UNIVERSIDAD
DE GRANADA

2ºC

Grado en Ingeniería Informática

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

Autor:
Antonio David Villegas Yeguas

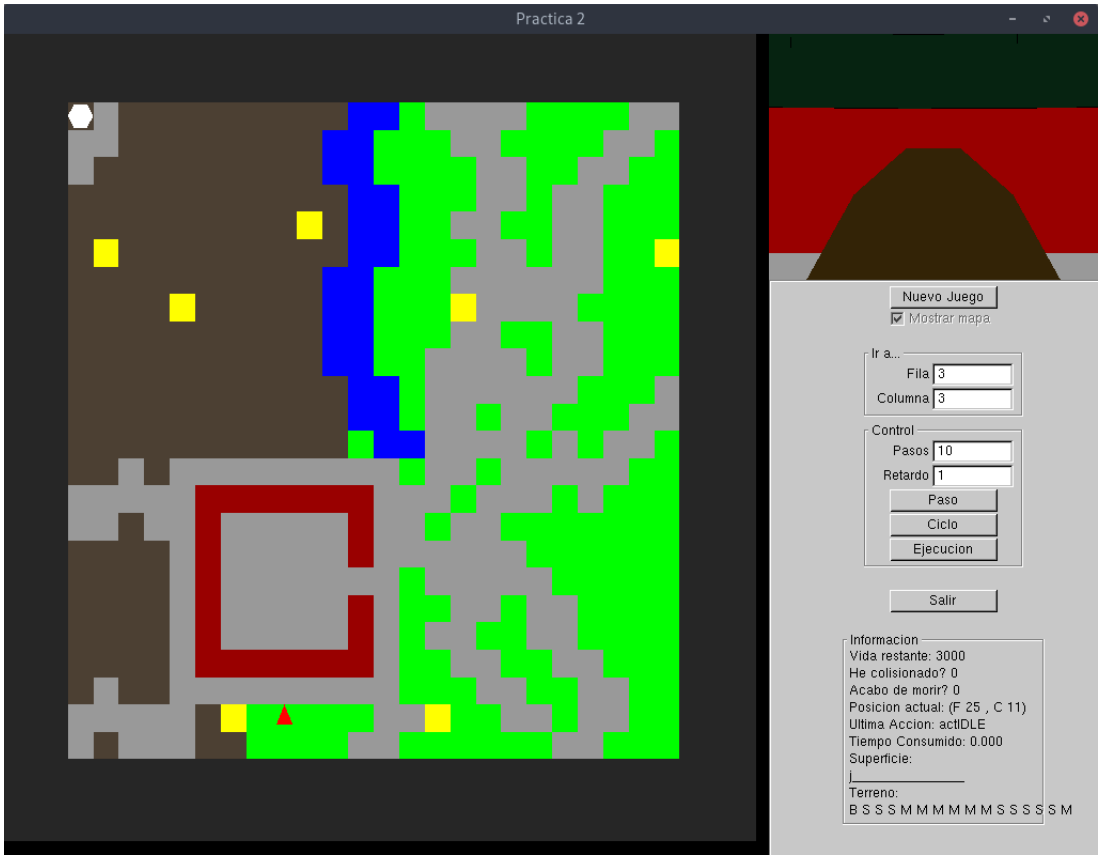
Asignatura:
Inteligencia Artificial

4 de mayo de 2019

Índice

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".



El videojuego se basa en un personaje (flecha roja) que aparece en un mundo, y ha de encontrar objetivo. El mundo tiene distintos tipos de terreno, sobre los que nuestro personaje tendrá distintos problemas para atravesarlos, como por ejemplo, no podrá atravesar muros (casillas rojas) , ni caminar por precipicios (casillas negras), sobre el agua (casillas azules) necesitará más tiempo para pasar que por bosque (casillas verdes), mientras que por tierra (casillas marrones) necesitará menos tiempo, mientras que por los senderos(casillas grises) será por donde menos tiempo necesite.

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:

```
//nivel 1

if (sensores.mensajeF != -1){
   fil = sensores.mensajeF;
   col = sensores.mensajeC;
   ultimaAccion = actIDLE;
}
```

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.

```
//Actualizar el efecto de la ultima accion
switch (ultimaAccion){
   case actTURN_R: brujula = (brujula+1)%4; break;
   case actTURN_L: brujula = (brujula+3)%4; break;
   case actFORWARD:
      switch (brujula){
         case 0: fil--; break;
         case 1: col++; break;
         case 2: fil++; break;
         case 3: col--; break;
      }
   break;
}
```

Si cambia el destino, volvemos a calcular el plan.

```
if (sensores.destinoF != destino.fila or sensores.destinoC != destino.columna){
   destino.fila = sensores.destinoF;
   destino.columna = sensores.destinoC;
   hayPlan = false;
}
```

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;.

```
if (!hayPlan){
   actual.fila = fil;
   actual.columna = col;
   actual.orientacion = brujula;
   hayPlan =  pathFinding(sensores.nivel, actual, destino, plan);
}
```

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

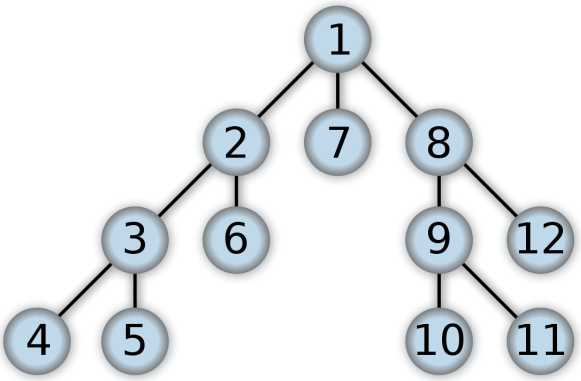
```
// Si tenemos un plan, y es viable (la longitud es mayor que 0):
if (hayPlan and plan.size()>0){

   sigAccion = actIDLE;

   // Comprobamos si la casilla de delante es un obstaculo, y si lo
   // es, establecemos hayPlan a falso, y recalculamos el plan
   if (EsObstaculo(sensores.terreno[2]) and plan.front() == actFORWARD ){
      hayPlan = false;
   } else if (sensores.superficie[2] != 'a' || plan.front() != actFORWARD){
      //en caso de que no sea un obstaculo o la accion sea un giro
      // si no hay un aldeano delante, o si es un giro realizamos el plan
      sigAccion = plan.front();
      plan.erase(plan.begin());
   }
   // en caso de que tengamos un aldeano delante y la accion del plan sea
   // avanzar, simplemente esperamos a que el aldeano se mueva

   //Esta situacion en principio no puede darse en el nivel 1 ya que
   // no hay aldeanos, aun asi, he decidido mantener esta comprobación
   // de cara a algún cambio imprevisto

}
else{
   //en caso de no encontrar plan, realizamos un comportamiento reactivo,
   // comprobamos si lo que tenemos delante es un problema a
   // la hora de moverse hacia delante
   if (sensores.terreno[2] == 'P' || sensores.terreno[2] == 'M' ||
       sensores.terreno[2] == 'D' || sensores.superficie[2] == 'a'){
      //si lo es, giramos
      sigAccion = actTURN_R;
   }
   else{
      //si no, avanzamos
      sigAccion = actFORWARD;
   }


}
```

Por último, ya estemos en el nivel 1 o 2, devolvemos la ultima acción

```
//Recordar ultima accion
ultimaAccion = sigAccion;
return sigAccion;
```

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.



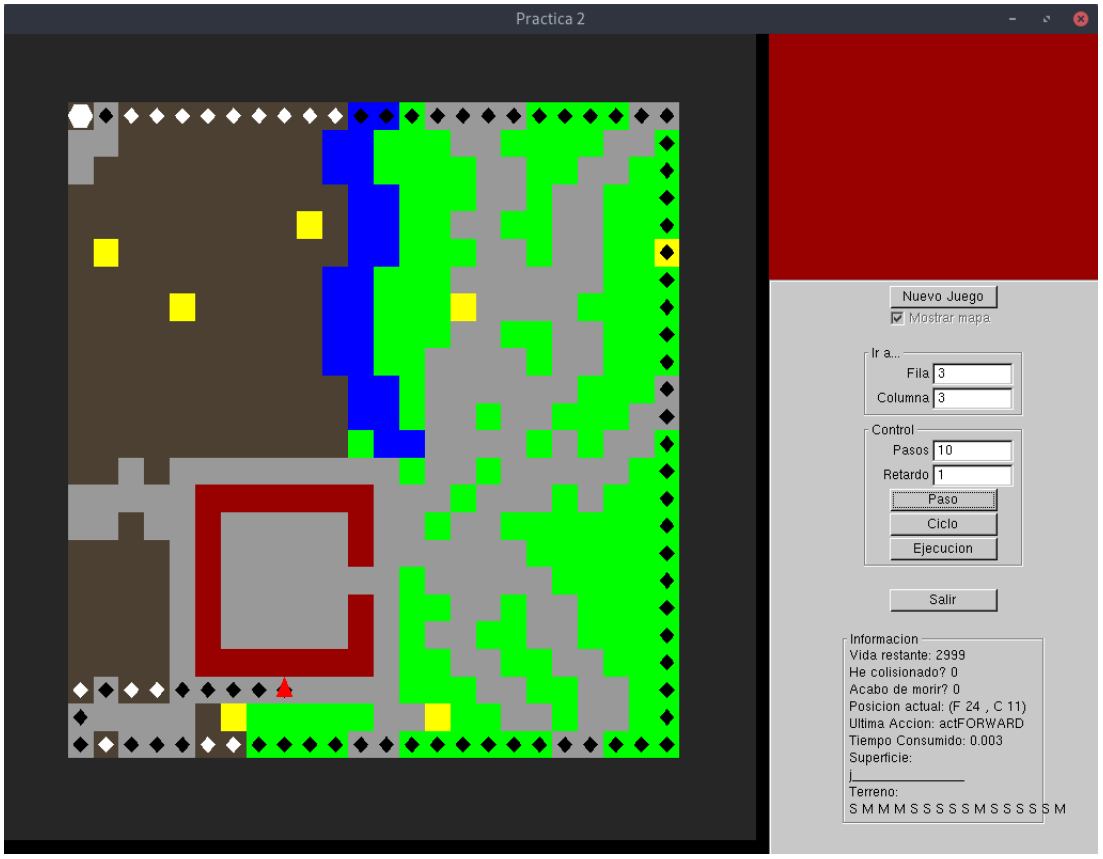
Esta implementación la podemos conseguir usando una pila, ya que si siempre abrimos los nodos de la misma forma, en la cima de la pila siempre tendremos el resultado de explorar por un camino concreto, y cuando exploremos totalmente ese camino, en la cima quedara el último nodo que se dejó sin explorar,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

```
// Implementación de la búsqueda en profundidad.
// Entran los puntos origen y destino y devuelve la
// secuencia de acciones en plan, una lista de acciones.
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

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

   pila.push(current);
```

```
 while (!pila.empty() and (current.st.fila!=destino.fila or \
                          current.st.columna != destino.columna)){

   pila.pop();
   generados.insert(current.st);

   // 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);
   }

   // Generar descendiente de girar a la izquierda
   nodo hijoTurnL = current;
   hijoTurnL.st.orientacion = (hijoTurnL.st.orientacion+3)%4;
   if (generados.find(hijoTurnL.st) == generados.end()){
      hijoTurnL.secuencia.push_back(actTURN_L);
      pila.push(hijoTurnL);
   }

   // Generar descendiente de avanzar
   nodo hijoForward = current;
   if (!HayObstaculoDelante(hijoForward.st)){
      if (generados.find(hijoForward.st) == generados.end()){
         hijoForward.secuencia.push_back(actFORWARD);
         pila.push(hijoForward);
      }
   }

   // Tomo el siguiente valor de la pila
   if (!pila.empty()){
      current = pila.top();
   }
}
```

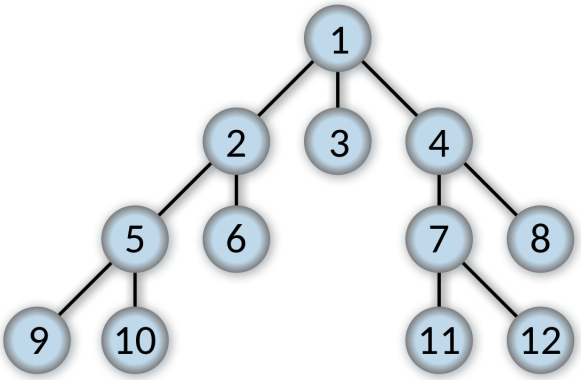
```
    cout << "Terminada la busqueda\n";


   if (current.st.fila == destino.fila and current.st.columna == destino.columna){
      cout << "Cargando el plan\n";
      plan = current.secuencia;
      cout << "Longitud del plan: " << plan.size() << endl;
      PintaPlan(plan);
      // ver el plan en el mapa
      VisualizaPlan(origen, plan);
      return true;
   }
   else {
      cout << "No encontrado plan\n";
   }


   return false;
}
```

Un ejemplo de la ejecución del algoritmo:



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.



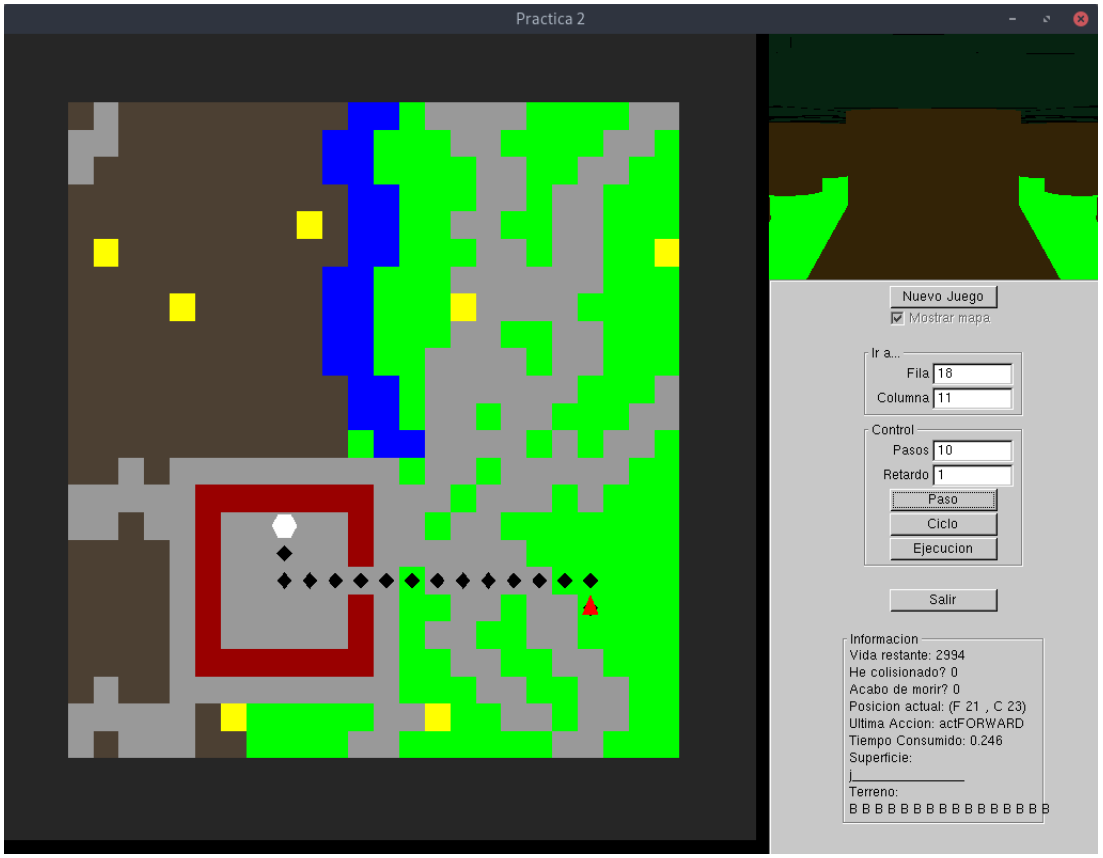
Esto lo podemos conseguir usando una cola, ya que exploramos los nodos en el orden que los introducimos.

```
bool ComportamientoJugador::pathFinding_Anchura(const estado &origen, \
                                                const estado &destino,\
                                                list<Action> &plan) {
   //Borro la lista
   cout << "Calculando plan\n";
   plan.clear();

   // usamos una cola
   queue<nodo> cola_nodos;
   set<estado,ComparaEstados> generados;

   // establecemos el primer nodo al origen, y lo inssertamos en la cola
   nodo current;
     current.st = origen;
     current.secuencia.clear();


   cola_nodos.push(current);
```

```
// mientras queden nodos por explirar y no lleguemos al destino
while (!cola_nodos.empty() and (current.st.fila!=destino.fila or \
                                current.st.columna != destino.columna)){

   // abrimos el primer nodo, y lo metemos en cerrados
   cola_nodos.pop();
   generados.insert(current.st);

   //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);
   }

   //Generamos nodo correspondiente a girar a la izquierda
   nodo leftSon = current;
   leftSon.st.orientacion = (leftSon.st.orientacion+3) % 4;
   if(generados.find(leftSon.st) == generados.end()){
      leftSon.secuencia.push_back(actTURN_L);
      cola_nodos.push(leftSon);
   }

   //Generamos nodo correspondiente a avanzar
   nodo fowardSon = current;
   if(!HayObstaculoDelante(fowardSon.st)){
      if(generados.find(fowardSon.st) == generados.end()){
         fowardSon.secuencia.push_back(actFORWARD);
         cola_nodos.push(fowardSon);
      }
   }

   //Cogemos el primer nodo de la cola
   if (!cola_nodos.empty()){
      current = cola_nodos.front();
   }

}

  cout << "Terminada la busqueda\n";
```

```
    if (current.st.fila == destino.fila and current.st.columna == destino.columna){
        cout << "Cargando el plan\n";
        plan = current.secuencia;
        cout << "Longitud del plan: " << plan.size() << endl;
        PintaPlan(plan);
        // ver el plan en el mapa
        VisualizaPlan(origen, plan);
        return true;
    }
    else {
        cout << "No encontrado plan\n";
    }


    return false;
}
```

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 el siguiente functor:

```
// Functor para ordenar los nodos por coste total
struct OrdenaNodo{
   bool operator()(const nodo &a, const nodo &n) const{
      return a.coste_total < n.coste_total;
   }
};
```

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) {
   return (a.fila == n.fila and a.columna == n.columna and \
           a.orientacion == n.orientacion);
}
```

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){
   for(auto it = abiertos.begin(); it!= abiertos.end(); ++it){
      if(ComparaEstado((*it).st, nodo.st )){
         return it;
      }
   }

   return abiertos.end();
}
```

Con todo esto, el algoritmo de búsqueda quedaría de la siguiente forma:

```
bool ComportamientoJugador::pathFinding_CostoUniforme(const estado &origen, \
                                                      const estado &destino,\
                                                      list<Action> &plan) {
   cout << "Calculando plan\n";
   plan.clear();

   //usamos un multiset en el que ordenamos por coste_total
   multiset<nodo, OrdenaNodo> abiertos;
   set<estado,ComparaEstados> generados;

   // almacenamos el nodo origen, con coste 0
   nodo current;
   current.st = origen;
   current.secuencia.clear();

   current.coste_total = 0;

   abiertos.insert(current);
```

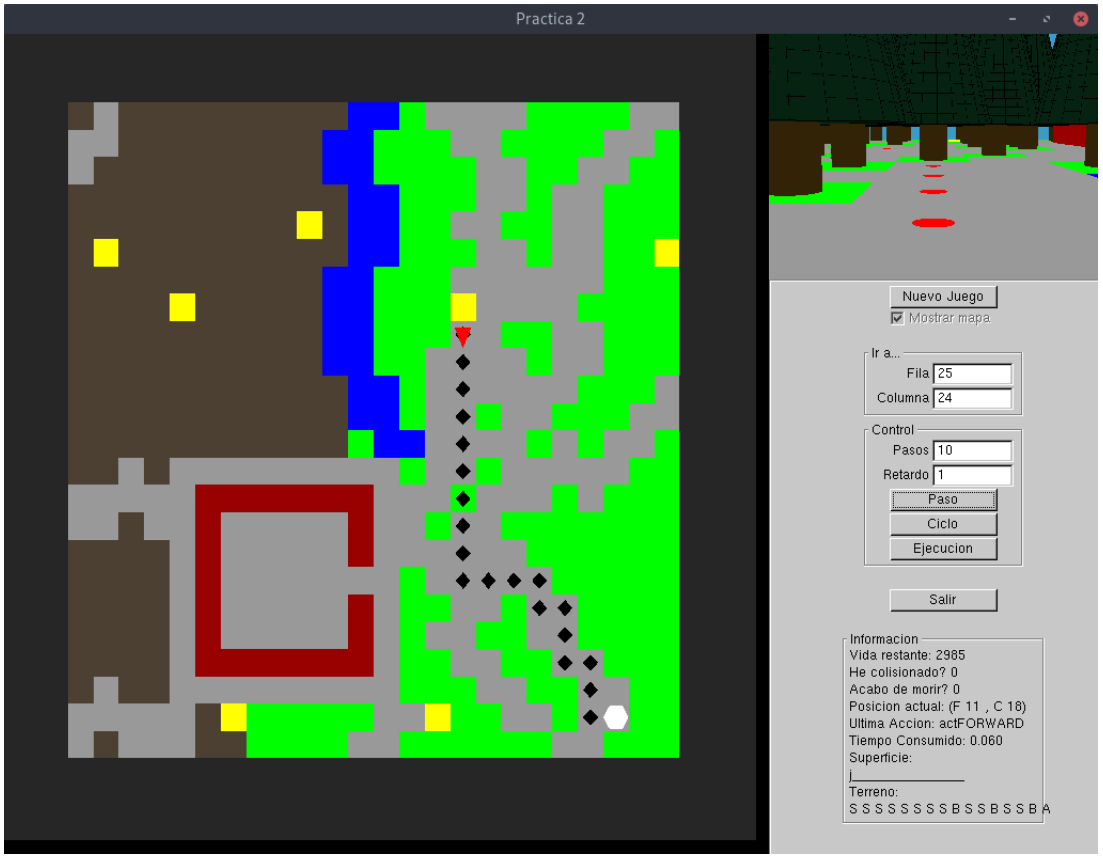
```
// mientras que queden nodos por explorar y no estemos en el destino
while (!abiertos.empty() and (current.st.fila!=destino.fila or \
                              current.st.columna != destino.columna)){

   // abrimos el primer nodo y lo almacenamos en cerrados
   abiertos.erase(abiertos.begin());
   generados.insert(current.st);

   // calculamos el nodo de girar a la derecha, y si no lo hemos explorado,
   // trabajamos con el
   nodo rightSon = current;
   rightSon.st.orientacion = (rightSon.st.orientacion+1) % 4;
   if(generados.find(rightSon.st) == generados.end()){

      //calculamos el coste de ir a ese estado
      rightSon.coste_total += calcularCoste(current.st, rightSon.st );

      // buscamos el nodo en abiertos
      auto it = nodoEncontrado(abiertos, rightSon);

      // si lo encontramos
      if (it != abiertos.end()){
         // nos quedamos con el de mejor coste
         if (rightSon.coste_total < it->coste_total){
            abiertos.erase(it);
            rightSon.secuencia.push_back(actTURN_R);
            abiertos.insert(rightSon);
         }
      } else{
         // si no lo encontramos lo añadimos a abiertos
         rightSon.secuencia.push_back(actTURN_R);
         abiertos.insert(rightSon);
      }

   }
```

```
// hijo resultante de girar a la izquierda
nodo leftSon = current;
leftSon.st.orientacion = (leftSon.st.orientacion+3) % 4;
if(generados.find(leftSon.st) == generados.end()){

   // calculamos el coste de ir a ese estado
   leftSon.coste_total += calcularCoste(current.st, leftSon.st );

   // lo buscamos en  abiertos
   auto it = nodoEncontrado(abiertos, leftSon);

   // si lo encontramos nos quedamos con el mejor de los dos
   if (it != abiertos.end()){
      if (leftSon.coste_total < it->coste_total){
         abiertos.erase(it);
         leftSon.secuencia.push_back(actTURN_L);
         abiertos.insert(leftSon);
      }
   } else{
      // si no lo añadimos
      leftSon.secuencia.push_back(actTURN_L);
      abiertos.insert(leftSon);
   }

}
```

```
    // nodo resultante de avanzar
    nodo fowardSon = current;
    //si no es un obstaculo ni lo hemos abierto, trabajamos con el
    if(!HayObstaculoDelante(fowardSon.st)){
       if(generados.find(fowardSon.st) == generados.end()){

          // calculamos el coste de cambiar de estado
          fowardSon.coste_total += calcularCoste(current.st, fowardSon.st );

          // lo buscamos en abiertos
          auto it = nodoEncontrado(abiertos, fowardSon);

          // si el estado ya esta en abiertos, nos quedamos con el de mejor coste
          if (it != abiertos.end()){
             if (fowardSon.coste_total < it->coste_total){
                abiertos.erase(it);
                fowardSon.secuencia.push_back(actFORWARD);
                abiertos.insert(fowardSon);
             }
          } else{
             //si no esta, lo añadimos abiertos
             fowardSon.secuencia.push_back(actFORWARD);
             abiertos.insert(fowardSon);
          }


       }
    }

    //mientras queden nodos por explorar, seguimos
    if (!abiertos.empty()){
       current = (*abiertos.begin());
    }

}

cout << "Terminada la busqueda\n";
```

```
    if (current.st.fila == destino.fila and current.st.columna == destino.columna){
        cout << "Cargando el plan\n";
        plan = current.secuencia;
        cout << "Longitud del plan: " << plan.size() << endl;
        cout << "Coste del plan " << current.coste_total << endl;
        PintaPlan(plan);
        // ver el plan en el mapa
        VisualizaPlan(origen, plan);
        return true;
    }
    else {
        cout << "No encontrado plan\n";
    }

    return false;
}
```

Un ejemplo de la ejecución del algoritmo:



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:

```
if (sensores.mensajeF != -1){
   fil = sensores.mensajeF;
   col = sensores.mensajeC;
   ultimaAccion = actIDLE;
   estoyBienSituado = true;
}
```

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.

```
//actualizo el estado conforme a la ultima accion
switch (ultimaAccion){
   case actTURN_R: brujula = (brujula+1)%4; break;
   case actTURN_L: brujula = (brujula+3)%4; break;
   case actFORWARD:
      switch (brujula){
         case 0: fil--; break;
         case 1: col++; break;
         case 2: fil++; break;
         case 3: col--; break;
      }
      break;
}

actual.orientacion = brujula;
```

Pintamos la parte del mapa que el personaje es capaz de observar usando *sensores.terreno*:

```
int k = 0;

for(unsigned int i = 0; i < 4; i++){
   for(unsigned int j = 0; j <= i*2; j++ ){
      switch (actual.orientacion) {
         //si estamos mirando al norte
         case 0:
               if (mapaResultado[fil - i][col - i + j] == '?')
                  mapaResultado[fil - i][col - i + j] = sensores.terreno[k];
               k++;
            break;

         //si estamos mirando al este
         case 1:

            if (mapaResultado[fil - i + j][col + i] == '?')
               mapaResultado[fil - i + j][col + i] = sensores.terreno[k];
            k++;
```

```
            break;

         //si estamos mirando al sur
         case 2:

            if (mapaResultado[fil + i][col + i - j] == '?')
               mapaResultado[fil + i][col + i - j] = sensores.terreno[k];
            k++;

            break;

         //si estamos mirando al oeste
         case 3:

            if (mapaResultado[fil + i - j][col - i] == '?')
               mapaResultado[fil + i - j][col - i] = sensores.terreno[k];
            k++;

            break;
      }
   }
}
```

Comprobamos si el destino se ha modificado, y en tal caso, invalidamos el plan

```
// comprobamos si ha cambiado el objetivo y en caso de que se mueva
// recolocamos el destino
if (sensores.destinoF != destino.fila or sensores.destinoC != destino.columna){
   destino.fila = sensores.destinoF;
   destino.columna = sensores.destinoC;
   hayPlan = false;
}
```

Si no hay plan, actualizamos el estado actual, y buscamos un plan.

```
// si no hay plan, actualizamos el estado actual y buscamos el plan
if (!hayPlan){
   actual.fila = fil;
   actual.columna = col;
   actual.orientacion = brujula;
   hayPlan =  pathFinding(sensores.nivel, actual, destino, plan);
```

```
}
```

Si encuentra un plan valido, lo ejecuta, siempre teniendo en cuenta posibles obstáculos

```
// si encuentra el plan, lo ejecutamos
if (hayPlan and plan.size()>0){

   sigAccion = actIDLE;

   // si el plan pasa por un obstaculo, tenemos que recalcular el plan
   // luego no hacemos nada, y establecemos hayPlan a falso
   if (EsObstaculo(sensores.terreno[2]) and plan.front() == actFORWARD ){
      hayPlan = false;
   } else if (sensores.superficie[2] != 'a' || plan.front() != actFORWARD ){
      sigAccion = plan.front();
      plan.erase(plan.begin());
   }

}
```

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.

```
// realizamos una accion, queda una menos para recalcular el plan
recalcular--;

// cuando hemos ejecutado 3 acciones, recalculamos el plan
if (recalcular == 0){
   hayPlan = false;

   // si tenemos mas de la mita de tiempo, recalculamos cada 4 acciones
```

```
    if (sensores.tiempo < 130){
        recalcular = 3;
    } else if (sensores.tiempo < 160){
        // si nos queda entre 130 y 160 segundos recalculamos
        // cada 5 acciones
        recalcular = 5;
    } else if (sensores.tiempo < 200){
        // si nos queda entre 160 y 200 segundos recalculamos
        // cada 15 acciones
        recalcular = 15;
    } else if (sensores.tiempo < 250) {
        // si nos queda entre 200 y 250 segundos recalculamos cada 40 acciones
        recalcular = 40;
    } else {
        // si hemos consumido 250 segundos recalculamos cada 80 acciones
        recalcular = 90;
    }
}
```

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 dependiendo de la acción realizada anteriormente.

```
// actualizamos unas variables "imaginarias" que contienen la
// una posicion relativa
switch (ultimaAccion){
   case actTURN_R: brujula = (brujula+1)%4; break;
   case actTURN_L: brujula = (brujula+3)%4; break;
   case actFORWARD:
      switch (brujula){
         case 0: fil_img--; break;
         case 1: col_img++; break;
         case 2: fil_img++; break;
         case 3: col_img--; break;
      }
      break;
}
```

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

```
// cada vez que pasamos un paso en una casilla, actualizamos su
// valor en 1
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:

```
// buscamos si hay una casilla PK en el campo de vision
int posK = -1;

for (auto it = sensores.terreno.begin(); it != sensores.terreno.end() && posK == -1; ++it){
   if ((*it) == 'K')
      posK = distance(sensores.terreno.begin(), it);
}
```

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 no la encontramos, seguimos un movimiento reactivo basado en
// un pulgarcito, es decir, almacenar sobre que casilla avanzamos
// y evitar pasar de nuevo por ahi
if (posK == -1){

   int c_izq, c_der, c_del;

   //si es un obstaculo, el valor de pasar por ahi lo ponemos
   // muy alto para que evite seleccionar ese camino
   if (EsObstaculo(sensores.terreno[2])){
      switch (brujula) {
         case 0: mapaImg[fil_img-1][col_img] = 999999; break;
         case 1: mapaImg[fil_img][col_img+1] = 999999; break;
         case 2: mapaImg[fil_img+1][col_img] = 999999; break;
         case 3: mapaImg[fil_img][col_img-1] = 999999; break;
      }
   }

   // seleccionamos los tres siguientes candidatos, ir a la
   // izquierda, a la derecha, o avanzar
   switch (brujula) {
```

```
        case 0:
            c_izq = mapaImg[fil_img][col_img-1];
            c_der = mapaImg[fil_img][col_img+1];
            c_del = mapaImg[fil_img-1][col_img];
            break;
        case 1:
            c_izq = mapaImg[fil_img-1][col_img];
            c_der = mapaImg[fil_img+1][col_img];
            c_del = mapaImg[fil_img][col_img+1];
            break;
        case 2:
            c_izq = mapaImg[fil_img][col_img+1];
            c_der = mapaImg[fil_img][col_img-1];
            c_del = mapaImg[fil_img+1][col_img];
            break;
        case 3:
            c_izq = mapaImg[fil_img+1][col_img];
            c_der = mapaImg[fil_img-1][col_img];
            c_del = mapaImg[fil_img][col_img-1];
            break;

    }

    //por defecto avanzamos
    sigAccion = actFORWARD;


    if (c_der < c_del){

        if (c_izq < c_der) {
            // si el de menor valor es ir a la izquierda
            sigAccion = actTURN_L;
        }
        else {
            // si el de menor valor es ir a la derecha
            sigAccion = actTURN_R;
        }

    } else if (c_izq < c_del) {
        if (c_izq < c_der) {
            // si el de menor valor es ir a la izquierda
            sigAccion = actTURN_L;
        }
        else {
            // si el de menor valor es ir a la derecha
            sigAccion = actTURN_R;
        }
```

```
    }
    // en otro caso, el mejor es avanzar, luego no tocamos nada


}
```

Si **posK != -1** quiere decir que hay un PK en el campo de visión, por lo que creamos un plan para llegar hasta el:

```
else{
    // si encontramos un PK en el campo de vision

    int av = 0;
    int lateral = 0;

    // comprobamos cuantas casillas hay que avanzar
    if ( posK > 0 and posK < 4 ){
        av = 1;
        // si tenemos que avanzar 1, miramos si tenemos qe ir a la
        // derecha o a la izquierda
        if (posK > 2 )
            lateral = 1;
        else if (posK < 2)
            lateral = -1;

    } else if (posK < 9){
        // si tenemos que avanzar 2
        av = 2;
        // miramos si tenemos que ir a la izquierda o a la derecha
        // y comprobamos si lateralmente tenemos que avanzar 1 o 2
        // casillas
        if (posK > 6 )
            if (posK % 2 == 0) lateral = 2; else lateral = 1;
        else if (posK < 6 )
            if (posK % 2 == 0) lateral = -2; else lateral = -1;

    } else {
        // si tenemos que avanzar 3 casillas
        av = 3;
        // miramos si tenemos que ir a la izquierda o a la derecha
        // y comprobamos si lateralmente tenemos que avanzar 1 o 2
        // casillas
        if (posK > 12 )
            if (posK % 2 == 0) lateral = 2; else lateral = 1;
```

```
      else if (posK < 12 )
         if (posK % 2 == 0) lateral = -2; else lateral = -1;

      // si son las ultimas casillas del campo de visión tenemos
      // que avanzar un poco mas
      if (posK == 9) lateral -= 2;
      if (posK == 15) lateral += 2;

   }

   // limpiamos el plan
   plan.clear();
   //introducimos los avances
   for (int i = 0; i < av; i++){
      plan.push_back(actFORWARD);
   }

   // dependiendo de si lateral es positivo o negativo
   // tenemos que girar a la derecha o a la izquierda
   if (lateral > 0){
      plan.push_back(actTURN_R);
   } else{
      plan.push_back(actTURN_L);
   }

   // introducimos los avances que realizamos una vez giramos
   for (int i = 0; i < abs(lateral); i++){
      plan.push_back(actFORWARD);
   }

   //pintamos el plan
   PintaPlan(plan);

   // ejecutamos el plan
   hayPlan = true;


}
```

Ahora comprobamos si hay plan para llegar al PK y lo ejecutamos

```
// si tenemos plan, lo ejecutamos
if (hayPlan and plan.size()>0){
```

```
   sigAccion = actIDLE;

   // si el plan pasa por un obstaculo, giramos a la derecha y recalculamos
   if (EsObstaculo(sensores.terreno[2]) and plan.front() == actFORWARD ){
      hayPlan = false;
      sigAccion = actTURN_R;
   } else if (sensores.superficie[2] != 'a' || plan.front() != actFORWARD ){
      //si no pasa por obstaculo, y delante no tenemos un aldeano
      // ejecutamos el plan
      sigAccion = plan.front();
      plan.erase(plan.begin());
   }

}
```

Por último, ya estemos en el nivel 1 o 2, devolvemos la ultima acción

```
//Recordar ultima accion
ultimaAccion = sigAccion;
return sigAccion;
```

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.

```
bool ComportamientoJugador::pathFinding_A_Estrella(const estado &origen, \
                                                   const estado &destino,\
                                                   list<Action> &plan) {
    cout << "Calculando plan\n";
    plan.clear();

    // almacenamos los nodos en un multiset ordenado por coste_total
    multiset<nodo, OrdenaNodo> abiertos;
    set<estado,ComparaEstados> generados;

    // seleccionamos el primer nodo como origen
    nodo current;
    current.st = origen;
    // establecemos el coste del estado a 0
    current.st.coste = 0;

    current.secuencia.clear();

    //establecemos el coste del nodo a 0
    current.coste = 0;
    current.coste_total = 0;

    // calculamos su heuristica usando distancia manhattan
    int distancia_manhattan = abs(current.st.fila - destino.fila) + \
                              abs(current.st.columna - destino.columna);

    // actualizamos su coste total e insertamos en abiertos
```

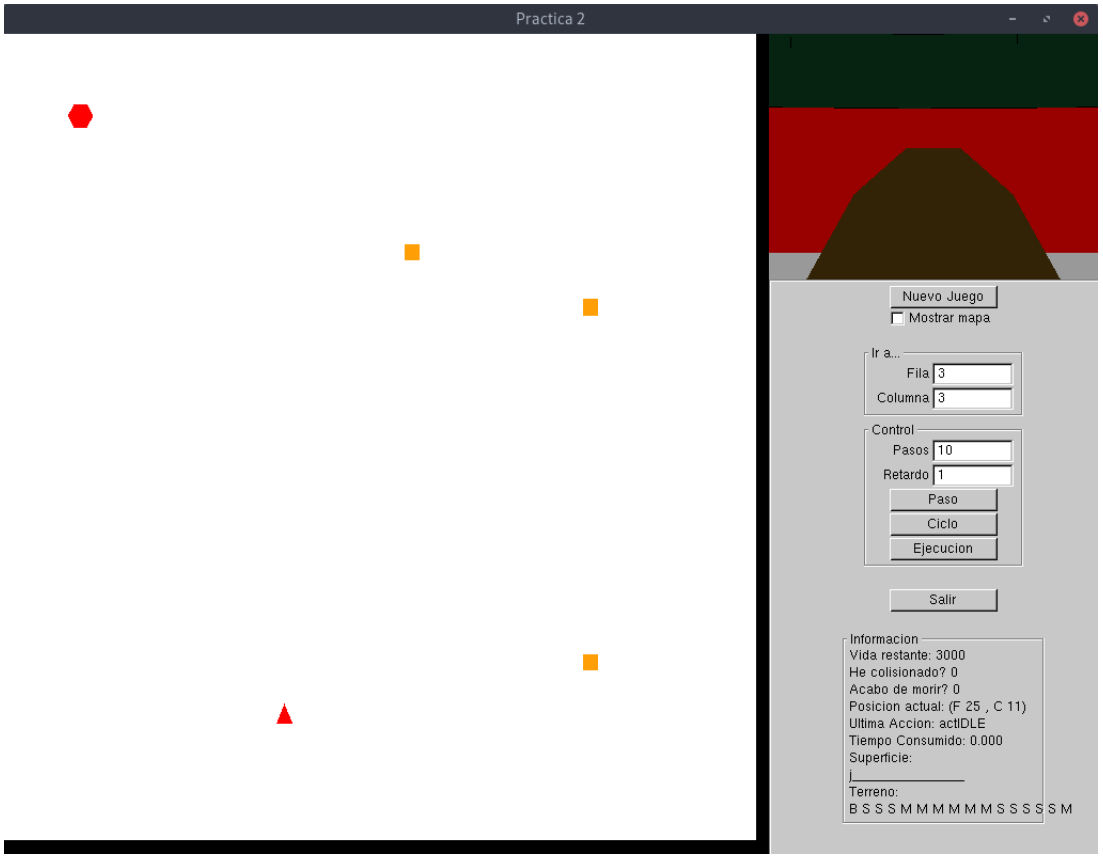
```
current.coste_total += distancia_manhattan;
current.heuristica = distancia_manhattan;
abiertos.insert(current);

// mientras queden nodos por explorar y no estemos en el destino
while (!abiertos.empty() and (current.st.fila!=destino.fila or  \
                              current.st.columna != destino.columna)){

   // borramos el nuevo nodo current y lo metemos en cerrados
   abiertos.erase(abiertos.begin());
   generados.insert(current.st);

   // hijo resultante de girar a la derecha
   nodo rightSon = current;
   rightSon.st.orientacion = (rightSon.st.orientacion+1) % 4;

   // calculamos su coste, heuristica, y coste total a partir de estas
   rightSon.coste += calcularCoste(current.st, rightSon.st );
   rightSon.heuristica = abs(rightSon.st.fila - destino.fila) + \
                         abs(rightSon.st.columna - destino.columna);

   rightSon.coste_total = rightSon.coste + rightSon.heuristica;

   rightSon.st.coste = rightSon.coste;

   // buscamos el nodo en generados, es decir en los cerrados
   auto it = generados.find(rightSon.st);

   if(it == generados.end()){

      // si no lo encontramos lo buscamos en abiertos
      auto it = nodoEncontrado(abiertos, rightSon);

      // si lo encontramos en abiertos, nos quedamos con el de mejor
      // coste_total
      if (it != abiertos.end()){
         if (rightSon.coste_total < it->coste_total){
            abiertos.erase(it);
            rightSon.secuencia.push_back(actTURN_R);
            abiertos.insert(rightSon);
         }
      } else{
         // si no lo encontramos lo añadimos a abiertos
         rightSon.secuencia.push_back(actTURN_R);
         abiertos.insert(rightSon);
      }
```

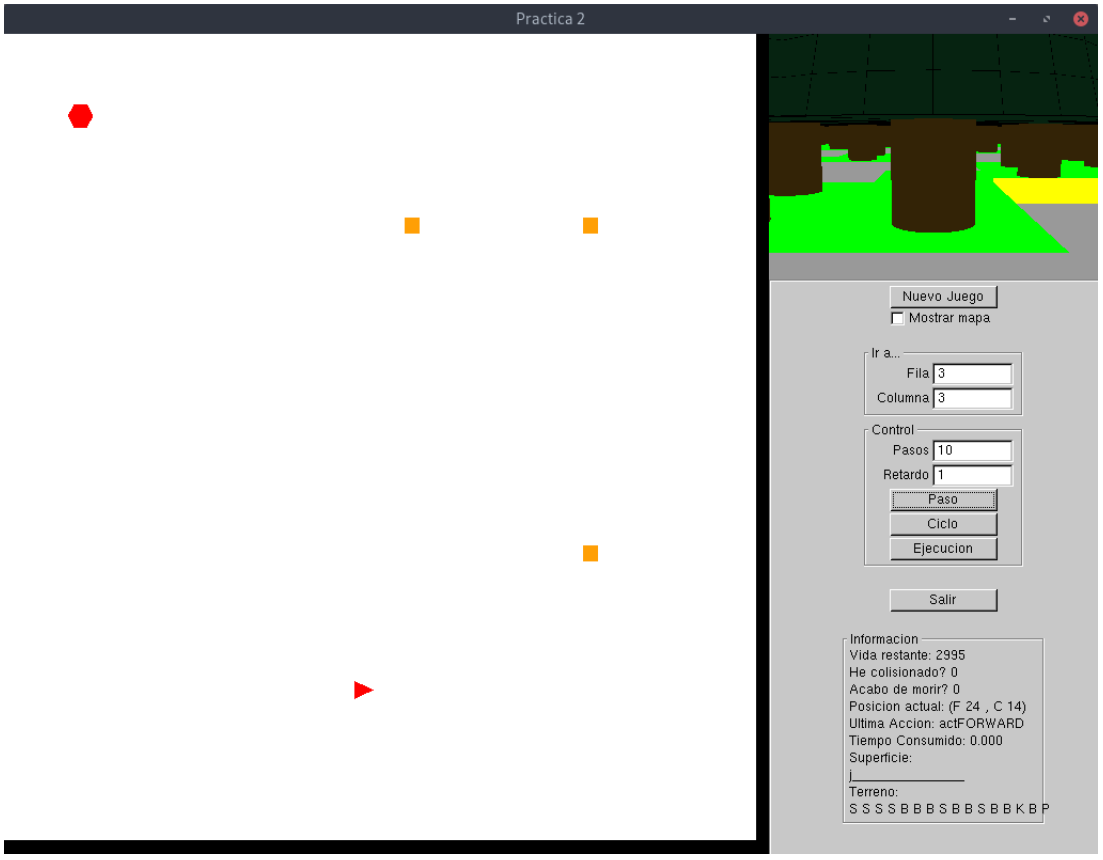
```
} else {
   // si lo encontramos, nos quedamos con el estado que tenga mejor coste
   if (it->coste > rightSon.coste){
      generados.erase(it);
      rightSon.secuencia.push_back(actTURN_R);
      abiertos.insert(rightSon);
   }
}

// hijo resultante de girar a la izquierda
nodo leftSon = current;
leftSon.st.orientacion = (leftSon.st.orientacion+3) % 4;

// calculamos su coste, heuristica, y coste total a partir de estas
leftSon.coste += calcularCoste(current.st, leftSon.st );
leftSon.heuristica = abs(leftSon.st.fila - destino.fila) + \
                     abs(leftSon.st.columna - destino.columna);
leftSon.coste_total = leftSon.coste + leftSon.heuristica;

leftSon.st.coste = leftSon.coste;

// buscamos el estado en generados, es decir los ya explorados
it = generados.find(leftSon.st);

if(it == generados.end()){

   //si no lo encontramos, buscamos el nodo en abiertos
   auto it = nodoEncontrado(abiertos, leftSon);

   // si encontramos el nodo en abiertos, nos quedamos con el de mejor coste
   if (it != abiertos.end()){
      if (leftSon.coste_total < it->coste_total){
         abiertos.erase(it);
         leftSon.secuencia.push_back(actTURN_L);
         abiertos.insert(leftSon);
      }
   } else{
      // si no lo encontramos, lo añadimos
      leftSon.secuencia.push_back(actTURN_L);
      abiertos.insert(leftSon);
   }



} else{
   // si lo encontramos, nos quedamos con el estado que tenga mejor coste
   if (it->coste > leftSon.coste){
```

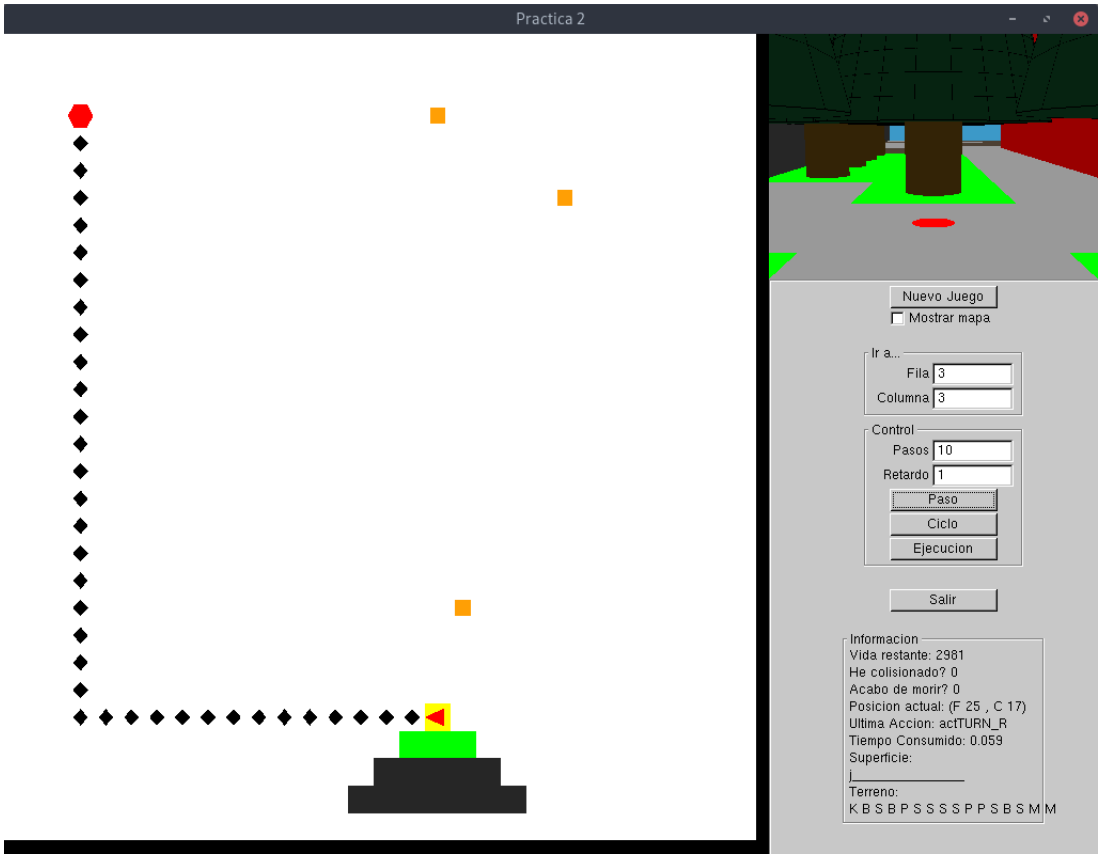
```
            generados.erase(it);
            leftSon.secuencia.push_back(actTURN_L);
            abiertos.insert(leftSon);
        }
    }

    // hijo resultante de avanzar
    nodo fowardSon = current;
    // si no es un obstaculo
    if(!HayObstaculoDelante(fowardSon.st)){

        // calculamos su coste, heuristica y coste total
        fowardSon.coste += calcularCoste(current.st, fowardSon.st );
        fowardSon.heuristica = abs(fowardSon.st.fila - destino.fila) + \
                               abs(fowardSon.st.columna - destino.columna);
        fowardSon.coste_total = fowardSon.coste + fowardSon.heuristica;

        fowardSon.st.coste = fowardSon.coste;

        // lo buscamos en generados, es decir, los ya explorados
        it = generados.find(fowardSon.st);

        if(it == generados.end()){
            // si no lo encontramos lo buscamos en abiertos
            auto it = nodoEncontrado(abiertos, fowardSon);

            // si lo encontramos en abiertos, nos quedamos con el de mejor coste
            if (it != abiertos.end()){
                if (fowardSon.coste_total < it->coste_total){
                    abiertos.erase(it);
                    fowardSon.secuencia.push_back(actFORWARD);
                    abiertos.insert(fowardSon);
                }
            } else{
                // si no lo encontramos en abiertos, lo añadimos
                fowardSon.secuencia.push_back(actFORWARD);
                abiertos.insert(fowardSon);
            }



        } else{
            // si lo encontramos en cerrados, nos quedamos con el estado con mejor coste
            if (it->coste > fowardSon.coste){
                generados.erase(it);
                fowardSon.secuencia.push_back(actFORWARD);
                abiertos.insert(fowardSon);
```

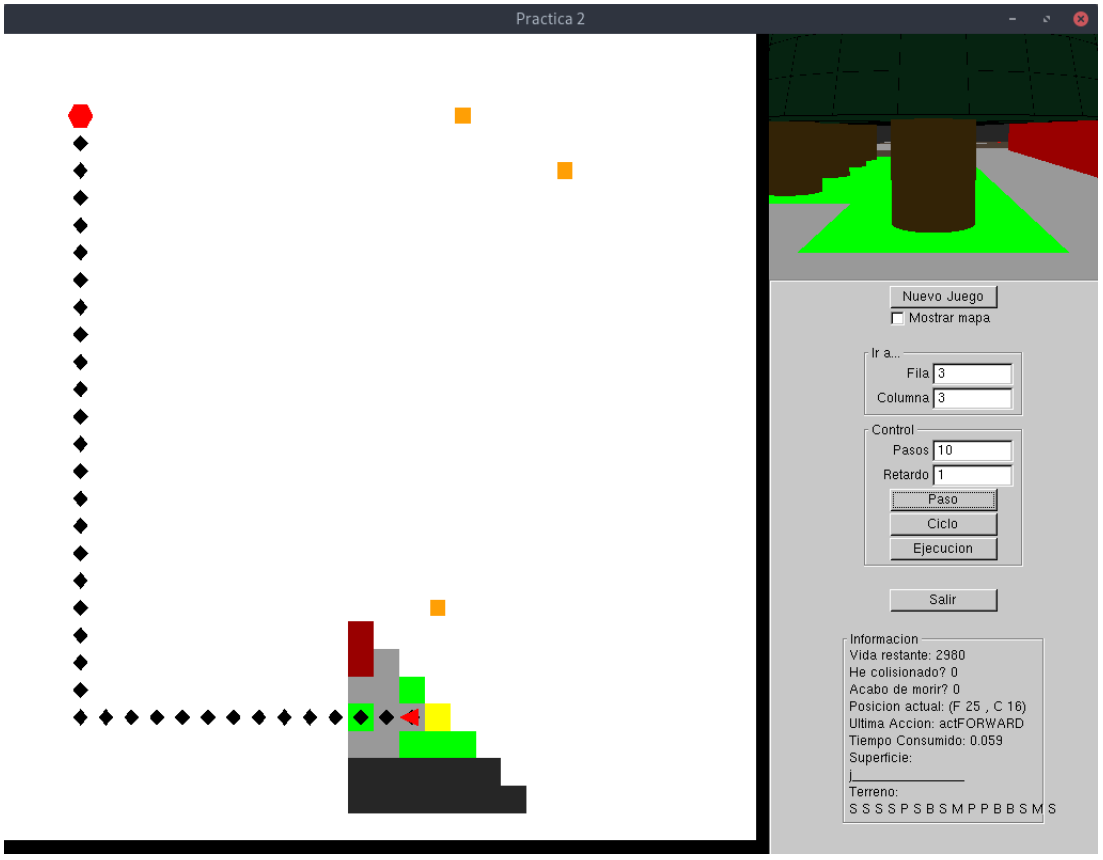
```
            }
         }
      }


      if (!abiertos.empty()){
         current = (*abiertos.begin());
      }

   }

   cout << "Terminada la busqueda\n";

   if (current.st.fila == destino.fila and current.st.columna == destino.columna){
      cout << "Cargando el plan\n";
      plan = current.secuencia;
      cout << "Longitud del plan: " << plan.size() << endl;
      cout << "Coste del plan " << current.coste << endl;
      PintaPlan(plan);
      // ver el plan en el mapa
      VisualizaPlan(origen, plan);
      return true;
   }
   else {
      cout << "No encontrado plan\n";
   }

   return false;
}
```

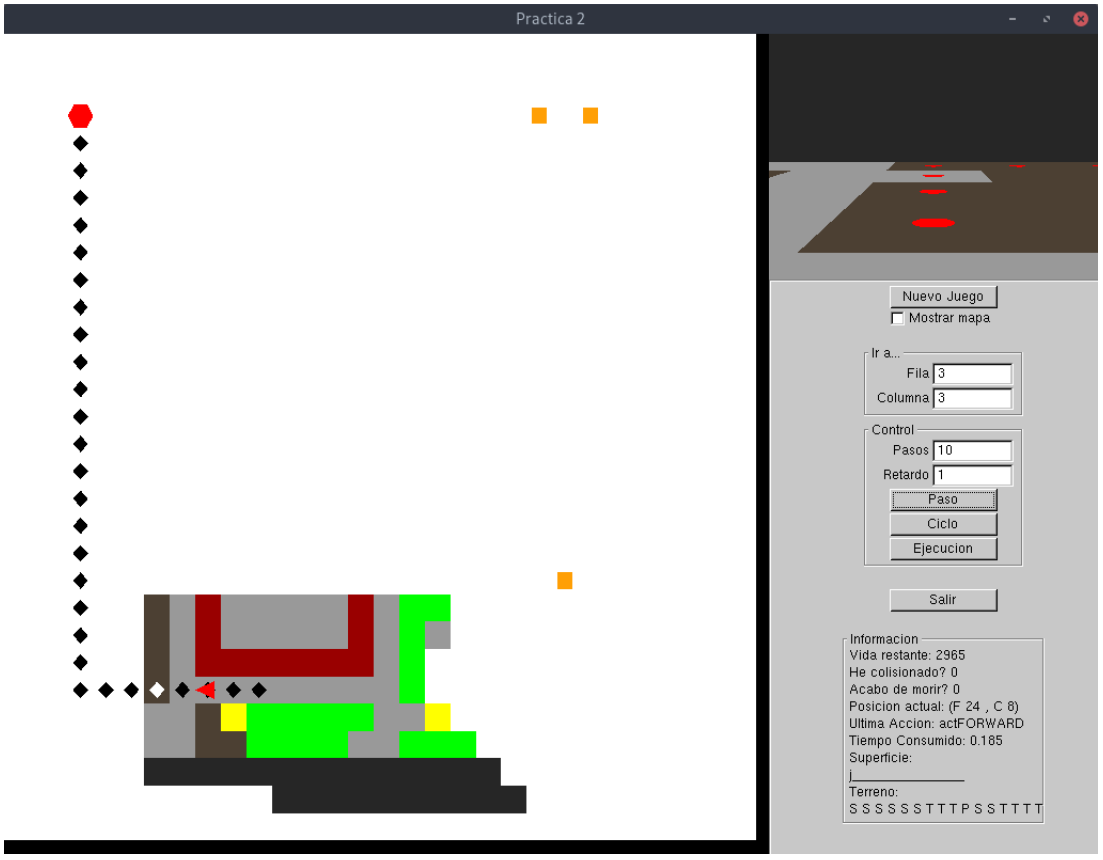
Un ejemplo de la ejecución del nivel 2:

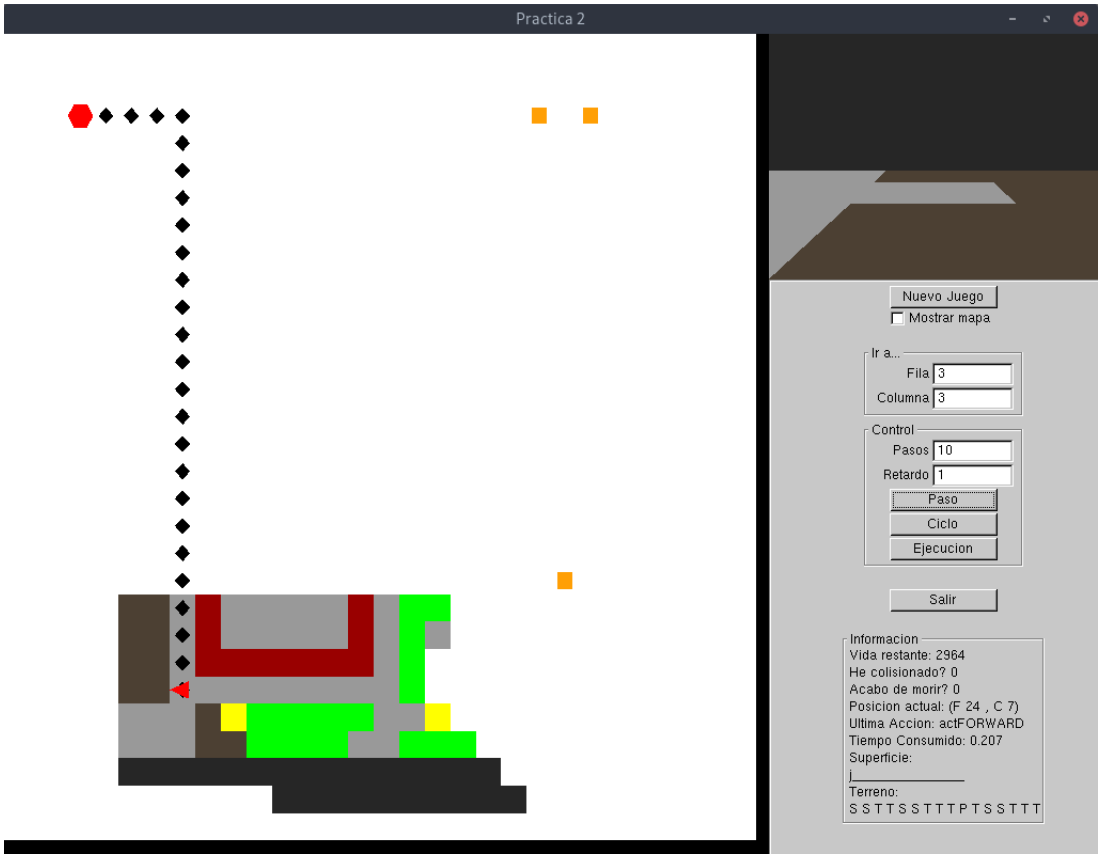

Practica 2
Nuevo Juego
Mostrar mapa
Ir a...
Fila 3
Columna 3
Control
Pasos 10
Retardo 1
Paso
Ciclo
Ejecucion
Salir
Informacion
Vida restante: 3000
He colisionado? 0
Acabo de morir? 0
Posicion actual: (F 25 , C 11)
Ultima Accion: actIDLE
Tiempo Consumido: 0.000
Superficie:
Terreno:
B S S S M M M M M M S S S S S M

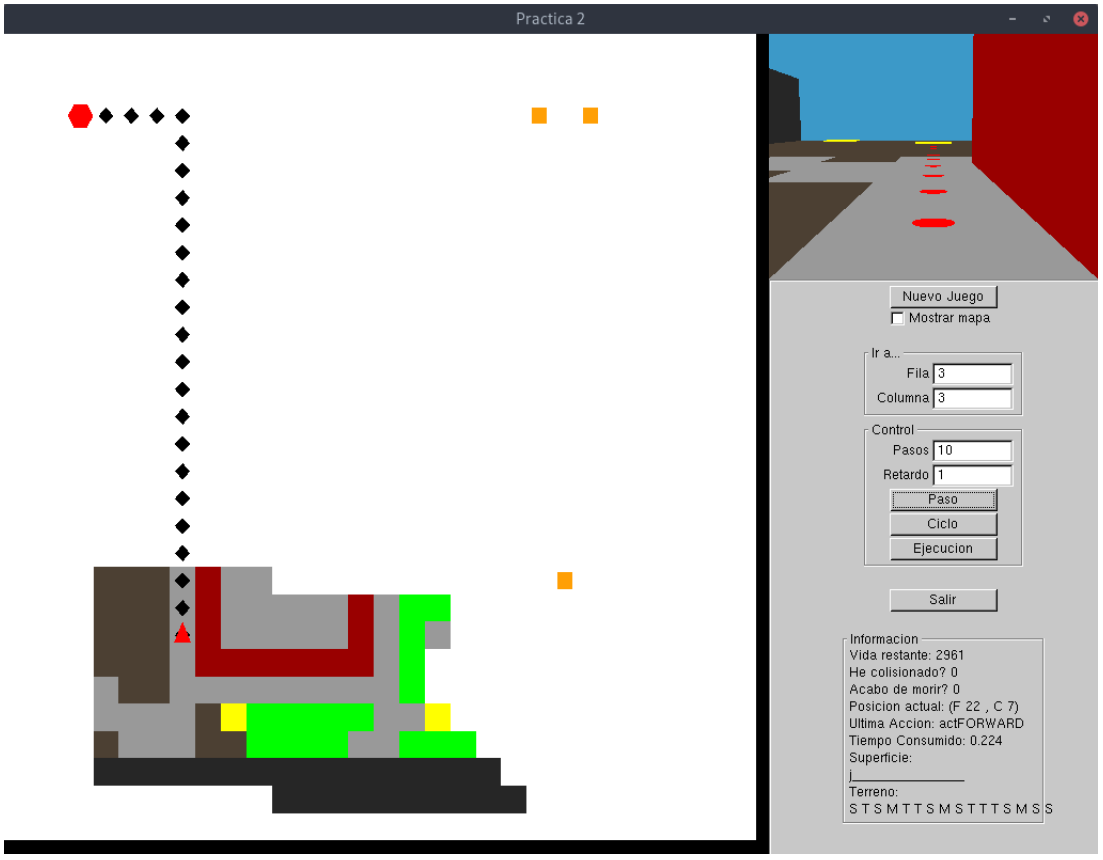

Practica 2
Nuevo Juego
Mostrar mapa
Ir a...
Fila 3
Columna 3
Control
Pasos 10
Retardo 1
Paso
Ciclo
Ejecucion
Salir
Informacion
Vida restante: 2995
He colisionado? 0
Acabo de morir? 0
Posicion actual: (F 24 , C 14)
Ultima Accion: actFORWARD
Tiempo Consumido: 0.000
Superficie:
Terreno:
S S S S B B B S B B S B B K B P

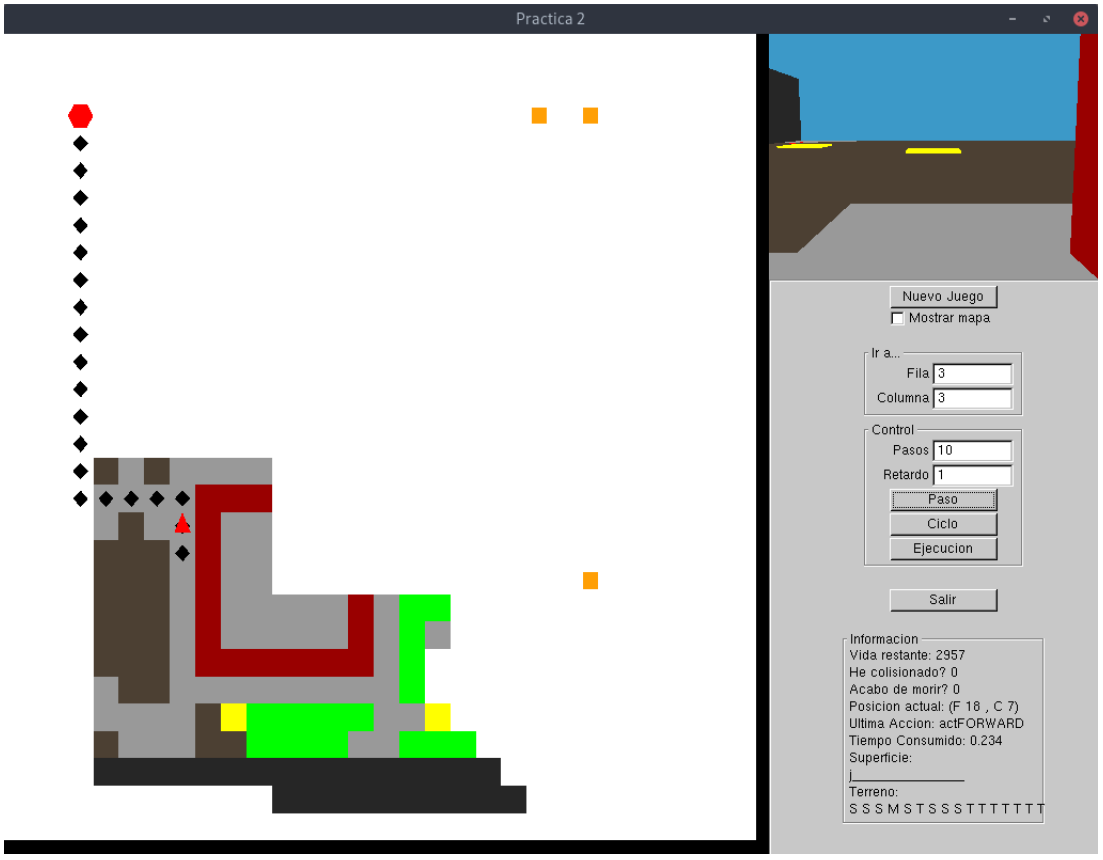

Practica 2
Nuevo Juego
Mostrar mapa
Ir a...
Fila 3
Columna 3
Control
Pasos 10
Retardo 1
Paso
Ciclo
Ejecucion
Salir
Informacion
Vida restante: 2981
He colisionado? 0
Acabo de morir? 0
Posicion actual: (F 25 , C 17)
Ultima Accion: actTURN_R
Tiempo Consumido: 0.059
Superficie:
Terreno:
K B S B P S S S S P P S B S M M

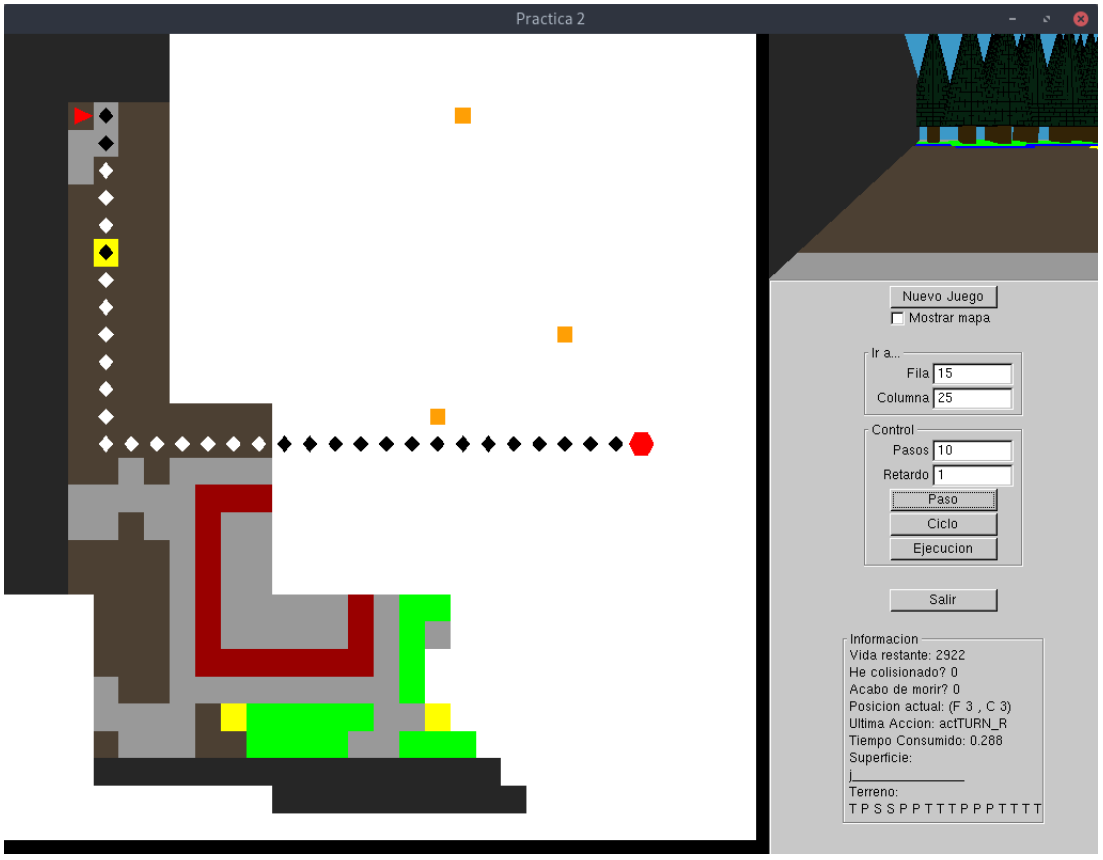

Practica 2
Nuevo Juego
Mostrar mapa
Ir a...
Fila 3
Columna 3
Control
Pasos 10
Retardo 1
Paso
Ciclo
Ejecucion
Salir
Informacion
Vida restante: 2980
He colisionado? 0
Acabo de morir? 0
Posicion actual: (F 25 , C 16)
Ultima Accion: actFORWARD
Tiempo Consumido: 0.059
Superficie:
Terreno:
S S S S P S B S M P P B B S M S


Practica 2
Nuevo Juego
Mostrar mapa
Ir a...
Fila 3
Columna 3
Control
Pasos 10
Retardo 1
Paso
Ciclo
Ejecucion
Salir
Informacion
Vida restante: 2965
He colisionado? 0
Acabo de morir? 0
Posicion actual: (F 24 , C 8)
Ultima Accion: actFORWARD
Tiempo Consumido: 0.165
Superficie:
Terreno:
S S S S S S T T T P S S T T T T


Practica 2
Nuevo Juego
Mostrar mapa
Ir a...
Fila 3
Columna 3
Control
Pasos 10
Retardo 1
Paso
Ciclo
Ejecucion
Salir
Informacion
Vida restante: 2964
He colisionado? 0
Acabo de morir? 0
Posicion actual: (F 24 , C 7)
Ultima Accion: actFORWARD
Tiempo Consumido: 0.207
Superficie:
Terreno:
S S T T S S T T T P T S S T T T


Practica 2
Nuevo Juego
Mostrar mapa
Ir a...
Fila 3
Columna 3
Control
Pasos 10
Retardo 1
Paso
Ciclo
Ejecucion
Salir
Informacion
Vida restante: 2961
He colisionado? 0
Acabo de morir? 0
Posicion actual: (F 22 , C 7)
Ultima Accion: actFORWARD
Tiempo Consumido: 0.224
Superficie:
Terreno:
S T S M T T S M S T T T S M S S


Practica 2
Nuevo Juego
Mostrar mapa
Ir a...
Fila 3
Columna 3
Control
Pasos 10
Retardo 1
Paso
Ciclo
Ejecucion
Salir
Informacion
Vida restante: 2957
He colisionado? 0
Acabo de morir? 0
Posicion actual: (F 18 , C 7)
Ultima Accion: actFORWARD
Tiempo Consumido: 0.234
Superficie:
Terreno:
S S S M S T S S S T T T T T T T


Practica 2
Nuevo Juego
Mostrar mapa
Ir a...
Fila 15
Columna 25
Control
Pasos 10
Retardo 1
Paso
Ciclo
Ejecucion
Salir
Informacion
Vida restante: 2922
He colisionado? 0
Acabo de morir? 0
Posicion actual: (F 3 , C 3)
Ultima Accion: actTURN_R
Tiempo Consumido: 0.288
Superficie:
Terreno:
T P S S P P T T T T P P P T T T T