2: Agente reactivo/deliberativo complejo

Para el algoritmo del nivel 2 he tenido que atualizar la matriz *mapaResultado* en función del valor de la brújula.

```
switch (brujula){

case 0:

  mapaResultado[fil][col] = sensores.terreno[0];
  mapaResultado[fil-1][col-1] = sensores.terreno[1];
  mapaResultado[fil-1][col] =sensores.terreno[2];
  mapaResultado[fil-1][col+1] = sensores.terreno[3];
  mapaResultado[fil-2][col-2] = sensores.terreno[4];
  mapaResultado[fil-2][col-1] = sensores.terreno[5];
  mapaResultado[fil-2][col] = sensores.terreno[6];
  mapaResultado[fil-2][col+1] = sensores.terreno[7];
  mapaResultado[fil-2][col+2] = sensores.terreno[8];
  mapaResultado[fil-3][col-3] = sensores.terreno[9];
  mapaResultado[fil-3][col-2] = sensores.terreno[10];
  mapaResultado[fil-3][col-1] = sensores.terreno[11];
  mapaResultado[fil-3][col] = sensores.terreno[12];
  mapaResultado[fil-3][col+1] = sensores.terreno[13];
  mapaResultado[fil-3][col+2] = sensores.terreno[14];
  mapaResultado[fil-3][col+3] = sensores.terreno[15];
  break;
```

Además, he tenido que diferenciar entre los siguientes casos:

- Encontrarse que no puede avanzar o hay un aldeano y que todavia no haya encontrado el punto amarillo, en cuyo caso girará a la izquierda.
- Si no se da lo anterior y no se ha encontrado el punto amarillo: Avanzará una casilla.
- Si no se da ninguna de las anteriores será porque habrá encontrado el punto amarillo. En este caso, se actualizará la matriz *mapaResultado*, las variables *fil, col,* etc. Además, se fijará la variable *amarillo* a true, que hará que a partir de ese momento en cada iteración llame a la función $pathFinding$, siendo el primer argumento de esta función el valor 3, que provocará que se realice la búsqueda por coste uniforme (siendo el coste de las casillas desconocidas por el momento igual a 1).

```
bool hay_plan=pathFinding(3, actual, destino, plan);
```

```
accion=plan_aux.front();
switch(accion){

  case actTURN_R:brujula=(brujula+1)%4;
  break;
  case actTURN_L:brujula=(brujula+3)%4;
  break;
  case actFORWARD:
  if (sensores.terreno[2]=='P' or sensores.terreno[2]=='M' or
    sensores.terreno[2]=='D' or sensores.superficie[2]=='a'){
    accion=actTURN_R;
    brujula=(brujula+1)%4;
  }
  else{
    if (brujula==0){
      fil--;
    }
    else if (brujula==1){
      col++;
    }
    else if (brujula==2){
      fil++;
    }
    else
      col--;
  }
}
actual.fila=fil;
actual.columna=col;
actual.orientacion=brujula;
```