PRÁCTICA E1

INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE GRANADA

DAVID MUÑOZ SÁNCHEZ
07258819C Grupo B1

Comportamiento implementado

Se han implementado los niveles 1 y 2 de la práctica así como el tutorial.
El **nivel 1** consiste en encontrar el camino óptimo para llegar a un objetivo. Para ello se hace uso del algoritmo A*.
Previo a la implementación del algoritmo, se añade al struct estado dos booleanos para saber si tenemos bikini o zapatillas y, en adelante, las comprobaciones referidas al bikini o a las zapatillas se harán con el estado de un nodo.
En cuanto a los nodos, se ha creado un nuevo tipo de nodo llamado nodo_estrella, que, además de lo que tiene un nodo, se añade lo necesario para el algoritmo A*, una variable para el coste, otra para la heurística y otra para la f, que es la suma de las dos anteriores. Además, como en el algoritmo se hará uso de una cola con prioridad, se sobrecarga el operador <.

```
struct nodo_estrella
{
    estado st;
    list<Action> secuencia;
    int g; // coste de lo recorrido
    int h; // heuristica
    int f; // suma g y h

    bool operator<(const nodo_estrella &n) const
    {
        return this->f > n.f;
    }
};
```

Así mismo, se crea otra struct para comparar los estados de los no nodos estrella.

```
struct comparaEstadosEstrella
{
    bool operator()(const nodo_estrella &a, const nodo_estrella &n) const
    {
        if ((a.st.fila > n.st.fila) or (a.st.fila == n.st.fila and a.st.columna >
n.st.columna) or
                (a.st.fila == n.st.fila and a.st.columna == n.st.columna and
a.st.orientacion > n.st.orientacion) or
                (a.st.fila == n.st.fila and a.st.columna == n.st.columna and
a.st.orientacion == n.st.orientacion and a.st.bikini > n.st.bikini) or
                (a.st.fila == n.st.fila and a.st.columna == n.st.columna and
a.st.orientacion == n.st.orientacion and a.st.bikini == n.st.bikini and a.st.zapatillas
> n.st.zapatillas))
                return true;
        else
                return false;
    }
};
```

En cuanto a la heurística usada, teniendo en cuenta que no se puede usar la distancia Manhattan por no ser admisible, he usado una ligera modificación de esta que sí lo es y es que, en vez de devolver la suma de las distancias en vertical y horizontal, devuelvo la mayor de estas.

```
int ComportamientoJugador::heuristica(const estado &origen, const estado &destino)
{
	return max(abs(origen.fila - destino.fila),abs(origen.columna -
destino.columna));
}
```

Por último, el cálculo del coste, se hace con una función y se tiene en cuenta la casilla donde nos encontremos, si nuestro estado tiene o no bikini y la acción. Los distintos valores están ajustados a los propuestos en la tabla del guion de esta práctica.

En cuanto a la búsqueda A*, el código se encuentra en jugador.cpp (no se inserta aquí por el límite de extensión de 5 páginas para toda la memoria).
Para cada nodo, comprobamos si se trata de una casilla K o D, para posteriormente crear todos los posibles movimientos teniendo en cuenta los dos movimientos introducidos en esta práctica OVERTURN, que difieren de los de la ordinaria (TURN). Este nuevo movimiento, consiste en un giro de 135° en vez de 90°. Los descendientes deben tener actualizada toda su información y se añadirá a abiertos si no está, y en caso de que esté, si el nodo estrella es menor que el que ya está, según el operador que hemos sobrecargado.

Una vez generados todos los descendientes, current será el top de Abiertos, a no ser que abiertos siga teniendo nodos y que se encuentre el nuevo current en cerrados. En ese caso, quitamos el nodo de abiertos y cogemos el siguiente. La búsqueda termina cuando llegamos al nodo objetivo o cuando abiertos no tiene más nodos.

Para finalizar, comentar que he realizado las tres ejecuciones que se proponen en el fichero de información adicional de Prado y todas salen con la misma secuencia que la que se indica, excepto la segunda, pero el gasto de batería es el mismo.

El **nivel 2** es una modificación del primero en el sentido de que queremos hacer lo mismo pero esta vez con tres objetivos. Para ello, se declara en estado un vector de bool para marcar los objetivos visitados, y un entero para contar los objetivos ya alcanzados. Además, es necesario declarar un nuevo struct para comparar los estados ya que ahora hay que tener en cuenta el número de objetivos visitados.

```
struct comparaEstadosEstrella2
{
    bool operator()(const nodo_estrella &a, const nodo_estrella &n) const
    {
        if ((a.st.fila > n.st.fila) or (a.st.fila == n.st.fila and a.st.columna >
n.st.columna) or
               (a.st.fila == n.st.fila and a.st.columna == n.st.columna and
a.st.orientacion > n.st.orientacion) or
               (a.st.fila == n.st.fila and a.st.columna == n.st.columna and
a.st.orientacion == n.st.orientacion and a.st.bikini > n.st.bikini) or
               (a.st.fila == n.st.fila and a.st.columna == n.st.columna and
a.st.orientacion == n.st.orientacion and a.st.bikini == n.st.bikini and a.st.zapatillas
> n.st.zapatillas) or
               (a.st.fila == n.st.fila and a.st.columna == n.st.columna and
a.st.orientacion == n.st.orientacion and a.st.bikini == n.st.bikini and a.st.zapatillas
== n.st.zapatillas and a.st.obj_visitados > n.st.obj_visitados))
               return true;
        else
               return false;
    }
};
```

En cuanto a la heurística, he empleado una función que devuelve la mejor heurística comparando la heurística que ya teníamos desde origen a todos los objetivos.

```
int ComportamientoJugador::mejorHeuristica(const estado &origen, vector<estado>
destinos, vector<bool> visitados)
{
    int valor = 300000;

    for (int i = 0; i < numero_obj; i++)
    {
        if (!visitados[i] and valor >= heuristica(origen, destinos[i]))
        {
               valor = heuristica(origen, destinos[i]);
        }
    }

    return valor;
}
```

En cuanto al método de búsqueda en sí, es muy parecido al descrito anteriormente, solo que como argumento se le pasa un vector de tres objetivos en vez de uno solo.

Además, ahora hay que inicializar más aspectos del nodo y del estado.

```
nodo_estrella current;
    current.st = origen;
    current.secuencia.empty();
    current.g = 0;
```

```
    current.st.bikini = bikini;
    current.st.zapatillas = zapatillas;
    current.st.obj_visitados = 0;

    for (int j = 0; j < numero_obj; j++)
    {
        current.st.visitados.push_back(false);
    }

    current.h = mejorHeuristica(current.st, destino, current.st.visitados);
    current.f = current.g + current.h;
```

Lo único que cambia es que, antes de generar todos los descendientes, debemos comprobar si se ha alcanzado alguno de los objetivos.

```
for (int i = 0; i < numero_obj; i++)
        {
                if (current.st.fila == destino[i].fila and current.st.columna ==
destino[i].columna and !current.st.visitados[i])
                {
                        current.st.visitados[i] = true;
                        current.st.obj_visitados += 1;
                }
        }
```

Después, en un if, debemos comprobar si hemos alcanzado todos los objetivos.

```
if (numero_obj == current.st.obj_visitados)
        {
                end = true;
        }
```

En el else se introduce todo lo referido a la generación de descendientes, de igual forma que en la implementación de A* para un solo objetivo, pero teniendo en cuenta la nueva heurística.

Para finalizar, comentar, que este nivel no ha obtenido el comportamiento deseado. El agente traza un plan que atraviesa los tres objetivos, pero no es el óptimo, lo cual es probable que se deba a que la heurística que he utilizado no es admisible.