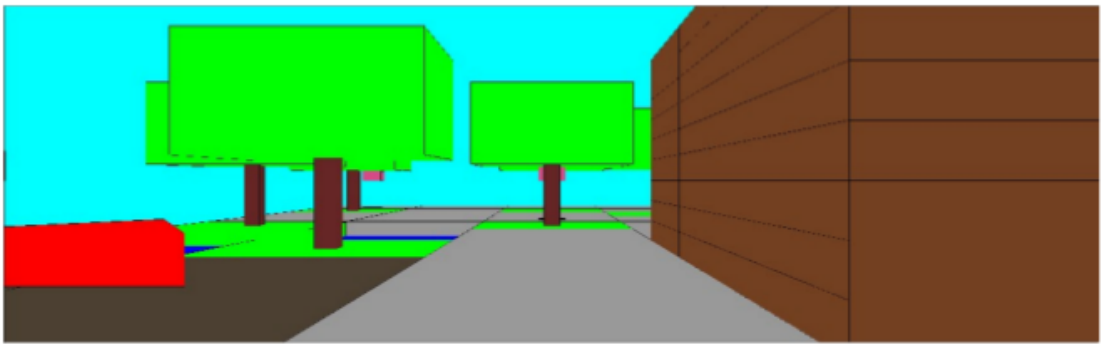
INTELIGENCIA ARTIFICIAL
PRÁCTICA 2



Agentes Reactivos/Deliberativos: los Extraños Mundos de BelKan

Memoria de la Práctica

Alumno: Daniel Bolaños Martínez
DNI: 76592621-E
DGIIM.

1. Objetivo

El objetivo de esta práctica es dotar de un comportamiento inteligente al personaje del juego **Los Extraños Mundos de Belkan** usando un agente reactivo/deliberativo para definir las habilidades que le permitan moverse dentro del juego según el nivel seleccionado y encontrar un número de objetivos.

La práctica se desarrolla en 3 niveles de dificultad en los que he trabajado los comportamientos que especificaré a continuación.

2. Niveles

Nivel 1

El objetivo de este nivel es llegar desde el origen del personaje hasta un objetivo seleccionado en un mapa desierto (sin aldeanos), utilizando algún algoritmo de los vistos en clase.

Para realizar tanto este nivel como el resto, he utilizado el *Algoritmo* A* para desarrollar la búsqueda del camino más corto.

En primer lugar he creado una clase *Nodo* para definir el comportamiento de cada casilla en el mapa, los atributos de la clase son los siguientes:

```
class Nodo{
    private:
        Nodo *padre;
        estado position;
        estado objetivo;
        Action accion;
        double costeG;
        double costeH;
};
```

La clase *Nodo* cuenta con un puntero al padre para saber de qué casilla se viene, con una posición en el mapa, el objetivo a llegar, la acción que se ha realizado para llegar a ese nodo y dos costes que determinarán el coste total.

El *costeG* que lo determinará la profundidad en el árbol de nodos y el *costeH* que lo determinará la heurística utilizada, en mi caso, la **Manhattan**.

```
double heuristica(){
    return abs(position.fila - objetivo.fila)+
    abs(position.columna - objetivo.columna);
}
```

He creado dos constructores de la clase *Nodo* uno para inicializar el nodo *Raíz* y otro para expandir los hijos de este.

Para el Nodo *Raíz* simplemente le pasamos el origen del personaje y el destino a alcanzar.

```
Nodo(estado origen, estado destino){
    padre = nullptr;
    position = origen;
    objetivo = destino;
    costeG = 0;
    costeH = heuristica();
    accion = actIDLE;
}
```

Para los hijos le paso una referencia al padre junto con la acción, que veremos más adelante como se expande:

Los hijos heredan el *costeG* del padre + 1 en profundidad, la posición del padre la cambian conforme a la acción que se realice y la heurística se calcula respecto a la nueva posición del Nodo hijo.

```
Nodo(Nodo *n, Action act){
    padre = n;
    position = n->position;
    objetivo = n->objetivo;
    costeG = n->costeG+1;
    costeH = heuristica();
    accion = act;
    if(accion == actFORWARD){
        if(position.orientacion==0)
            position.fila = position.fila-1;
        if(position.orientacion==1)
            position.columna = position.columna+1;
        if(position.orientacion==2)
            position.fila = position.fila+1;
        if(position.orientacion==3)
            position.columna = position.columna-1;
    }
    if(accion == actTURN_L)
        position.orientacion = ((position.orientacion-1)+4)%4;
    if(accion == actTURN_R)
        position.orientacion = (position.orientacion+1)%4;
}
```

Ahora explicaré mi programación de las funciones del *Algoritmo* A* basadas sencillamente en dos funciones *pathFinding* y *expandNodo*, que mostraré a continuación:

La función *pathFinding* que consta de los siguientes pasos:

1. Creamos el nodo *Raíz* con la posición de origen del personaje y la de destino y lo metemos en una *list* de **abiertos**
2. Expandimos el Nodo *Raíz*, esto quiere decir que sacamos el Nodo *Raíz* de la lista de abiertos, lo

metemos a una lista de **cerrados** y metemos sus tres hijos posibles en la lista de **abiertos**.
3. Seguimos el algoritmo expandiendo en cada paso el Nodo de la lista de abiertos con mejor *costeF* que es la suma del coste de la heurística más la profundidad (*costeG*) y la condición de parada se dará cuando lleguemos al nodo objetivo (*nodoFinal*).

```
bool pathFinding(const estado &origen,
    const estado &destino, list<Action> &plan) {
    Nodo nRoot(origen, destino);
    abiertos.push_back(nRoot);
    expandNodo(nRoot);
    bool fin = false;
    list<Nodo>::iterator pos;
    double mas_prometedor;

    while(!abiertos.empty() && !fin){
        pos = abiertos.begin();
        mas_prometedor = abiertos.front().getCostF();
        for(list<Nodo>::iterator it=abiertos.begin();
        it != abiertos.end() && !fin; ++it){
            if(mas_prometedor > it->getCostF()){
                pos = it;
                mas_prometedor = it->getCostF();
            }
        }
        if(pos->isFinal()){
            fin=true;
            PintaPlan(plan);
            backtrace(*pos);
            }

        if(!fin)
            expandNodo(*pos);
    }


    return fin;
}
```

Aquí podemos ver la función *expandNodo* que trabaja como he comentado anteriormente:

La utilidad de la función *isWalkable* es para ver si ese nodo es válido de caminar por encima en ese mapa y solo se verifica si se tiene que dar un paso hacia adelante, ya que con los giros, no se cambia de posición, solo de orientacion.

```
void expandNodo(Nodo &n){
    cerrados.push_back(n);
    borrarAbierto(n);

    Nodo hijo1(&(cerrados.back()), actFORWARD);
    Nodo hijo2(&(cerrados.back()), actTURN_L);
    Nodo hijo3(&(cerrados.back()), actTURN_R);

    if(hijo1.isWalkable(auxMapa) && !esCerrado(hijo1))
```

```
            abiertos.push_back(hijo1);

    if(!esCerrado(hijo2))
        abiertos.push_back(hijo2);

    if(!esCerrado(hijo3))
        abiertos.push_back(hijo3);
}
```

Finalmente una vez tenemos planificado el recorrido, hacemos un *backtrace* de las acciones de cada nodo, desde el Nodo Final hacía atrás, pasando por sus respectivos padres y llegando al Nodo *Raíz* y ejecutamos una a una las acciones hasta que se agoten. Todo esto se hará en la función *think*.

Al principio de la ejecución, la función *think* tendrá que guardar tanto el destino como el origen usando los sensores.

Para llevar la cuenta de las acciones utilizo estas funciones, que modifican la brújula, fila y columna dependiendo de las acciones realizadas en cada paso.

```
void actForward(int &f, int &c, int br){
    if(br==0)
        f--;
    if(br==1)
        c++;
    if(br==2)
        f++;
    if(br==3)
        c--;
}

void turnLeft(int &br){
    br = (--br+4)%4;
}

void turnRight(int &br){
    br = (++br)%4;
}
```

Nivel 2

El objetivo de este nivel es similar al del **nivel 1**, con la modificación de que hay aldeanos moviéndose que presentan obstáculos dinámicos, esto presenta una pequeña modificación en el código del **nivel 1** que explicaré brevemente.

Podríamos arreglar este nivel simplemente diciendo que cuando viese a un aldeano esperase sin hacer nada hasta que se fuese, pero he implementado un contador de paciencia para que si tarda más de esas acciones recalcule el camino usando *pathFinding*, tratando la casilla de aldeano como si de una casilla *noWalkable* se tratase.

Después de recalcular, pone la casilla a la misma que había anteriormente donde estaba el aldeano, para evitar crear un callejón sin salida.

```
if(sensores.superficie[2]=='a'){
    if(seg < ESPERA){
        action = actIDLE;
        seg = seg+1;
        hay_ald = true;
    }
    else{
        estado orig;
        orig.fila = fil;
        orig.columna = col;
        orig.orientacion = brujula;

        casillaDelante(auxMapa, ald);
        plan.clear();
        abiertos.clear();
        cerrados.clear();
        pathFinding(orig, destino, plan);
        auxMapa[ald.fila][ald.columna]=ald.casilla;
        seg = 0;
        hay_ald = false;
        action = plan.front();
    }
}
```

Nivel 3

El **nivel 3** es más complejo debido a que no tenemos visibilidad en el mapa y debemos localizarnos en el mismo gracias a los *PK* del mapa, este nivel lo he dividido en dos partes, la primera conseguir el *PK* y la siguiente recalcular los caminos para conseguir el máximo número de objetivos posibles.

Primero he creado una función que vaya rellenando con la información de los sensores cualquier mapa que le pases como parámetro, para buscar el *PK* he creado un mapa tipo *matrizAmpliada[200][200]* con todo *'?'* para simular al *mapaResultado* que tenemos disponible.

```
  addVisionMapa(sensor, matrizAmpliada, filG, colG, brujulaG);
```

Voy realizando movimientos diagonales y sobre todo dando prioridad a las esquinas que es donde predominan los *PK* para buscar uno y cuando lo tengo en los sensores alcanzarlo:

```
int puntoPK(Sensores &sensores){
    int pos;
    for(int i=0; i<16; i++){
        if(sensores.terreno[i] == 'K')
            pos = i;
    }
    return pos;
}
```

También realizo movimientos aleatorios por si pudiese estar un *PK* en esa zona pero no haberlo visto el personaje:

```
else if (contador < RANDOM){
    int cual = aleatorio(12);
    switch (cual) {
    case 1: action = actTURN_L; turnLeft(brujulaG); break;
    case 0: case 2: case 3: case 4:
    case 5: case 6: case 7: case 8: case 9: case 10: case 11:
        if (sensor.terreno[2] != 'P' && sensor.terreno[2]
        != 'B' && sensor.terreno[2] != 'A'
        && sensor.terreno[2] != 'M'
        && sensor.terreno[2] != 'D'
        && sensor.superficie[2]!='a'){
            action = actFORWARD;
            actForward(filG, colG, brujulaG);
        }
        else{
            action = actTURN_L; turnLeft(brujulaG);
        }
        break;
    }
```

Una vez que llegamos al *PK* hago un volcado de la matriz de 200x200 a la original de *mapaResultado* para aprovechar todo lo que se haya visto buscando el *PK*:

```
void trasladarMatriz(){
    int k=0;
    for(int i=abs(filG-fil); k < tam; i++){
        int l=0;
        for(int j=abs(colG-col); l < tam; j++){
            mapaResultado[k][l] = matrizAmpliada[i][j];
            ++l;
        }
        ++k;
    }
}
```

Finalmente, una vez encontrado y guardada mi posición sobre el *PK* me dispongo a ir al objetivo:

Cabe destacar que a las casillas con *'?'* les he asignado más coste para evitar que el personaje se vaya demasiado lejos del verdadero camino por creer que es más corto.

```
if(map[getFila()][getCol()]=='?')
    costeH = costeH+5*heuristica();
```

El programa recalculará en los siguientes casos:

1. La siguiente acción a realizar es ir hacia delante y hay un obstáculo físico que le impide pasar.

```
if(action==actFORWARD){
    if(sensores.terreno[2]=='A' || sensores.terreno[2]=='B'
    || sensores.terreno[2]=='P' || sensores.terreno[2]=='M'){
        crea = false;
        action = actIDLE;
    }
```

1. Hay un aldeano delante y ha superado el máximo de espera (código equivalente al del **nivel 2**).

Para que cambie el destino cuando consigue uno, he agregado a la función *think* lo siguiente:

```
if(destino.fila != sensores.destinoF && destino.columna
    != sensores.destinoC){
    destino.fila = sensores.destinoF;
    destino.columna = sensores.destinoC;
    destino.orientacion = 0;
    crea = false;
}
```