UNIVERSIDAD DE GRANADA

Memoria Práctica 2: Agentes Reactivos/Deliberativos

Inteligencia Artificial Grupo C2
Amador Carmona Mendez
amadorcm@correo.ugr.es

Nivel 0: Demo.

He realizado el tutorial, y tras realizarlo funciona correctamente. Analizo la implementación del algoritmo dado puesto que utilizare su estructura para el siguiente nivel.
Implementación:
En la parte inicial del método tenemos dos contenedores que almacenan los nodos cerrados y abiertos, para los cerrados se ha utilizado un set que almacena toda la estructura de nodos y para los abiertos se utiliza una pila para ir guardando el nodo padre y así poder recorrerlo en profundidad.

```
199  bool ComportamientoJugador::pathFinding_Profundidad(const estado &origen, const estado &destino, list<Action> &plan)
200  {
201      // Borro la lista
202      cout << "Calculando plan\n";
203      plan.clear();
204      set<estado, ComparaEstados> Cerrados; // Lista de Cerrados
205      stack<nodo> Abiertos;                 // Lista de Abiertos
206
207      nodo current;
208      current.st = origen;
209      current.secuencia.empty();
210
211      Abiertos.push(current);
212
213      while (!Abiertos.empty() and (current.st.fila != destino.fila or current.st.columna != destino.columna))
214      {
215
216          Abiertos.pop();
217          Cerrados.insert(current.st);
218
219          // Generar descendiente de girar a la derecha 90 grados
220          nodo hijoTurnR = current;
221          hijoTurnR.st.orientacion = (hijoTurnR.st.orientacion + 2) % 8;
222          if (Cerrados.find(hijoTurnR.st) == Cerrados.end())
223          {
224              hijoTurnR.secuencia.push_back(actTURN_R);
225              Abiertos.push(hijoTurnR);
226          }
```

Y así se ha tomado el siguiente valor de Abiertos:

```
265
266             // Tomo el siguiente valor de la Abiertos
267             if (!Abiertos.empty())
268             {
269                 current = Abiertos.top();
270             }
271         }
272
```

Nivel 1: Búsqueda en Anchura.

Algoritmo: El algoritmo de búsqueda en anchura es un tanto parecido a la hora de implementarlo que el algoritmo dado en el nivel 0 de búsqueda en profundidad.
He copiado y pegado el código del algoritmo de búsqueda en profundidad para tener como base la expansión de los hijos, y he modificado las estructuras y el código para cambiarlo al algoritmo en anchura. La estructura que guardaba los nodos abiertos pasa a ser una cola para poder recorrer así los nodos por niveles, los nodos cerrados sigue siendo un set, ya que almacena el resto de los nodos, y para avanzar al siguiente nivel al tener una cola debemos hacer un .front() a la cola de abiertos es decir pasar al siguiente nivel de nodos.

Implementacion:
En la parte inicial del método como he dicho anteriormente, hemos cambiado el contenedor que almacena los nodos abiertos, por una cola para poder recorrerlo por nivel.

```
413  bool ComportamientoJugador::pathFinding_Anchura(const estado &origen, const estado &destino, list<Action> &plan)
414  {
415      // Borro la lista
416      cout << "Calculando plan\n";
417      plan.clear();
418      set<estado, ComparaEstados> Cerrados; // Lista de Cerrados
419      queue<nodo> Abiertos;                 // Lista de Abiertos
420
421      nodo current;
422      current.st = origen;
423      current.secuencia.empty();
424
425      Abiertos.push(current);
```

La parte central del método, donde tenemos el bucle que recorre toda la estructura de nodos abiertos y mira si hemos llegado al final de los nodos. Luego en cada if vamos expandiendo a los siguientes nodos dependiendo de la dirección que tome el hijo, aquí solo indico como es un if ya que todos son iguales cambiando las direcciones.

```
426
427      while (!Abiertos.empty() and (current.st.fila != destino.fila or current.st.columna != destino.columna))
428      {
429
430          Abiertos.pop();
431          Cerrados.insert(current.st);
432
433          // Generar descendiente de girar a la derecha 90 grados
434          nodo hijoTurnR = current;
435          hijoTurnR.st.orientacion = (hijoTurnR.st.orientacion + 2) % 8;
436          if (Cerrados.find(hijoTurnR.st) == Cerrados.end())
437          {
438              hijoTurnR.secuencia.push_back(actTURN_R);
439              Abiertos.push(hijoTurnR);
440          }
441
```

Y por último aquí tomamos el siguiente nodo abierto:

```
479
480          // Tomo el siguiente valor de la Abiertos
481          if (!Abiertos.empty())
482          {
483              current = Abiertos.front();
484          }
485      }
486
```

En este nivel no hemos necesitado de funciones auxiliares, es decir no hemos implementado ninguna función más de las que vienen en la práctica para desarrollarlo.

Nivel 2: Algoritmo de búsqueda Costo Uniforme.

Algoritmo: Para ese nivel utilizaremos el Algoritmo de Costo Uniforme, para este algoritmo necesitamos tener dos contenedores para los nodos, como en los niveles anteriores uno al que llamaremos cerrados que será un set como en los niveles anteriores y una cola con prioridad en el que almacenemos los nodos con menor coste posible hasta llegar a la solución.
Como en el nivel anterior he cogido el código del algoritmo del nivel 1 como base para implementar este.
Implementación:
Estructuras auxiliares: Para este nivel hemos tenido que desarrollar estructuras y funciones auxiliares que nos ayudasen con los cálculos de coste de batería y comprobaciones de casillas para que el código fuera sencillo y no muy lioso.

Aqui vemos que hemos sobrecargado el comparador < para la cola con prioridad que utilizaremos en la implementación del algoritmo, y también para la ordenación del set hemos sobrecargado teniendo en cuenta los parámetros bikini y zapatillas es decir si nuestro muñeco lleva bikini o zapatillas.

```
//Estructura para comparar el costo de los nodos
struct compCosto{
    bool operator()(const nodo &n1,const nodo &n2){
        return n2 < n1;
    }
};
struct ComparaEstados{
    bool operator()(const estado &a, const estado &n) const
    {
        if ((a.fila > n.fila) or (a.fila == n.fila and a.columna > n.columna) or
            (a.fila == n.fila and a.columna == n.columna and a.orientacion > n.orientacion))
            return true;
        else
            return false;
    }
};
struct ComparaEstadosBateria{
    bool operator()(const estado &a, const estado &n) const
    {
        if ((a.fila > n.fila) or (a.fila == n.fila and a.columna > n.columna) or
            (a.fila == n.fila and a.columna == n.columna and a.orientacion > n.orientacion) or
            (a.fila == n.fila and a.columna == n.columna and a.orientacion == n.orientacion and a.zapatillas > n.zapatillas) or
            (a.fila == n.fila and a.columna == n.columna and a.orientacion == n.orientacion and a.zapatillas > n.zapatillas and a.bikini > n.bikini))
            return true;
        else
            return false;
    }
};
```

La estructura nodo finalmente la hemos alojado en el .hpp y hemos añadido los parámetros que necesitábamos, zapatillas, bikini y costeUni que se refiere al coste uniforme total, también hemos sobrecargado el operador < para que funcione la sobrecarga anterior llama comCosto.

```
struct nodo{
  estado st;
  list<Action> secuencia;
  int costeUni;
  bool bikini;
  bool zapatillas;
  bool operator<(const nodo &n) const{
    return this->costeUni<n.costeUni;}
};
```

Aquí tenemos el inicio y el final de la función que calcula el coste de una casilla según los parámetros del guión, como era demasiado larga no la he incluido entera ya que era una consecución de if else dependiendo de la casilla y de la acción se le asigna un coste y al final se devuelve dicho coste.

```
//devuelve lo que cuesta cada casilla
int ComportamientoJugador::costeCasilla(nodo &n,const Action &accion){
    char flag = mapaResultado[n.st.fila][n.st.columna];
    int coste = 0;
    if(accion == actIDLE){
        return coste;
    }else{
        if(accion==actWHEREIS){
            coste=200;
        }else if(accion==actFORWARD){
            if(flag=='A'){
                coste=200;
                if(n.bikini){
                    coste-=10;
                }
            }else if(flag=='B'){
                coste=100;
                if(n.zapatillas){
                    coste-=15;
                }
            }else if(flag=='T'){
                coste=2;
            }else{
                coste=1;
            }
        }else if(accion==actTURN_L){
            if(flag=='A'){
                coste=500;
                if(n.bikini){
                    coste-=5;
                }
            }else if(flag=='B'){
                coste=3;
                if(n.zapatillas){
                    coste-=1;
                }
            }else if(flag=='T'){
                coste=2;
            }else{
                coste=1;
            }
        }else if(accion==actTURN_R){
```

```
            }
        }else if(accion==actSEMITURN_L){
            if(flag=='A'){
                coste=300;
                if(n.bikini){
                    coste-=2;
                }
            }else if(flag=='B'){
                coste=3;
                if(n.zapatillas){
                    coste-=1;
                }
            }else if(flag=='T'){
                coste=2;
            }else{
                coste=1;
            }
        }else if(accion==actSEMITURN_R){
            if(flag=='A'){
                coste=300;
                if(n.bikini){
                    coste-=2;
                }

            }else if(flag=='B'){
                coste=3;
                if(n.zapatillas){
                    coste-=2;
                }
            }else if(flag=='T'){
                coste=2;
            }else{
                coste=1;
            }
        }
    }
    return coste;

}
```

Y aqui la funcion que detecta qué tipo de casilla es y dependiendo de si es una casilla especial cambia los atributos que tiene que cambiar, es decir si lleva bikini pone el bikini a true y si lleva zapatillas igual, como en el guión se indica que solo puede llevar uno puesto si tiene uno y cae en una casilla se lo cambia al objeto de la nueva casilla.

```
726  void ComportamientoJugador::casillaEspecial(estado &st){
727      char flag = mapaResultado[st.fila][st.columna];
728      if(flag == 'K' and !st.bikini){
729          st.bikini = true;
730          st.zapatillas = false;
731      }else if(flag == 'D' and !st.zapatillas){
732          st.zapatillas = true;
733          st.bikini = false;
734      }
735  }
```

Estructura del algoritmo: En la parte inicial tenemos la declaración de contenedores que utilizaremos para los nodos, en este caso un set para los nodos cerrados donde utilizaremos la sobrecarga de la estructura de compararEstadosBateria, para ordenar el set, y una cola con prioridad donde se utiliza la sobrecarga de la estructura para compara el costo uniforme de los nodos, inicialización de las variables comprobamos si la primera casilla es especial y añadimos a abiertos el nodo inicial, entramos en el bucle que tiene la misma condición de parada que los algoritmos anteriores y quitamos el nodo que vamos a expandir de abiertos.

```
518  ****************************************************************************************/
519  bool ComportamientoJugador::pathFinding_CostoUniforme(const estado &origen, const estado &destino, list<Action> &plan)
520  {
521      // Borro la lista
522      cout << "Calculando plan\n";
523      plan.clear();
524      set<estado,ComparaEstadosBateria> Cerrados;
525      priority_queue<nodo,vector<nodo>,compCosto> Abiertos;                // Lista de Abiertos
526
527      nodo current;
528      current.st = origen;
529      current.costeUni=0;
530      current.bikini=false;
531      current.zapatillas=false;
532      current.secuencia.empty();
533
534      casillaEspecial(current.st);
535      int costo=costeCasilla(current, actIDLE);
536      Abiertos.push(current);
537      while (!Abiertos.empty() and (current.st.fila!=destino.fila or current.st.columna != destino.columna)){
538          //cout<<"Depurando1";
539          //eliminamos el nodo que vamos a expandir de los abuiertos
540          Abiertos.pop();
```

Expansión de los nodos sumando el coste al costo uniforme:

```
543          if(Cerrados.find(current.st) == Cerrados.end()){
544              //Almacenamos en cerrados el nuevo nodo y comprobamos si es una casilla especial
545              Cerrados.insert(current.st);
546              casillaEspecial(current.st);
547
548              //Expandimos los nodos
549              nodo hijoTurnR = current;
550              hijoTurnR.st.orientacion = (hijoTurnR.st.orientacion+ 2) % 8;
551              if(Cerrados.find(hijoTurnR.st) == Cerrados.end()){
552                  costo=0;
553                  costo=costeCasilla(hijoTurnR,actTURN_R);
554                  //Actualizamos el costo del descendiente
555                  hijoTurnR.costeUni = costo + current.costeUni;
556                  hijoTurnR.secuencia.push_back(actTURN_R);
557                  Abiertos.push(hijoTurnR);
558              }
```

Aquí tomamos el valor del siguiente nodo.

```
601             PintaPlan(current.secuencia);
602             //cout<<current.costeUni<<endl;
603             //Tomamos el siguiente valor de abiertos
604             if(!Abiertos.empty()){
605                 current = Abiertos.top();
606             }
607         }
```