MEMORIA INTELIGENCIA ARTIFICIAL

Agentes Reactivos/Deliberativos y los extraños mundos de Belkan

Trabajo realizado por Nikita Stetskiy

Índice

Introducción ***3***

Profundidad ***4***

Anchura ***5***

Coste Uniforme ***6***

Reto ***9***

Think ***11***

Resultados ***13***

Introducción

Esta práctica consiste en la implementación de un agente reactivo/deliberativo, en el cual este es capaz de percibir el ambiente y actuar considerando una representación de su entorno y de las consecuencias de sus acciones. En nuestro caso es la búsqueda de caminos mediante distintos algoritmos que veremos a continuación.

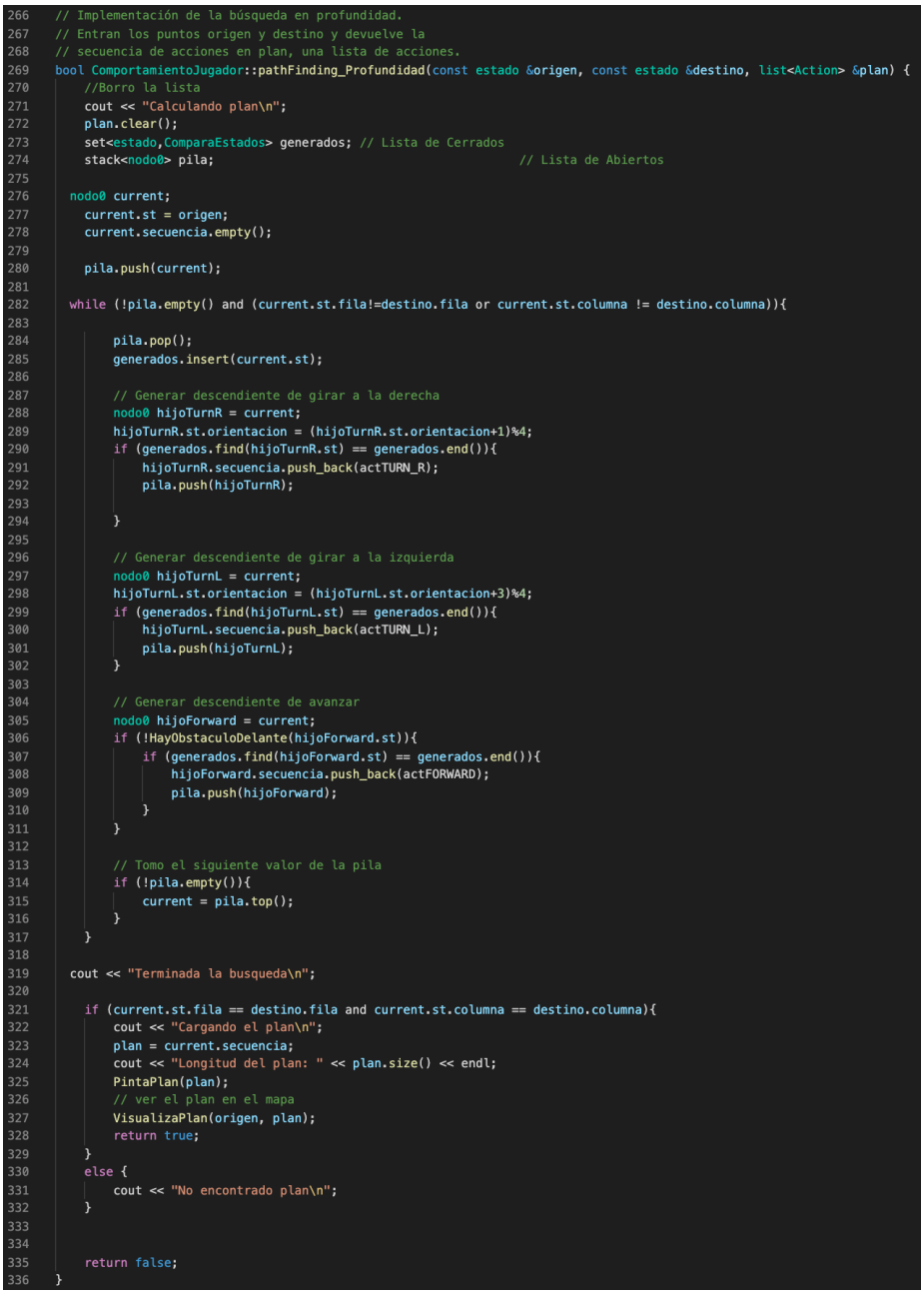
El objetivo de la practica es dotar de un comportamiento inteligente a nuestro agente, definiendo así las habilidades que corresponden a cada nivel del juego seleccionado. Los niveles se dividen en distintos comportamientos; 1 Profundidad, 2 Anchura, 3 Coste Uniforme y 4 Reto.

```
152  // Llama al algoritmo de busqueda que se usará en cada comportamiento del agente
153  // Level representa el comportamiento en el que fue iniciado el agente.
154  bool ComportamientoJugador::pathFinding (int level, const estado &origen, const estado &destino, list<Action> &plan){
155      switch (level){
156          case 1: cout << "Busqueda en profundad\n";
157                  return pathFinding_Profundidad(origen,destino,plan);
158                      break;
159          case 2: cout << "Busqueda en Anchura\n";
160                  return pathFinding_Anchura(origen,destino,plan);
161                      break;
162          case 3: cout << "Busqueda Costo Uniforme\n";
163                      return pathFinding_Uniforme(origen,destino,plan);
164                      break;
165          case 4: cout << "Busqueda para el reto\n";
166                      //return pathFinding_Uniforme(origen,destino,plan);
167                      return pathFinding_Reto(origen,destino,plan);
168                      break;
169      }
170      cout << "Comportamiento sin implementar\n";
171      return false;
172  }
```

En si, el programa se divide entre lo que sucede realmente y lo que piensa el agente. Es decir, lo real sucede en la función think y mediante esta se controla el funcionamiento del plan para llegar a la casilla objetivo. Por el otro lado está la función PathFinding que es la que se encarga de buscar el camino que hayamos elegido mediante los distintos algoritmos (Profundidad, Anchura, Coste Uniforme o Reto, que en mi caso he utilizado el algoritmo A *).

Profundidad

Se entiende por una búsqueda en Profundidad como un algoritmo que permite recorrer los nodos de manera ordenada, pero no uniforme. Es decir, que su funcionamiento se basa en la expansión de todos y cada uno de los nodos que localice, de forma recurrente, en un camino concreto.



```
// 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<nodo0> pila;                                    // Lista de Abiertos

  nodo0 current;
    current.st = origen;
    current.secuencia.empty();

    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
        nodo0 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
        nodo0 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
        nodo0 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;
}
```

Este algoritmo ha sido implementado anteriormente, pero su funcionamiento es sencillo de entender. Se utiliza una lista para los nodos abiertos y cerrados, en el caso de los abiertos su implementación se hace por medio de una pila. De ahí el inconveniente al sacar los nodos tarde más que la búsqueda en Anchura.

Anchura

Se entiende por búsqueda en Anchura un algoritmo para recorrer grafos, cuyo funcionamiento consiste en empezar por la raíz y explorar todos los hijos del nodo. Después se explora cada unos de los hijos de los hermanos y así sucesivamente hasta encontrar la solución.

```
341  bool ComportamientoJugador::pathFinding_Anchura(const estado &origen, const estado &destino, list<Action> &plan) {
342      //Borro la lista
343      cout << "Calculando plan\n";
344      plan.clear();
345      set<estado,ComparaEstados> generados; // Lista de Cerrados
346      queue<nodo1> cola;                                      // Lista de Abiertos
347
348    nodo1 current;
349      current.st = origen;
350      current.secuencia.empty();
351
352      cola.push(current);
353
354    while (!cola.empty() and (current.st.fila!=destino.fila or current.st.columna != destino.columna)){
355
356          cola.pop();
357          generados.insert(current.st);
358
359          // Generar descendiente de girar a la derecha
360          nodo1 hijoTurnR = current;
361          hijoTurnR.st.orientacion = (hijoTurnR.st.orientacion+1)%4;
362          if (generados.find(hijoTurnR.st) == generados.end()){
363              hijoTurnR.secuencia.push_back(actTURN_R);
364              cola.push(hijoTurnR);
365
366          }
367
368          // Generar descendiente de girar a la izquierda
369          nodo1 hijoTurnL = current;
370          hijoTurnL.st.orientacion = (hijoTurnL.st.orientacion+3)%4;
371          if (generados.find(hijoTurnL.st) == generados.end()){
372              hijoTurnL.secuencia.push_back(actTURN_L);
373              cola.push(hijoTurnL);
374          }
375
376          // Generar descendiente de avanzar
377          nodo1 hijoForward = current;
378          if (!HayObstaculoDelante(hijoForward.st)){
379              if (generados.find(hijoForward.st) == generados.end()){
380                  hijoForward.secuencia.push_back(actFORWARD);
381                  cola.push(hijoForward);
382              }
383          }
384
385          // Tomo el siguiente valor de la cola
386          if (!cola.empty()){
387              current = cola.front();
388              while(generados.find(current.st) != generados.end()  && !cola.empty()){
389                  cola.pop();
390                  current = cola.front();
391              }
392          }
393      }
394
395    cout << "Terminada la busqueda\n";
396
397      if (current.st.fila == destino.fila and current.st.columna == destino.columna){
398          cout << "Cargando el plan\n";
399          plan = current.secuencia;
400          cout << "Longitud del plan: " << plan.size() << endl;
401          PintaPlan(plan);
402          // ver el plan en el mapa
403          VisualizaPlan(origen, plan);
404          return true;
405      }
406      else {
407          cout << "No encontrado plan\n";
408      }
409
410
411      return false;
412  }
```

En este caso en vez de una pila, se utiliza una cola. Lo cual hace que la búsqueda de nodos funcione de una manera FIFO, es decir donde los nodos se insertan por un final y se extraen por otro (la pila funciona al revés). También he implementado al final una optimización en la cual se eliminan los nodos de la lista de cerrados ya explorados.

Coste Uniforme

La gran diferencia en este algoritmo es que se utiliza como su nombre indica, costes para recorrer sobre grafos el camino de costo mínimo entre un nodo raíz y un nodo destino. La búsqueda comienza por el nodo raíz y continúa visitando el siguiente nodo que tiene menor costo total desde la raíz. Los nodos son visitados de esta manera hasta que el nodo destino es alcanzado.

```
445  bool ComportamientoJugador::pathFinding_Uniforme(const estado &origen, const estado &destino, list<Action> &plan) {
446     //Borro la lista
447     cout << "Calculando plan\n";
448     plan.clear();
449     set<estado,ComparaEstadosAbsoluto> generados; // Lista de Cerrados
450     priority_queue<nodo2, vector<nodo2>, CompararCoste> cola;                                        // Lista de Abiertos
451
452   nodo2 current;
453     current.st = origen;
454     current.secuencia.empty();
455     current.coste = 0;
456     current.bikini = bikini;
457     current.zapatillas = zapatillas;
458
459     cola.push(current);
460
461   while (!cola.empty() and (current.st.fila!=destino.fila or current.st.columna != destino.columna)){
462
463         cola.pop();
464
465         if(mapaResultado[current.st.fila][current.st.columna] == 'K'){
466             current.bikini = true;
467             current.st.bikini = true;
468         }
469
470         if(mapaResultado[current.st.fila][current.st.columna] == 'D'){
471             current.zapatillas = true;
472             current.st.zapatillas = true;
473         }
474
475         generados.insert(current.st);
476
477         // Generar descendiente de girar a la derecha
478         nodo2 hijoTurnR = current;
479         hijoTurnR.st.orientacion = (hijoTurnR.st.orientacion+1)%4;
480         if (generados.find(hijoTurnR.st) == generados.end()){
481             hijoTurnR.coste+=CalcularCoste(current.st, current.zapatillas, current.bikini);
482             hijoTurnR.secuencia.push_back(actTURN_R);
483             cola.push(hijoTurnR);
484         }
485
486         // Generar descendiente de girar a la izquierda
487         nodo2 hijoTurnL = current;
488         hijoTurnL.st.orientacion = (hijoTurnL.st.orientacion+3)%4;
489         if (generados.find(hijoTurnL.st) == generados.end()){
490             hijoTurnL.coste+=CalcularCoste(current.st, current.zapatillas, current.bikini);
491             hijoTurnL.secuencia.push_back(actTURN_L);
492             cola.push(hijoTurnL);
493         }
494
495         // Generar descendiente de avanzar
496         nodo2 hijoForward = current;
497         if (!HayObstaculoDelante(hijoForward.st)){
498             if (generados.find(hijoForward.st) == generados.end()){
499                 hijoForward.coste+=CalcularCoste(current.st, current.zapatillas, current.bikini);
500                 hijoForward.secuencia.push_back(actFORWARD);
501                 cola.push(hijoForward);
502             }
503         }
504
505         // Tomo el siguiente valor de la cola
506         if (!cola.empty()){
507             current = cola.top();
508             while(generados.find(current.st) != generados.end() && !cola.empty()){
509                 cola.pop();
510                 current = cola.top();
511             }
512         }
513     }
514
515   cout << "Terminada la busqueda\n";
```

En este algoritmo se ha tenido que utilizar nodos diferentes, en los cuales se identificaban por sus costes. También se han implementado structs como ComparaEstadosAbsoluto y CompararCoste, para la correcta implementación de las listas de abiertos y cerrados. Para el coste uniforme he utilizado una cola con prioridad.

```
420  int ComportamientoJugador::CalcularCoste(estado st, bool zapatillass, bool bikinii){
421      int cos = 1, z=0, b=0;
422
423      if(zapatillass)
424          z=45;
425      if(bikinii)
426          b=90;
427
428      switch (mapaResultado[st.fila][st.columna]){
429          case 'A' : cos = 100-b; break;
430          case 'B' : cos = 50-z; break;
431          case 'T' : cos = 2; break;
432          case '?' : cos = 3; break;
433          case 'X' :
434              bateria.bikini = bikini;
435              bateria.zapatillas = zapatillas;
436              bateria.columna = st.columna;
437              bateria.fila = st.fila;
438              bateria.orientacion = st.orientacion;
439              cos = 0;
440              break;
441      }
442      return cos;
443  }
```

Para calcular coste se ha utilizado un switch con una funcionalidad muy sencilla, en la cual se comprobaba si se tenia zapatillas o bikini para disminuir el coste de las casillas correspondientes. Si la casilla es una batería, se guardaban su localización para el nivel 4. He utilizado una resta ya que es lo más optimo.

```
414  struct CompararCoste{
415      bool operator()(const nodo2& lhs, const nodo2& rhs){
416          return lhs.coste>rhs.coste;
417      }
418  };
```

Se ha utilizado un struct para poder ordenar la cola con prioridad, el cual de retorna true si el nodo “lhs” es mayor que “rhs”.

```
243  struct ComparaEstados{
244      bool operator()(const estado &a, const estado &n) const{
245          if ((a.fila > n.fila) or (a.fila == n.fila and a.columna > n.columna) or
246            (a.fila == n.fila and a.columna == n.columna and a.orientacion > n.orientacion))
247              return true;
248          else
249              return false;
250      }
251  };
252
253  struct ComparaEstadosAbsoluto{
254      bool operator()(const estado &a, const estado &n) const{
255          if ((a.fila > n.fila) or (a.fila == n.fila and a.columna > n.columna) or
256            (a.fila == n.fila and a.columna == n.columna and a.orientacion > n.orientacion) or
257           (a.fila == n.fila and a.columna == n.columna and a.orientacion == n.orientacion && a.bikini > n.bikini) or
258         (a.fila == n.fila and a.columna == n.columna and a.orientacion == n.orientacion && a.bikini == n.bikini && a.zapatillas > n.zapatillas))
259              return true;
260          else
261              return false;
262      }
263  };
```

El struct de ComparaEstadosAbsolutos tiene la misma función que CompararCoste, es decir establecer un orden de prioridad. Aquí se añade la comprobación de bikini y zapatillas al anterior struct ComparaEstados.

```
227   struct nodo2{
228       estado st;
229       int coste;
230       bool zapatillas;
231       bool bikini;
232       list<Action> secuencia;
233   };
```

Para la correcta implementación de la cola con prioridad, a los nodos se les ha añadido el atributo de coste y el bool de zapatillas y bikini. El bool se inicia al principio conforme a los atributos que tiene declarados en las variables de estado.

```
50    private:
51      // Declarar Variables de Estado
52      int fil, col, brujula;
53      int cont=0;
54      int aux=1;
55      estado actual, destino, bateria;
56      list<Action> plan;
57      bool hayplan, zapatillas=false, bikini=false, cargandoBateria=false;;
58
59      Action ultimaAccion;
60      // Métodos privados de la clase
61      bool pathFinding(int level, const estado &origen, const estado &destino, list<Action> &plan);
62      bool pathFinding_Profundidad(const estado &origen, const estado &destino, list<Action> &plan);
63      bool pathFinding_Anchura(const estado &origen, const estado &destino, list<Action> &plan);
64      bool pathFinding_Uniforme(const estado &origen, const estado &destino, list<Action> &plan);
65      bool pathFinding_Reto(const estado &origen, const estado &destino, list<Action> &plan);
66
67      int DistanciaAbsoluta(const estado &st1, const estado &st2);
68      void guardarVisitado(Sensores sensores);
69      int CalcularCoste(estado st, bool zapatillas, bool bikini);
70      void PintaPlan(list<Action> plan);
71      bool HayObstaculoDelante(estado &st);
```

Al comprobar un nodo que pasa por zapatillas o bikini, se declara como true el bool correspondiente en el estado y en el atributo de nodo. Cuando el agente pase realmente por la casilla (este cambia el atributo privado mediante la función correspondiente de think).

```
 8   struct estado {
 9     int fila;
10     int columna;
11     int orientacion;
12     bool bikini = false;
13     bool zapatillas = false;
14   };
```

Los bools de struct de estado existe para CompararEstados, como anteriormente se ha explicado. Es decir, que si se llegan a cambiar los estados, se le asignará diferente orden de prioridad.

Reto

La diferencia entre este algoritmo y el anterior, es la rapidez de calculo del camino. Ya que como he podido comprobar posteriormente en los resultados, la diferencia es mínima.

Así, el algoritmo A* utiliza una función de evaluación:

$$f(n) = g(n) + h'(n)$$

```
// Voy a utilizar el algoritmo A * (estrella)
bool ComportamientoJugador::pathFinding_Reto(const estado &origen, const estado &destino, list<Action> &plan) {
    //Borro la lista
    cout << "Calculando plan\n";
    plan.clear();
    set<estado,ComparaEstadosAbsoluto> generados; // Lista de Cerrados
    priority_queue<nodo3, vector<nodo3>, CompararCosteAbsoluto> cola;                                    // Lista de Abiertos

    int costeG, costeH;

  nodo3 current;
    current.st = origen;
    current.secuencia.empty();
    current.costeG = 0;
    current.costeH = 0;
    current.costeF = 0;
    current.bikini = bikini;
    current.zapatillas = zapatillas;

    cola.push(current);

  while (!cola.empty() and (current.st.fila!=destino.fila or current.st.columna != destino.columna)){

        cola.pop();

        if(mapaResultado[current.st.fila][current.st.columna] == 'K'){
            current.bikini = true;
            current.st.bikini = true;
        }

        if(mapaResultado[current.st.fila][current.st.columna] == 'D'){
            current.zapatillas = true;
            current.st.zapatillas = true;
        }

        generados.insert(current.st);

        // Generar descendiente de girar a la derecha
        nodo3 hijoTurnR = current;
        hijoTurnR.st.orientacion = (hijoTurnR.st.orientacion+1)%4;
        if (generados.find(hijoTurnR.st) == generados.end()){
            hijoTurnR.costeH+=DistanciaAbsoluta(hijoTurnR.st, current.st);
            hijoTurnR.costeG+=CalcularCoste(current.st, current.zapatillas, current.bikini);
            hijoTurnR.costeF+=hijoTurnR.costeH + hijoTurnR.costeG;
            hijoTurnR.secuencia.push_back(actTURN_R);
            cola.push(hijoTurnR);
        }

        // Generar descendiente de girar a la izquierda
        nodo3 hijoTurnL = current;
        hijoTurnL.st.orientacion = (hijoTurnL.st.orientacion+3)%4;
        if (generados.find(hijoTurnL.st) == generados.end()){
            hijoTurnL.costeH+=DistanciaAbsoluta(hijoTurnL.st, current.st);
            hijoTurnL.costeG+=CalcularCoste(current.st, current.zapatillas, current.bikini);
            hijoTurnL.costeF+=hijoTurnL.costeH + hijoTurnL.costeG;
            hijoTurnL.secuencia.push_back(actTURN_L);
            cola.push(hijoTurnL);
        }

        // Generar descendiente de avanzar
        nodo3 hijoForward = current;
        if (!HayObstaculoDelante(hijoForward.st)){
            if (generados.find(hijoForward.st) == generados.end()){
                hijoForward.costeH+=DistanciaAbsoluta(hijoForward.st, current.st);
                hijoForward.costeG+=CalcularCoste(current.st, current.zapatillas, current.bikini);
                hijoForward.costeF+=hijoForward.costeH + hijoForward.costeG;
                hijoForward.secuencia.push_back(actFORWARD);
                cola.push(hijoForward);
            }
        }
```

Donde g(n) representa el valor heurístico del nodo a evaluar desde el actual, n, hasta el final, y h(n), el coste real del camino recorrido para llegar a dicho nodo, n, desde el nodo inicial. A* mantiene dos estructuras de datos auxiliares, que podemos denominar *abiertos*, implementado como una cola de prioridad y *cerrados*, donde se guarda la información de los nodos que ya han sido visitados. En cada paso del algoritmo, se expande el nodo que esté primero en abiertos, y en caso de que no sea un nodo objetivo, calcula la f(n) de todos sus hijos, los inserta en abiertos, y pasa el nodo evaluado a cerrados.

```
235  struct nodo3{
236      estado st;
237      int costeG, costeH, costeF;
238      bool zapatillas;
239      bool bikini;
240      list<Action> secuencia;
241  };
```

Think

En esta parte del programa, corresponde a lo que sucede en la realidad alrededor del agente, donde se percibe mediante sensores.

```
// Este es el método principal que debe contener los 4 Comportamientos_Jugador
// que se piden en la práctica. Tiene como entrada la información de los
// sensores y devuelve la acción a realizar.
Action ComportamientoJugador::think(Sensores sensores) {

    // Mirar si se ha cambiado el destino

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

    // Calcular camino hasta el destino

    if(sensores.nivel != 4){
        if(!hayplan){
            actual.fila = sensores.posF;
            actual.columna = sensores.posC;
            actual.orientacion = sensores.sentido;
            destino.fila = sensores.destinoF;
            destino.columna = sensores.destinoC;
            hayplan = pathFinding(sensores.nivel, actual, destino, plan);
        }
    }

    Action sigAccion;

    if(sensores.nivel == 4){
        guardarVisitado(sensores);

        if (plan.size()>0 and (sensores.terreno[2]=='P' or
            sensores.terreno[2]=='M') and plan.front() == actFORWARD){
            hayplan = false;
        }

        if(cont<1000 && (sensores.terreno[2] == 'B' or sensores.terreno[2] == 'A') and plan.front() == actFORWARD)
            hayplan = false;

        if(sensores.terreno[0]=='K')
            bikini = true;

        if(sensores.terreno[0]=='D')
            zapatillas = true;

        if(!hayplan){
            actual.fila = sensores.posF;
            actual.columna = sensores.posC;
            actual.orientacion = sensores.sentido;
            destino.fila = sensores.destinoF;
            destino.columna = sensores.destinoC;
            hayplan = pathFinding(sensores.nivel, actual, destino, plan);
        }

        if(cont<3){
            sigAccion = actTURN_R;
            hayplan = false;
        }



        if(sensores.bateria < (500/aux) && bateria.columna != -1){
            actual.fila = sensores.posF;
            actual.columna = sensores.posC;
            actual.orientacion = sensores.sentido;
            hayplan = pathFinding(sensores.nivel, actual, bateria, plan);
            cargandoBateria = true;
        }

        if(sensores.bateria < (2700/aux) && cargandoBateria && sensores.posC == bateria.columna && sensores.posF == bateria.fila){
            sigAccion = actIDLE;
        }
        if(sensores.bateria >= (2700/aux) && cargandoBateria && sensores.posC == bateria.columna && sensores.posF == bateria.fila){
            hayplan = false;
            cargandoBateria = false;
            aux++;
        }
    }

    if(hayplan and plan.size()>0){ // Hay un plan no vacio
        if (sensores.superficie[2]=='a' and plan.front() == actFORWARD){
            sigAccion = actIDLE;
        }
        else{
            sigAccion = plan.front(); // tomamos la siguiente accion del hayplan
            plan.erase(plan.begin()); // eliminamos la accion de la lista de acciones
        }
    }

    cont++;

    return sigAccion;
}
```

Al principio compruebo si se cambia de objetivo, para que el plan se adapte. Después me organizo si se trata del nivel 4 o no. Si no lo es, calculo el plan y devuelvo la acción.

Si es nivel 4, guardo el mapa mediante los sensores, a través de la función guardarVisitado.

```
668  void ComportamientoJugador::guardarVisitado(Sensores sensores)
669  {
670      int fil=sensores.posF, col=sensores.posC;
671
672      mapaResultado[fil][col] = sensores.terreno[0];
673
674      switch (sensores.sentido)
675      {
676      case norte :
677          mapaResultado[fil-1][col-1] = sensores.terreno[1];
678          mapaResultado[fil-1][col] = sensores.terreno[2];
679          mapaResultado[fil-1][col+1] = sensores.terreno[3];
680
681          mapaResultado[fil-2][col-2] = sensores.terreno[4];
682          mapaResultado[fil-2][col-1] = sensores.terreno[5];
683          mapaResultado[fil-2][col] = sensores.terreno[6];
684          mapaResultado[fil-2][col+1] = sensores.terreno[7];
685          mapaResultado[fil-2][col+2] = sensores.terreno[8];
686
687          mapaResultado[fil-3][col-3] = sensores.terreno[9];
688          mapaResultado[fil-3][col-2] = sensores.terreno[10];
689          mapaResultado[fil-3][col-1] = sensores.terreno[11];
690          mapaResultado[fil-3][col] = sensores.terreno[12];
691          mapaResultado[fil-3][col+1] = sensores.terreno[13];
692          mapaResultado[fil-3][col+2] = sensores.terreno[14];
693          mapaResultado[fil-3][col+3] = sensores.terreno[15];
694          break;
695      case este :
696          mapaResultado[fil-1][col+1] = sensores.terreno[1];
697          mapaResultado[fil][col+1] = sensores.terreno[2];
```

En los casos generales, switch funciona más rápido por lo tanto mediante los sentidos del agente se van guardando las distintas casillas.

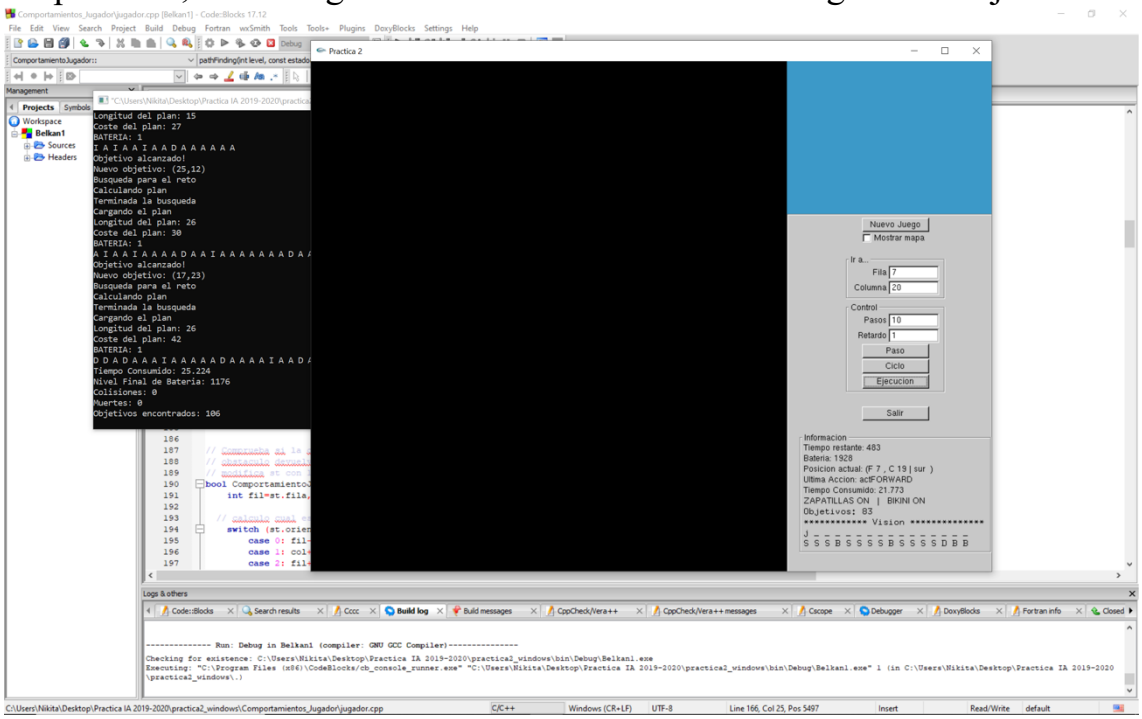
Después se comprueba, que si la casilla delante del agente es muro o pared, se cancele el plan. También lo hace con agua y bosque, las primeras mil iteraciones (ya que después conoce el mapa y puede llegar a hacer bucles raros). Con el aldeano solamente espera.

Después de esto calcula el plan, pero en los primeros tres turnos da una vuelta de 360 grados para ver mejor el mapa.

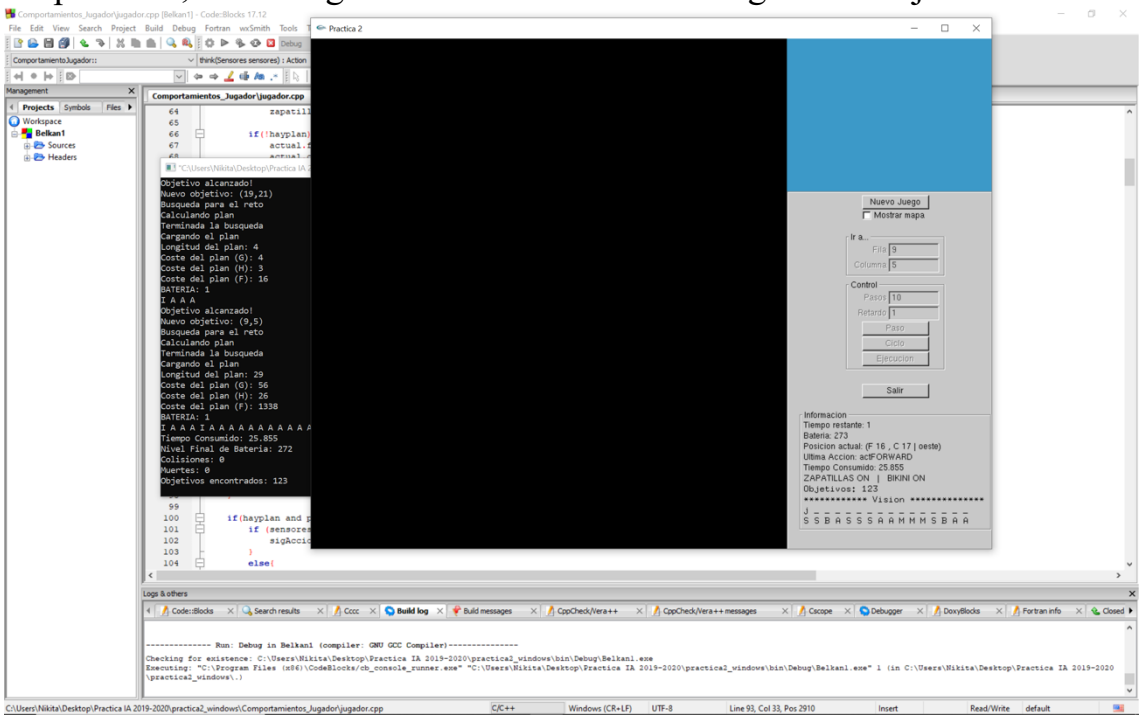
Al final si la batería esta por debajo de 500 mira a ver donde esta la batería y va a recargar hasta 2700. Después de esto, lo comprueba pero con los valores por la mitad.

Resultados

Mapa de 30, con el algoritmo de coste uniforme. Consigue 106 objetivos.



Mapa de 50, con el algoritmo de A estrella. Consigue 123 objetivos.



Mapa de 50, con el algoritmo de A estrella. Consigue 68 objetivos.