Los extraños mundos de Belkan

Aclaraciones nivel 4.

Para empezar tengo inicializado unas variables globales con los macros de c++ para poder determinar 2 cosas: **"Puedo cargar bateria", "tengo que tener en cuenta el límite de tiempo para pensar de 5 minutos (300 segundos)" , "tengo un máximo de movimientos" y "Debo rodear el aldeano".**

Cómo la mayoría de estas variables están desactivadas, obviaré la explicación por detrás de su implementación ya que a su vez, su nombre es explicativo de por si.

Empezemos por el **"if(!Parameters)"**. Es mi manera de poder inicializar ciertas variables del jugador ya que no sé en que parte del código se realiza una llamada al constructor del jugador y/o comportamiento suyo. (Sale de que las variables bool siempre suelen estar inicializadas a false).

```
#define CHARGE true //Por si queremos que tenga en cuenta cargar la bateria
#define NOTIMELIMIT false //Por si podríamos quitar el temporizador de think.
#define NOMOVEMENTLIMIT false
#define AVOIDVILLAGER false


void Cost(unsigned char casilla, nodoCoste& obj, const estado destino, double FactorOptimizacion);
// Este es el método principal que se piden en la practica.
// Tiene como entrada la información de los sensores y devuelve la acción a realizar.
// Para ver los distintos sensores mirar fichero "comportamiento.hpp"
Action ComportamientoJugador::think(Sensores sensores) {
    if(!Parameters){
        Parameters = true;
        THOUGHT = false;
        charging = false;
        aldeano = false;
        cout << "[MAPSIZE]: " <<  mapaResultado.size() << endl;
        //For faster calculus, además utiliza menos movimientos.
       // FP = (((double)mapaResultado.size())/100+0.05)/2; //Este valor se me ocurrió en un sueño.
        FP = 0; //Habría que probar valores para encontrar un óptimo global, pero me da pereza.
        MINBATTERY = (((double)mapaResultado.size())/10) * 100 + 300; //Valores que se me ocurrió en un sueño.
        THRESHOLD = ((((double)mapaResultado.size())) * 10)/2 + mapaResultado.size(); //Same ^^^
        MAXCHARGE = 3000; //Because yes
        total_cost = 0;
        ITER = 3000; //Is this cheating??... xd
        diff = 0;
        exp_rate = 1;
        cout << "[MINBATTERY]: " << MINBATTERY << endl;
        cout << "[THRESHOLD]: " << THRESHOLD << endl;
        cout << "[MAXCHARGE]: " << MAXCHARGE << endl;
        cout << "[FP]: " << FP << endl;
```

Dentro del condicional primeramente nos aseguramos de que no volveremos a entrar poniendo la variable inicial a true. Luego vamos a definir ciertas variables:

- **THOUGHT**: variable booleana que utilizo para ir calculando cuanto he tardado en calcular un cierto camino hacia un objetivo y/o batería.
- **charging**: variable booleana que sirve para saber que estoy cargando la batería.
- **aldeano**: me he chocado con un aldeano.
- **FP**: factor de peso de la distancia en el algoritmo A*.

- **MINBATTERY**: batería mínima que puede poseer el agente y seguir en busca de objetivos.
- **THRESHOLD**: Valor máximo al que puede cambiar el MINBATTERY.
- **MAXCHARGE**: valor máximo de batería a la que debe cargar el agente.
- **ITER**: Número de iteraciones en total.
- **diff:** variable double en la cuál voy a almacenar cuánto he tardado en calcular un camino. (Se auxilia de **OLD** que guarda el valor anterior de think).
- **exp_rate:** Cambió en el valor de las casillas desconocidas para decrementar las "ganas de explorar" del agente, para que a medida que avanza, cómo conocerá más el mapa, que tienda a utilizar los caminos ya conocidos para evitar encontrarse con sorpresas.

```
    }
    Action accion = actIDLE;
    if(!NOMOVEMENTLIMIT) {
        if (MINBATTERY > THRESHOLD) {
            MINBATTERY -= 0.001;
        } else {
            MINBATTERY = THRESHOLD;
        }
        ITER--;
        if (ITER <= 0)
            ITER = 0;
        MAXCHARGE = MINBATTERY + THRESHOLD + ITER;
        if (MAXCHARGE > 2980)
            MAXCHARGE = 2980;
        exp_rate += 0.0002;
        if(exp_rate >= 3){
            exp_rate = 3;
        }
    }
```

Luego como podemos observar a medida que avanza el agente por el nivel los valores determinados anteriormente para MINBATTERY/MAXCHARGE/exp_rate son alterados para prioritizar encontrar objetivos en vez de perder iteraciones cargando o tiempo buscando la batería cuándo no es del todo necesario. (Teniendo en cuenta el THRESHOLD y valores predeterminados).

```
ct is not loaded
            aldeano = false;
        }else
            rellenaMapa(actual,sensores.terreno);
        bool charger = NearByCharger();
        EsObjeto(actual);

        if(plan.front()==actFORWARD) {
            if(HayObstaculoDelante(temp)) {
                hay_plan = false;
                busca_objeto = false;
            }
            unsigned char verify = mapaResultado[temp.fila][temp.columna];
            //Cómo al final he optimizado el cálculo de los caminos, diff ya no sirve para este valor, pero lo dejo por s
            if((diff<40 or NOTIMELIMIT) and !charging){ //Si ha tardado demasiado en calcular el camino no renta volver a
                if (verify == 'B' and actual.objeto != 'D' and !alrededorCostoso(temp)) {
                    cout << "[QUITE EXPENSIVE INNIT?]" << endl;
                    hay_plan = false;
                    busca_objeto = false;
                } else if (verify == 'A' and actual.objeto != 'K' and !alrededorCostoso(temp)) {
                    cout << "[QUITE EXPENSIVE INNIT?]" << endl;
                    hay_plan = false;
                    busca_objeto = false;
                }
            }
            //Miramos una casilla hacia alante para evitar entrar en situaciones peligrosas para la batería.
            double next = LookAhead(actual);
            if(sensores.bateria-next<=MINBATTERY and !baterias.empty() and !charging){
                if(verify=='A' and actual.objeto!='K' or st_bat>=sensores.bateria){
                    cout << "[BETTER GET MY JUICE]:" << next << endl;
                    hay_plan = false;
                    busca_bateria = true;
```

Aquí es dónde realizo los cálculos del tiempo tardado en calcular el camino, además entra en juego los condicionales a tener en cuenta del agente, dónde esté tendrá en cuenta sus alrededores una vez rellenado el mapa, para evitar malgastar la batería y/o "suicidarse". Realizo llamada a tres funciones auxiliares que explicaré a continuación:

1. RellenarMapa.

```
void ComportamientoJugador::rellenaMapa(estado st,vector<unsigned char> terreno){
    int fil = st.fila;
    int col = st.columna;
    int cont = 0;
    int x = 1;
    switch(st.orientacion){
        case 0:
            for(int i=0;i<4;i++){
                for(int j=0;j<i+x;j++){
                    if(fil-i>=0 and fil-i<mapaResultado.size() and col-i+j>=0 and col-i+j<mapaResultado[fil-i].size())
                        mapaResultado[fil-i][col-i+j] = terreno[cont];
                    cont++;
                    if(i==0)
                        break;
                }
                x++;
            }
            break;
```

Dónde solo tenemos que distinguir entre las 4 posibles orientaciones y además realizar un juego con los índices para poder automatizar el cálculo de la visión y poder adaptarlo si incrementamos el radio de visión del jugador.

2. NearByCharger.

```
bool ComportamientoJugador::NearByCharger(){
    bool found = false;
    for(int i=-2;i<=2;i++) {
        for (int j = -2; j <= 2; j++) {
            if (actual.fila + i >= 0 and actual.fila + i < mapaResultado.size() and
                actual.columna + j >= 0 and actual.columna + j < mapaResultado[actual.fila + i].size()) {
                unsigned char temp = mapaResultado[actual.fila + i][actual.columna + j];
                if (!EsObstaculo(temp)) {
                    estado aux = actual;
                    if (!EsObjeto(aux)) {
                        if (temp == 'X') {
                            estado res;
                            res.fila = actual.fila+i;
                            res.columna = actual.columna + j;
                            baterias.insert(res);
                            found = true;
                            break;
                        }
                    }
                }
            }
        }
    }
    return found;
}
```

Aquí se realiza una búsqueda en **“mapaResultado”** una vez que hayamos calculado su valor con **“rellenaMapa”.** La búsqueda es de dos casilla alrededor debido a cómo funciona la “visión” del agente, que en máximo se observa dos casillas a sus lados. Pero se podría probar incrementar si fuera necesario.

3. EsObjeto.

```
bool  ComportamientoJugador::EsObjeto(estado &obj){
    if(mapaResultado[obj.fila][obj.columna]=='D') {
        obj.objeto = 'D';
        busca_objeto = true; //Saber que hemos elegido un camino para el cual necesitamos un objeto
        return true;
    }
    else if(mapaResultado[obj.fila][obj.columna]=='K') {
        obj.objeto = 'K';
        busca_objeto = true; //Saber que hemos elegido un camino para el cual necesitamos un objeto
        return true;
    }
    return false;
}
```

Determina si estamos una casilla objeto, y queda registrado en unos de los atributos del jugador que este camino posiblemente se haya cogido por necesidad de un objeto para llegar al objetivo.

Condicionales:

Si seguimos avanzando en el método “think” nos deparamos con varios condicionales:

El primer condicional refleja que si nos hemos encontrado con un obstáculo entonces debemos volver a calcular una ruta, ya que por esta misma no podemos seguir.

El segundo y el tercero, solamente se podrían cumplir si no tenemos limite de tiempo para calcular los movimientos o no hemos tardado más de 40 segundos en hallar un camino. Este segundo condicional tiene en cuenta de que tengamos el objeto necesario para cruzar el camino en caso de que sea uno de las casillas costosas como es el caso del agua y los árboles y, además, que haya alguna casilla “no costosa” en los alrededores, es decir, barro, tierra, batería u objeto.

A continuación se refleja el código de la función “alrededorCostoso” que sigue el mismo esquema que “NearByCharger”:

```
bool ComportamientoJugador::alrededorCostoso(estado temp){
    bool cost = true;
    for(int i=-1;i<=1;i++) {
        for (int j = -1; j <=1; j++) {
            if (temp.fila + i >= 0 and temp.fila + i < mapaResultado.size() and
                temp.columna + j >= 0 and temp.columna + j < mapaResultado[temp.fila + i].size()) {
                unsigned char var = mapaResultado[temp.fila + i][temp.columna + j];
                if (!EsObstaculo(var)) { //NOT 'P' not 'M'
                    estado aux = temp;
                    if (var != 'B' and var != 'A') {
                        cost = false;
                        break;
                    }
                }
            }
        }
    }
    return cost;
}
```

Volviendo al tercer condicional:

```
            //Miramos una casilla hacia alante para evitar entrar en situaciones peligrosas para la batería.
            double next = LookAhead(actual);
            if(sensores.bateria-next<=MINBATTERY and !baterias.empty() and !charging){
                if(verify=='A' and actual.objeto!='K' or st_bat>=sensores.bateria){
                    cout << "[BETTER GET MY JUICE]:" << next << endl;
                    hay_plan = false;
                    busca_bateria = true;
                    busca_objeto = false;
                }else if(verify=='B' and actual.objeto!='D' or st_bat>=sensores.bateria){
                    cout << "[BETTER GET MY JUICE]: " << next << endl;
                    hay_plan = false;
                    busca_bateria = true;
                    busca_objeto = false;
                }
            }
        }
```

Aquí miramos si al entrar en la siguiente casilla vamos a tener suficiente bateria para luego seguir con el juego. Para ello nos auxiliamos de “LookAhead” que actualiza el coste de bateria estimado una vez que hayamos actualizado “mapaResultado”.

LookAhead:

```
}
double ComportamientoJugador::LookAhead(estado st){
    int fil=st.fila, col=st.columna;
    double next_bat = 0;
    // calculo cual es la casilla de delante del agente
    for(int i=0;i<1;i++) {
        switch (st.orientacion) {
            case 0: fil--; break;
            case 1: col++; break;
            case 2: fil++; break;
            case 3: col--; break;
        }
        if (fil >= 0 and fil < mapaResultado.size() and col >= 0 and col < mapaResultado[fil].size()) {
            unsigned char var = mapaResultado[fil][col];
            nodoCoste obj;
            obj.st = st;
            Cost(var, obj, st, 0);
            next_bat += obj.bat;
            st_bat += next_bat;
        }
    }
    return next_bat;
}
```

Utiliza un switch para determinar la casilla que tenemos delante según la orientación, y a continuación si los valores son válidos llamamos a la función **“Cost” del nivel 2.**

La distancia a las baterias y a los objetivos utilizo la distancia euclidiana por como se expande los nodos del arból. Sin embargo, si aplicamos A* la distancia utilizada es una lineal.

DistBat:

```
double ComportamientoJugador::DistBat(estado temp){
    auto bat = baterias.begin();
    auto res = bat;
    min_distance = numeric_limits<double>::max();
    double distance = 0;
    int cont = -1;
    for(;bat!=baterias.end();++bat){
        cont++;
        int df =  abs(temp.fila - bat->fila);
        int dc =  abs(temp.columna - bat->columna);
        if(df==0) df = 1;
        if(dc==0) dc = 1;
        df = df*df; dc = dc * dc;
        distance = df + dc;
        if (distance <= min_distance) {
            index_bateria = cont;
            min_distance = distance;
            res = bat;
        }
    }
    return min_distance;
}
```

Lo demás es simplemente realizar comprobaciones de si hemos cargado, si hemos encontrado objetivos y si necesitamos ir a cargar. Según varios tests que hice y tras analizar las condiciones del juego he determinado los valores por defecto de varios parámetros. (EJ: Casi siempre es mejor ir a la batería se esta está a menos de unas 5 casillas u algo así para no perder tiempo luego calculando y/o moviéndome hacia ella dentro de poco si lo voy a hacer igualmente).

PD: La razones por la que he añadido “diff” y cosas para tener en cuenta el tiempo de cálculo es realmente por que antes mis cálculos de “costeUniforme”, “NObjetivos” y etc.. se realizaban muy despacio y tardaban realmente mucho. (120segs vs 30segs ahora). Y el error era mantener estados obsoletos en Abiertos. Se arregló fácilmente con:

```
    auto it = Abiertos.begin();
    while(Cerrados.find(current.st)!=Cerrados.end()){
        Abiertos.erase(it);
        it = Abiertos.begin();
        current = *it;
        if(Abiertos.empty())
            break;
    }

    Cerrados.insert(current.st);
```