Inteligencia Artificial

grado en ingeniería informática

Práctica 2.

Los mundos de Belkan

Autor
Carlos Sánchez Páez

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2017-2018

Índice

1. Introducción

Esta práctica consiste en el desarrollo de un agente deliberativo que deberá encontrar el camino hacia un objetivo evitando una serie de obstáculos (fijos y móviles). Se divide en tres niveles:

1. **Nivel 1**. Encontrar el camino a un destino sin obstáculos móviles.
2. **Nivel 2**. Encontrar el camino a un destino con obstáculos móviles.
3. **Nivel 3**. Deliberativo + reactivo: encontrar un objetivo a la vez que se va descubriendo el mapa.

Figura 1: Los mundos de Belkan.

2. Nivel 1.

Para desarrollar el nivel 1 necesitamos utilizar un algoritmo de búsqueda para encontrar un camino al objetivo. En mi caso, utilizaré la **búsqueda por anchura**. Su pseudocódigo sería el siguiente:

1. Introducimos el origen en la cola de abiertos (por visitar)
2. Mientras que queden elementos en abiertos o no hayamos encontrado el camino:
   a) Saco el primer elemento de abiertos.
   b) Si es la solución, añado el nodo actual al histórico y salgo del bucle.
   c) En caso contrario:
      1) Lo meto en cerrados (ya visitados).
      2) Para cada uno de sus adyacentes (delante,detrás,izquierda y derecha):
         - Si es viable (no está ni en abiertos ni en cerrados y es transitable), lo meto en abiertos.
3. Devuelvo la solución

Consideraciones:

- Como la cola de cerrados solo la utilizaremos para ver si un elemento ha sido ya explorado, la implemento mediante una matriz de booleanos. De esta forma, si quiero ver si una casilla ha sido explorada únicamente tengo que acceder al valor correspondiente de la matriz, pasando de $O(n)$ a $O(1)$ a costa de un gasto mayor de memoria.
- En cuanto a la cola de abiertos, también implemento una matriz de booleanos que servirá para la operación de comprobación.
- Para no tener que reconstruir el camino una vez encontrada la solución, en el *struct* estado he añadido una list¡estado¿que contenga los estados predecesores. De esta forma, una vez llegue al final solo tendré que devolver este campo (añadiendo el estado destino).
- Como solamente necesito conocer los antecesores de los estados que son válidos, cada vez que creo un nuevo adyacente libero la lista de antecesores de su predecesor, evitando duplicidad.

Aplicado a nuestro caso, el código sería el siguiente:

```
list<estado> ComportamientoJugador::BusquedaEnAnchura
    (const estado & origen, const estado & destino) {
  queue<estado> abiertos;   // Cola de abiertos.
  abiertos.push(origen);   //Introducimos el origen
  InicializarMatrices();   // Ponemos las matrices a false.
  m_abiertos[origen.fila][origen.columna] = true;
  int dx[4] = { -1, 0, 1 , 0};  //Para calcular la adyacencia
  int dy[4] = {0, 1, 0, -1};
  bool encontrado = false;
  list<estado> resultado;    //Resultado a devolver

  while (!abiertos.empty() && !encontrado) {
    estado actual = abiertos.front();  //Sacamos un estado de la cola
    abiertos.pop();
    if ( actual == destino) {  //Si hemos llegado, devolvemos
          // los pasos que hemos seguido
      resultado = actual.anteriores;
      resultado.push_back(actual);
      encontrado = true;
    }
    else {
      m_cerrados[actual.fila][actual.columna] = true; // Añadimos a
       ↪  cerrados
      for (int i = 0; i < 4; i++) { //Adjustamos adyacentes
        int fila_ady = dx[i] + actual.fila;
        int col_ady = dy[i] + actual.columna;
        if (EsViable(fila_ady, col_ady)) { //Si puedo pasar por el
         ↪  adyacente,
              // lo añado a la cola de abiertos
          estado adyacente = CrearAdyacente(fila_ady, col_ady, i,
           ↪  actual);
          abiertos.push(adyacente);
          m_abiertos[adyacente.fila][adyacente.columna] = true;
        }
      }
    }
    actual.anteriores.clear(); //Libero memoria,
    // ya que esta información estará en el adyacente.
  }
  return resultado;
}
```

Figura 2: Busqueda en anchura

```
bool ComportamientoJugador::EsViable(int fila, int columna) {
  return !m_abiertos[fila][columna] && !m_cerrados[fila][columna]
  && PUEDO_PASAR.count(mapaResultado[fila][columna]);
}
```

Figura 3: Es viable

PUEDO_PASAR es un set de la *STL* que contiene los valores de las casillas transitables (S,T y K).

```
estado ComportamientoJugador::CrearAdyacente(int f, int c, int o, estado
↪  &actual) {
  estado adyacente;
  adyacente.fila = f;
  adyacente.columna = c;
  adyacente.orientacion = o;
  for (auto it = actual.anteriores.begin();
    it != actual.anteriores.end(); ++it) {
    it->anteriores.clear();   //Elimino los padres, ya que sólo necesito
    ↪  conservar fila, columna y orientación.
    adyacente.anteriores.push_back(*it);
  }
  adyacente.anteriores.push_back(actual);
  return adyacente;
}
```

Figura 4: Crear adyacente

```
void ComportamientoJugador::InicializarMatrices() {
  for (int i = 0; i < TAM; i++) {
    for (int j = 0; j < TAM; j++) {
      m_cerrados[i][j] = false;
      m_abiertos[i][j] = false;
    }
  }
}
```

Figura 5: Inicializar matrices

```
bool operator==(const estado &uno, const estado &otro) {
  return uno.fila == otro.fila && uno.columna == otro.columna;
}

bool operator!=(const estado &uno, const estado &otro) {
  return !(uno == otro);
}
```

Figura 6: Operadores lógicos

Una vez llegados a este punto, hemos encontrado una secuencia de estados que nos llevan desde el origen hasta el destino. Por tanto, lo único que resta es transformarlos a las acciones que deberá llevar a cabo nuestro personaje. Para ello, cogeremos los elementos de la lista de dos en dos e iremos añadiendo las acciones en función de la orientación.

```
list<Action> ComportamientoJugador::calcularListaAcciones(const
↪  list<estado> &lista) {
  list<Action> resultado;
  list<estado>::const_iterator it_anterior;
  list<estado>::const_iterator it_siguiente;
  for (it_siguiente = it_anterior = lista.begin(); *it_siguiente !=
  ↪  lista.back() ; ++it_anterior) {    //Dos iteradores: anterior y
  ↪  siguiente.
    ++it_siguiente;
    //Cuatro  casos: avanzo, giro izquierda , giro derecha o retrocedo
    if (it_anterior->fila > it_siguiente->fila)  {  //ARRIBA
      switch (it_anterior->orientacion) {
      case 0:     //Estoy mirando al norte
        break;
      case 1:      //Este
        resultado.push_back(actTURN_L);
        break;
      case 2:      //Sur
        resultado.push_back(actTURN_R);
        resultado.push_back(actTURN_R);
        break;
      case 3:      //Oeste
        resultado.push_back(actTURN_R);
        break;
      }
    }
    else if (it_anterior->fila < it_siguiente->fila)  {  //ABAJO
      switch (it_anterior->orientacion) {
      case 0:     //Estoy mirando al norte
        resultado.push_back(actTURN_R);
        resultado.push_back(actTURN_R);
        break;
      case 1:      //Este
```

```
31          resultado.push_back(actTURN_R);
32          break;
33        case 2:       //Sur
34          break;
35        case 3:       //Oeste
36          resultado.push_back(actTURN_L);
37          break;
38        }
39      }
40      else if (it_anterior->columna > it_siguiente->columna) {  //IZQUIERDA
41        switch (it_anterior->orientacion) {
42        case 0:     //Estoy mirando al norte
43          resultado.push_back(actTURN_L);
44          break;
45        case 1:       //Este
46          resultado.push_back(actTURN_R);
47          resultado.push_back(actTURN_R);
48          break;
49        case 2:       //Sur
50          resultado.push_back(actTURN_R);
51          break;
52        case 3:       //Oeste
53          break;
54        }
55      }
56      else {    //DERECHA
57        switch (it_anterior->orientacion) {
58        case 0:     //Estoy mirando al norte
59          resultado.push_back(actTURN_R);
60          break;
61        case 1:       //Este
62          break;
63        case 2:       //Sur
64          resultado.push_back(actTURN_L);
65          break;
66        case 3:       //Oeste
67          resultado.push_back(actTURN_R);
68          resultado.push_back(actTURN_R);
69          break;
70        }
71      }
72      resultado.push_back(actFORWARD);
73    }
74    return resultado;
75  }
```

Figura 7: Transcripción a acciones

Por tanto, nuestro método pathfinding sería así:

```
bool ComportamientoJugador::pathFinding(const estado & origen, const estado & destino, list<Action> &plan) {
  high_resolution_clock::time_point tantes;
  high_resolution_clock::time_point tdespues;
  duration<double> tiempo;

  plan.clear();
  tantes = high_resolution_clock::now();
  list<estado> lista = BusquedaEnAnchura(origen, destino);
  tdespues = high_resolution_clock::now();
  tiempo = duration_cast<duration<double>>(tdespues - tantes);
  cout << "Tiempo empleado en el cálculo del plan: " << tiempo.count() << "s." << endl;

  tantes = high_resolution_clock::now();
  plan = calcularListaAcciones(lista);
  tdespues = high_resolution_clock::now();
  tiempo = duration_cast<duration<double>>(tdespues - tantes);
  cout << "Tiempo empleado en la transcripción a acciones: " << tiempo.count() << "s." << endl;

  VisualizaPlan(origen, plan);
  return !lista.empty();  //True si no está vacía
}
```

Figura 8: Pathfinding

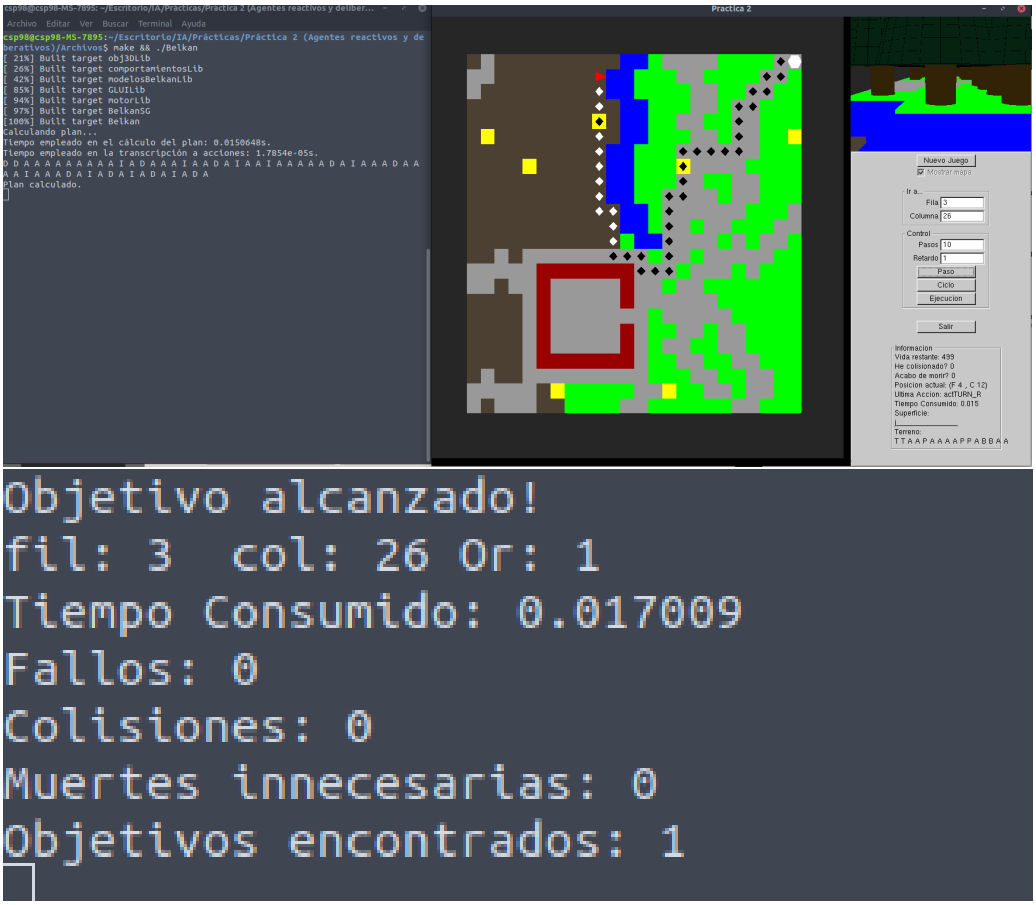

Figura 9: Nivel 1. Ejemplo de funcionamiento.

3. Nivel 2.

En este nivel hay una serie de aldeanos que tendremos que esquivar si nos impiden el paso. La estrategia a seguir es la siguiente:

1. Trazar un plan al igual que en el nivel 1.
2. Si en el trascurso del plan encuentro un aldeano en mi camino:
   a) Coloco temporalmente un muro en la posición del aldeano.
   b) Llamo de nuevo a pathfinding, generando un nuevo camino. Al haber situado un muro en la posición del aldeano, nuestro algoritmo anterior lo esquivará.
   c) Devuelvo la casilla del aldeano a su estado original.

Por tanto, nuestro método *think()* quedaría así:

```
1  Action ComportamientoJugador::think(Sensores sensores) {
2    if (sensores.mensajeF != -1 && primeraVez) {
3      primeraVez = false;
4      fil = sensores.mensajeF;
5      col = sensores.mensajeC;
6    }
7
8    // Actualizar el efecto de la ultima accion
9    switch (ultimaAccion) {
10   case actTURN_R: brujula = (brujula + 1) % 4; break;
11   case actTURN_L: brujula = (brujula + 3) % 4; break;
12   case actFORWARD:
13     switch (brujula) {
14     case 0: fil--; break;
15     case 1: col++; break;
16     case 2: fil++; break;
17     case 3: col--; break;
18     }
19     cout << "fil: " << fil << "  col: " << col << " Or: " << brujula <<
       ↪  endl;
20     break;
21   }
22
23   // Determinar si ha cambiado el destino desde la ultima planificacion
24   if (hayPlan and (sensores.destinoF != destino.fila or sensores.destinoC
     ↪  != destino.columna)) {
25     cout << "El destino ha cambiado\n";
26     hayPlan = false;
27   }
28
29   // Determinar si tengo que construir un plan
30   if (!hayPlan) {
31     //Capto origen y destino.
```

```
32      estado origen;
33      origen.fila = fil;
34      origen.columna = col;
35      origen.orientacion = brujula;
36      destino.fila = sensores.destinoF;
37      destino.columna = sensores.destinoC;
38
39      cout << "Calculando plan..." << endl;
40      hayPlan = pathFinding(origen, destino, plan);
41      PintaPlan(plan);
42      cout << "Plan calculado." << endl;
43      if (!hayPlan)
44        cout << "El objetivo es inalcanzable." << endl;
45    }
46
47
48    // Ejecutar el plan
49    Action sigAccion;
50    if (hayPlan and plan.size() > 0) {
51      sigAccion = plan.front();
52      plan.erase(plan.begin());
53
54      if (sigAccion == actFORWARD and sensores.superficie[2] == 'a') {
55        int f_aux = fil;
56        int c_aux = col;
57
58        estado origen;
59        origen.fila = fil;
60        origen.columna = col;
61        origen.orientacion = brujula;
62
63        destino.fila = sensores.destinoF;
64        destino.columna = sensores.destinoC;
65
66        //Calculo la posición del aldeano.
67        switch (brujula) {
68        case 0:
69          f_aux--;
70          break;
71        case 1:
72          c_aux++;
73          break;
74        case 2:
75          f_aux++;
76          break;
77        case 3:
78          c_aux--;
79          break;
```

```
80          }
81          // Arreglo temporal: pongo un muro en el lugar del aldeano para
            ↪  que no sea transitable y no tener que modificar pathFinding()
82          char aux = mapaResultado[f_aux][c_aux];
83          mapaResultado[f_aux][c_aux] = 'M';
84
85          cout << "Recalculando ruta..." << endl;
86
87          hayPlan = pathFinding(origen, destino, plan);
88
89          mapaResultado[f_aux][c_aux] = aux;
90
91          PintaPlan(plan);
92          cout << "Plan calculado." << endl;
93
94          if (!plan.empty()) {      //Si el plan no está vacío, saco la
            ↪  primera acción. En caso contrario, espero.
95            sigAccion = plan.front();
96            plan.erase(plan.begin());
97          }
98          else {
99            sigAccion = actIDLE;
100         }
101       }
102     }
103     else {
104       sigAccion = actIDLE;
105     }
106     ultimaAccion = sigAccion;
107     return sigAccion;
108 }
```

Figura 10: Think

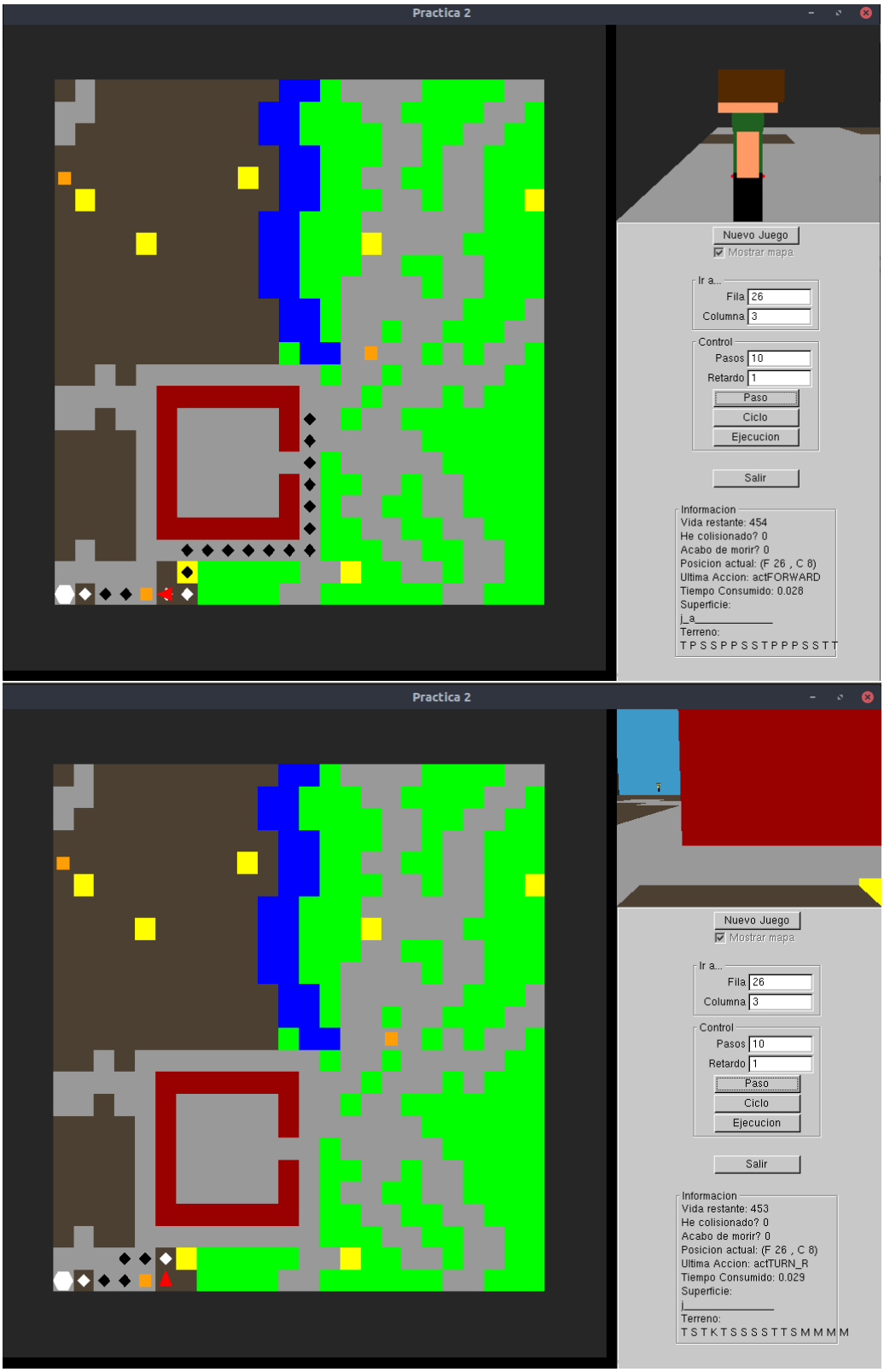

Figura 11: Nivel 2. Ejemplo de funcionamiento.