Práctica 2: Los Mundos de BelKan

José María Martín Luque

29 de abril de 2018

Consideraciones previas

En el presente documento se recoge la implementación del algoritmo A* para la práctica 2 de la asignatura Inteligencia Artificial, así como de otras funciones y estructuras auxiliares necesarias para el correcto funcionamiento del mismo. En el guion de prácticas se establecía que la memoria no podía extenderse más de cinco páginas, algo que creo que es imposible si se pretende incluir y documentar el código.

He realizado la práctica hasta el llamado *nivel 3*, sin embargo, el personaje se detiene al llegar al primer objetivo. Esto es así para mostrar mi rechazo al sistema de evaluación de las prácticas 2 y 3 de esta asignatura, en concreto, al aspecto competitivo del mismo.

La evaluación debe basarse en la comprobación de la adquisición de las competencias que corresponden a cada práctica, no en averiguar qué estudiante puede superar al resto de sus compañeros. Esta forma de evaluar rompe además las dinámicas de grupo de colaboración y compañerismo (algo en lo que también influye la insistencia constante sobre el carácter individual de las prácticas y el plagio).

La educación no es una competición.

1. Implementación hasta el *nivel 2*

Estructuras de apoyo

Estructura `PosicionCuadricula`

He definido este `struct` para almacenar posiciones de la cuadrícula. Se han definido también operaciones básicas de suma, diferencia, salida por *stream*, igualdad y comparación.

```
1  struct PosicionCuadricula
2  {
3    int x, y;
4
5    inline bool operator ==(const PosicionCuadricula& a) const {
6      return a.x == x && a.y == y;
7    }
8
9    inline bool operator !=(const PosicionCuadricula& a) const {
10     return a.x != x || a.y != y;
11   }
12
13   inline bool operator <(const PosicionCuadricula& a) const {
14     return tie(a.x, a.y) < tie(x, y);
15   }
16
17   friend PosicionCuadricula operator+(const PosicionCuadricula &p1
       ↪ , const PosicionCuadricula &p2);
18   friend PosicionCuadricula operator-(const PosicionCuadricula &p1
       ↪ , const PosicionCuadricula &p2);
19   friend ostream& operator<<(ostream& os, const PosicionCuadricula
       ↪ & pos);
20 };
```

Estructura `ColaPrioridad`

Se trata de una implementación genérica de una cola con prioridad.

```
1  template<typename T, typename prioridad_t>
2  struct ColaPrioridad {
3    typedef std::pair<prioridad_t, T> elemento_cp;
4    priority_queue<elemento_cp, vector<elemento_cp>, greater<
       ↪ elemento_cp>>elementos;
```

```
5
6   inline bool vacia() const {
7       return elementos.empty();
8   }
9
10  inline void insertar(T elemento, prioridad_t prioridad) {
11    elementos.emplace(prioridad, elemento);
12  }
13
14  T obtener() {
15    T mejor_elemento = elementos.top().second;
16    elementos.pop();
17    return mejor_elemento;
18  }
19 };
```

Funciones de apoyo

Funciones con un objetivo concreto que serán utilizadas varias veces durante el funcionamiento del programa.

Función **`PosicionCorrecta`**

Comprueba si una `PosicionCuadricula` está dentro de los límites del mapa.

```
1 bool ComportamientoJugador::PosicionCorrecta(PosicionCuadricula
    ↪ pos) {
2         int ancho = mapaResultado.size();
3         int largo = mapaResultado[0].size();
4
5         return 0 <= pos.x && pos.x < ancho && 0 <= pos.y && pos.y
              ↪ < largo;
6 }
```

Función **`PosicionAtravesable`**

Comprueba si nuestro personaje puede atravesar una `PosicionCuadricula` concreta.

```
1 bool ComportamientoJugador::PosicionAtravesable(PosicionCuadricula
    ↪  pos) {
```

```
        char casilla = mapaResultado[pos.x][pos.y];

        if (casilla == 'B' || casilla == 'A' || casilla == 'P' ||
            ↪ casilla == 'M') {
                return false;
        } else {
                return true;
        }
}
```

Funciones del algoritmo

A continuación se describen las funciones principales del algoritmo utilizado para el *pathfinding* del personaje.

Función `ObtenerVecinos`

Esta función nos proporciona las `PosicionCuadricula` colindantes a una dada.

```
map<char, PosicionCuadricula> ComportamientoJugador::
    ↪ ObtenerVecinos(
     PosicionCuadricula pos) {

        // Posibles vecinos de una casilla
        map<char, PosicionCuadricula> direcciones = {
            {'S', PosicionCuadricula{1,0}},
            {'O', PosicionCuadricula{0, -1}},
            {'N', PosicionCuadricula{-1, 0}},
            {'E', PosicionCuadricula{0, 1}}
    };

        map<char, PosicionCuadricula> vecinos;

        for (pair<char, PosicionCuadricula> dir : direcciones) {
        PosicionCuadricula siguiente{pos.x + dir.second.x,
                                     pos.y + dir.second.y};

        if (PosicionCorrecta(siguiente) && PosicionAtravesable(
            ↪ siguiente)) {
             vecinos[dir.first] = siguiente;
        }
        }
```

```
22
23          return vecinos;
24  }
```

Función `Heuristica`

Función que define la heurística usada en el algoritmo. Es la conocida como *distancia Manhattan*.

```
1  double ComportamientoJugador::Heuristica(PosicionCuadricula a,
2                                           PosicionCuadricula b) {
3
4      return abs(a.x - b.x) + abs(a.y - b.y);
5  }
```

Función `AEstrella`

Es la función principal del algoritmo. Es una función de gran tamaño, por lo que la voy a desglosar. Veamos primero la cabecera de la función:

```
1  void ComportamientoJugador::AEstrella(
2          PosicionCuadricula inicio,
3          PosicionCuadricula destino,
4          map<PosicionCuadricula, tuple<PosicionCuadricula, list<
               ↪ Action>,int>>& recorrido) {
```

La función recibe un `inicio`, un `destino` y un `recorrido`, que es un `map` que asigna a cada `PosicionCuadricula` la posición anterior a ella, las acciones que debe realizar el personaje para llegar hasta ella y la orientación que deberá tener.

A continuacón se declaran los valores iniciales de las variables auxiliares usadas durante el algoritmo.

```
1    ColaPrioridad<PosicionCuadricula, double> frontera;
2
3    map<PosicionCuadricula, double> coste_hasta_ahora;
4
5    // Valores iniciales de nuestro recorrido de movimiento
6    frontera.insertar(inicio, 0);
7    list<Action> acciones_iniciales = {actIDLE};
8    recorrido[inicio] = make_tuple(inicio, acciones_iniciales ,
         ↪ brujula);
9    coste_hasta_ahora[inicio] = 0;
```

```
10
11    int coste = 0;
```

En la siguiente línea comienza el bucle principal del algoritmo:

```
1     while (!frontera.vacia()) {
```

La `frontera` son las `PosicionCuadricula` adyacentes a las casillas que ya hemos explorado. Dentro del bucle obtenemos la siguiente `PosicionCuadricula` de la frontera y comprobamos si es el `destino`. Si no lo es, continuamos la ejecución obteniendo los *vecinos* de la casilla actual e iteraremos entre las 4 posibilidades (o menos si quedan fuera del mapa).

```
1       PosicionCuadricula actual = frontera.obtener();
2
3       if (actual == destino) {
4         break;
5       }
6
7       // Orientación del personaje
8       int orientacion_actual = get<2>(recorrido[actual]);
9       int orientacion_siguiente;
10
11      map<char, PosicionCuadricula> vecinos = ObtenerVecinos(actual)
            ↪ ;
12
13      // Iteramos entre todos los posibles vecinos de la casilla
            ↪ actual
14      for (pair<char, PosicionCuadricula> siguiente : vecinos) {
```

Dentro de este bucle comenzamos asignando las variables auxiliares que vamos a utilizar:

```
1         PosicionCuadricula siguiente_pos = siguiente.second;
2         char siguiente_dir = siguiente.first;
3
4         list<Action> acciones;
```

Las acciones que tendrá que realizar nuestro personaje para avanzar a esta `PosicionCuadricula` dependerá de su orientación actual así como de dónde se encuentre dicha posición. Esto lo resolveremos en el siguiente `switch`:

```
1         // Para mover a nuestro personaje será necesario cambiar
2         // su orientación
3         switch(siguiente_dir) {
4           case 'N':
```

```
5           switch (orientacion_actual) {
6             case 1: acciones.push_front(actTURN_L);
7                     break;
8             case 2: acciones.push_front(actTURN_L);
9                     acciones.push_front(actTURN_L);
10                    break;
11            case 3: acciones.push_front(actTURN_R);
12                    break;
13          }
14          orientacion_siguiente = 0;
15          break;
16        case 'E':
17          switch (orientacion_actual) {
18            case 0: acciones.push_front(actTURN_R);
19                    break;
20            case 2: acciones.push_front(actTURN_L);
21                    break;
22            case 3: acciones.push_front(actTURN_R);
23                    acciones.push_front(actTURN_R);
24                    break;
25          }
26          orientacion_siguiente = 1;
27          break;
28        case 'S':
29          switch (orientacion_actual) {
30            case 0: acciones.push_front(actTURN_R);
31                    acciones.push_front(actTURN_R);
32                    break;
33            case 1: acciones.push_front(actTURN_R);
34                    break;
35            case 3: acciones.push_front(actTURN_L);
36                    break;
37          }
38          orientacion_siguiente = 2;
39          break;
40        case 'O':
41          switch (orientacion_actual) {
42            case 0: acciones.push_front(actTURN_L);
43                    break;
44            case 1: acciones.push_front(actTURN_L);
45                    acciones.push_front(actTURN_L);
46                    break;
47            case 2: acciones.push_front(actTURN_R);
48                    break;
49          }
```

```
50            orientacion_siguiente = 3;
51        }
52        // Una vez orientado, lo movemos hacia adelante
53        acciones.push_front(actFORWARD);
```

Como ya sabemos, el algoritmo A* utiliza dos métricas para expresar la optimalidad de un camino. En nuestro caso una de ellas será el `coste`, definido como el número de acciones necesario para llegar hasta esa casilla:

```
1         double coste_siguiente = coste_hasta_ahora[actual] +
              ↪ acciones.size();
```

Antes de añadir una `PosicionCuadricula` a nuestro recorrido debemos comprobar si

- Esta `PosicionCuadricula` tiene un coste menor a otra que ya tengamos.
- No hemos pasado nunca por esta `PosicionCuadricula`.

```
1         bool posicion_no_explorada = coste_hasta_ahora.find(
              ↪ siguiente_pos)
2           == coste_hasta_ahora.end();
3         bool coste_menor = coste_siguiente < coste_hasta_ahora[
              ↪ siguiente_pos];
```

Realizamos las comprobaciones y añadimos la `PosicionCuadricula` al recorrido, indicando la `PosicionCuadricula` anterior, las acciones requeridas para llegar hasta ella y la orientación final al llegar.

```
1         if (posicion_no_explorada || coste_menor) {
2
3           coste_hasta_ahora[siguiente_pos] = coste_siguiente;
4
5           double prioridad = coste_siguiente +
6             Heuristica(siguiente_pos, destino);
7           frontera.insertar(siguiente_pos, prioridad);
8
9           recorrido[siguiente_pos] = make_tuple(actual, acciones,
                ↪ orientacion_siguiente);
10        }
```

Funciones ya existentes

A continuación se describe la Implementación de las funciones que se proporcionaron con el resto del código del programa y que teníamos que completar.

Función `PathFinding`

Es la función que se encarga de encontrar el recorrido desde la posición actual hasta el destino.

```
1  bool ComportamientoJugador::pathFinding(const estado &origen,
       ↪ const estado &destino, list<Action> &plan) {
2
3    plan.clear();
4
5    map<PosicionCuadricula,
6      tuple<PosicionCuadricula,
7      list<Action>,
8      int>> recorrido;
9
10   PosicionCuadricula pos_origen = {origen.fila, origen.columna};
11   PosicionCuadricula pos_destino = {destino.fila, destino.columna
       ↪ };
12
13   AEstrella(pos_origen, pos_destino, recorrido);
14
15   PosicionCuadricula pos = pos_destino;
16
17   int n_acciones = 0;
18   int n_pasos = 0;
19
20   // Vemos el recorrido desde el destino hasta el origen y vamos
       ↪ añadiendo
21   // las acciones al plan de movimiento
22   while (pos != pos_origen) {
23
24     n_pasos++;
25
26     list<Action> acciones = get<1>(recorrido[pos]);
27
28     for (Action act : acciones) {
29       n_acciones++;
30       plan.push_front(act);
31     }
```

```
32
33      pos = get<0>(recorrido[pos]);
34
35    }
36
37    if (!plan.empty()) {
38      PintaPlan(plan);
39      VisualizaPlan(origen, plan);
40      cout << "n_acciones: " << n_acciones << endl;
41      cout << "n_pasos: " << n_pasos << endl;
42      return true;
43    }
44
45    return false;
46  }
```

Función `think`

Esta función se encarga de decidir la acción que debe realizar el personaje. Si no hay plan, pide a `PathFinding` que lo cree; lo mismo ocurre si el destino ha cambiado. Si hay un plan procede con él, excepto si hay un aldeano delante, parándose hasta que éste se mueva. En cualquier caso, actualiza la brújula del personaje según las acciones realizadas.

```
1  Action ComportamientoJugador::think(Sensores sensores) {
2
3      if (sensores.mensajeF != -1) {
4          fil = sensores.mensajeF;
5          col = sensores.mensajeC;
6      }
7
8      if (hay_plan && (sensores.destinoF != destino.fila || sensores
           ↪ .destinoC != destino.columna)) {
9          hay_plan = false;
10     }
11
12     if (!hay_plan) {
13         estado origen = {fil, col, brujula};
14         destino = {sensores.destinoF, sensores.destinoC, 0};
15         hay_plan = pathFinding(origen, destino, plan);
16     }
17
18     Action sig_action;
```

```
19
20      if (hay_plan && plan.size() > 0) {
21          if (sensores.superficie[2] == 'a') {
22              sig_action = actIDLE;
23              PintaPlan(plan);
24          } else {
25              sig_action = plan.front();
26              plan.erase(plan.begin());
27          }
28      } else {
29          sig_action = actIDLE;
30      }
31
32      // Actualizamos las variables de estado del personaje
33
34      switch(sig_action) {
35          case actTURN_R: brujula = (brujula + 1) % 4; break;
36          case actTURN_L: brujula = (brujula + 3) % 4; break;
37          case actFORWARD:
38              switch(brujula) {
39                  case 0: fil--; break;
40                  case 1: col++; break;
41                  case 2: fil++; break;
42                  case 3: col--; break;
43              }
44      }
45
46    return sig_action;
47  }
```

2. Implementación del *nivel 3*

Funciones de apoyo

Para implementar el *nivel 3* he reconvertido algunas de las funciones usadas en los niveles anteriores, creando `PosicionLocalCorrecta`, `PosicionLocalAtravesable`, `ObtenerVecinosLocales`, `AEstrellaLocal` y `pathFindingLocal`, que realizan el mismo trabajo que las otras versiones pero utilizando un mapa *local* que representa únicamente la visión de nuestro personaje.

La función `Deambular()` se utiliza cuando nuestro personaje no conoce su posición en el mundo. Escanea los sensores del personaje y si encuentra un *PK*, lo

almacena en una variable. Devuelve una acción al azar.

Función `ActualizarMapa`

Esta función se encarga de escribir en la variable `mapaResultado` el contenido de los sensores, una vez que el personaje sepa la posición en la que se encuentra.

Funciones ya definidas

He modificado el funcionamiento de la función `think()` para adaptarla a las peculiaridades del *nivel 3*. Si el personaje no conoce su posición, deambulará hasta que aviste un PK, en cuyo caso se dirigirá hacia él. Una vez se encuentre en el PK, llamará a `pathFindingLocal()` para que trace un camino a él.

Al alcanzar el PK conoceremos las coordenadas de nuestro personaje, por lo que ya podemos utilizar la función `AEstrella`, que tratará las casillas `?` como atravesables. Puesto que el algoritmo se ejecuta cada vez que se llama a `think()`, el camino irá cambiando conforme nuestro personaje reciba información del entorno.