Memoria Práctica 2

Desarrollo de juegos con inteligencia artificial

Santiago Rubio Montero
Javier de las Peñas Férnandez

ÍNDICE

INTRODUCCIÓN

El objetivo de la práctica es implementar un agente inteligente cuyo comportamientosea aprendido por técnicas de machine learning. En concreto, se emplean técnicas de Q-Learning. Para ello se han creado dos archivos C# llamados “*Mytrainer*” en el cual se creará la tabla de exploración a medida que nuestro agente descubre el tablero para posteriormente exportarlo en formato CSV, y “*Mytester*” el cual recibirá la tabla con el aprendizaje y decidirá qué camino es el más óptimo para huir del enemigo.

TABLEQ

La clase TableQ consta de los siguientes atributos: numero de filas, que corresponden a las acciones (N, E, S y O), numero de columnas, que son todos los estados posibles, que corresponden a las posiciones que son walkable, es decir, 16, (N, E, S, O -> 4*4) y la posición relativa del enemigo respecto del agente (north/aligned/south , west/aligned/east). Además de esto se incluye una matriz de floats que corresponderá a los valores de la tabla Q, y un array con todos los estados posibles (16*9 = 144).

```
public int nRows;
public int nCols;
public float[,] value; //valores de la tabla Q
public State[] statesArray; //array de estados

1 referencia
public TableQ()
{
    // Tamaño de la tabla
    this.nRows = 4; // Acciones posibles
    this.nCols = (4 * 4) * (3 * 3); // Estados posibles
    this.value = new float[this.nRows, this.nCols]; // Matriz de valores, es decir tabla Q

}
```

También se implementa métodos para inicializar la matriz de valores y el array de estados.

```
1 referencia
public void InitializeTableQ()
{

    for (int i = 0; i < this.nRows; i++)
    {
        for (int j = 0; j < this.nCols; j++)
        {
            this.value[i, j] = 0.0f;
        }
    }

    this.statesArray = InitStatesArray(); //
}
```

```
1 referencia
public State[] InitStatesArray()
{
    int indice = 0;
    State[] states = new State[16 * 9]; // 16 combinaciones de walkableNeighbours y 9 combinaciones de enemyRelativePosition

    //Se inicializa como una tabla de verdad, haciendo todas las combinaciones posibles
    for (int i = 0; i < 16; i++)
    {
        bool[] walkableNeighbours = new bool[4];
        walkableNeighbours[0] = (i & 8) != 0; // North
        walkableNeighbours[2] = (i & 4) != 0; // South
        walkableNeighbours[1] = (i & 2) != 0; // East
        walkableNeighbours[3] = (i & 1) != 0; // West

        for (int j = -1; j <= 1; j++)
        {
            for (int k = -1; k <= 1; k++)
            {
                int[] enemyRelativePosition = new int[2] { j, k };
                states[indice] = new State(indice, walkableNeighbours, enemyRelativePosition);
                indice++;
            }
        }
    }
    return states;
}
```

STATE

La clase State corresponde a los diferentes estados de la tabla q, como hemos explicado anteriormente se guardaran los casillas walkable, lo cual se hará con un array de booleanos, y por otra parte la posición relativa del enemigo, que será un array de ints ya que hay un total de 3 posibilidades por cada posición del array.

```
public class State : MonoBehaviour
{
    public int id;
    public bool[] walkableNeighbours; // [north, east, south, west] walkables

    // [north/aligned/south , west/aligned/east] = [1/0/-1 , 1/0/-1]
    public int[] enemyRelativePosition;

    2 referencias
    public State(int id, bool[] walkableArray, int[] enemyPosArray)
    {
        this.id = id;
        walkableNeighbours = walkableArray;
        enemyRelativePosition = enemyPosArray;
    }
}
```

MYTRAINER

El objetivo de esta clase es explorar el tablero y calcular todos los valores Q de lasceldas por las cuales nuestro agente vaya pasando. Para ello se hace uso de distintas funciones explicadas a continuación.

Initialize():

En este método inicializaremos las distintas variables y la tabla q e iniciaremos un primer episodio.

DoStep():

Este método se ejecutará a cada paso que dé el agente, es decir es el método más importante de nuestro código. En primer lugar, se eligirá una acción aleatoria y se calculará esa celda, a continuación, se comprobará si el agente va a ir a una celda aleatoria, o si va a elegir la celda que tenga mayor valor Q, para esto se buscará el valor más alto que corresponda al estado y devolverá la acción. Una vez se ha hecho esto, se calculará el valor Q de la celda en la que se encuentra el agente, para ello se ha implementado el método getQ().

getQ():

Cada vez que se quiera obtener un estado se va a implementar un código parecido, ya que se calculará un array auxiliar de booleanos donde se guarda si las casillas vecinas dela agente son walkable, y por otra parte se calcula otro array auxiliar de ints para guardar la posición relativa del enemigo respecto del agente. Este es el código que lo implementa:

```
//Casillas vecinas
CellInfo north = QMind.Utils.MoveAgent(0, currentCell, worldInfo);
CellInfo east = QMind.Utils.MoveAgent(1, currentCell, worldInfo);
CellInfo south = QMind.Utils.MoveAgent(2, currentCell, worldInfo);
CellInfo west = QMind.Utils.MoveAgent(3, currentCell, worldInfo);

bool[] tempWalkableArray = new bool[] {
    north.Walkable, east.Walkable, south.Walkable, west.Walkable
};

//Posicion relativa del enemigo
int[] tempEnemyRelativePosition = new int[2];
if (OtherPosition.y > currentCell.y)
{
    tempEnemyRelativePosition[0] = 1;
}
if (OtherPosition.y == currentCell.y)
{
    tempEnemyRelativePosition[0] = 0;
}
if (OtherPosition.y < currentCell.y)
{
    tempEnemyRelativePosition[0] = -1;
}

if (OtherPosition.x > currentCell.x)
{
    tempEnemyRelativePosition[1] = 1;
}
if (OtherPosition.x == currentCell.x)
{
    tempEnemyRelativePosition[1] = 0;
}
if (OtherPosition.x < currentCell.x)
{
    tempEnemyRelativePosition[1] = -1;
}
```

Una vez hecho esto recorreremos todos los estados de la tabla Q hasta que corresponda al de la acción que hemos calculado previamente, y se devuelve el valor de este estado según la acción.

```
//Buscamos el estado que corresponda a la casilla vecina y posicion del enemigo relativa calculada
for (int i = 0; i < tableQ.statesArray.Length; i++)
{
    if (compararArraysBooleanos(tempWalkableArray, tableQ.statesArray[i].walkableNeighbours)
                                &&
        compararArraysInt(tempEnemyRelativePosition, tableQ.statesArray[i].enemyRelativePosition))
    {
        return tableQ.value[action, i];
    }
}
```

Además, se han creado dos métodos auxiliares para comparar los arrays:

```
#region COMPARE ARRAYS METHODS
//Métodos para comparar array de booleanos y de ints
4 referencias
public static bool compararArraysBooleanos(bool[] array1, bool[] array2)
{
    if (array1.Length != array2.Length)
        return false;

    for (int i = 0; i < array1.Length; i++)
    {
        if (array1[i] != array2[i])
            return false;
    }

    return true;
}

4 referencias
public static bool compararArraysInt(int[] array1, int[] array2)
{
    if (array1.Length != array2.Length)
        return false;

    for (int i = 0; i < array1.Length; i++)
    {
        if (array1[i] != array2[i])
            return false;
    }

    return true;
}
#endregion
```

Una vez hecho esto, se calcula el valor de Q máximo de la siguiente celda, con el método GetMaxQ().

GetMaxQ():

Este busca el estado de la misma forma que getQ(), pero este método se diferencia en que guarda el índice cuando se encuentra el estado, y a partir de ahí calcula el mayor valor de Q de las distintas acciones:

```
int indice = 0;
//Buscamos el estado que corresponda a la casilla vecina y posicion del enemigo relativa calculada
for (int i = 0; i < tableQ.statesArray.Length; i++)
{
    if (compararArraysBooleanos(tempWalkableArray, tableQ.statesArray[i].walkableNeighbours)
                                    &&
        compararArraysInt(tempEnemyRelativePosition, tableQ.statesArray[i].enemyRelativePosition))
    {
        indice = i;
    }
}

float best_q = -1000f;

for (int actualAction = 0; actualAction < tableQ.nRows; actualAction++)
{
    if (tableQ.value[actualAction, indice] > best_q)
    {
        best_q = tableQ.value[actualAction, indice];
    }
}

return best_q;
```

Posteriormente calcularemos la recompensa con el método GetReward().

GetReward():

La recompensa seguirá el siguiente patrón: si se aleja del enemigo (se calcula con la distancia manhattan), se recompensará con 100, si es una casilla non-walkable, se penalizará con -1000, y si no es ninguna de estas dos opciones se dará 0.

Volviendo al DoStep(), se calculará el nuevo valor de Q.

Update_rule():

Aquí se aplica la regla del aprendizaje, regida por la siguiente fórmula:

```
public float Update_rule(float currentQ, float reward, float maxQ)
{
    float aux = 0.0f;
    aux = (1 - parameters.alpha) * currentQ + parameters.alpha * (reward + parameters.gamma *
   maxQ);
    return aux;
}
```

Por último se actualizará la tablaQ con el método UpdateTableQ().

UpdateTableQ():

En este método se buscará el estado como hemos estado haciendo y cuando se encuentre este se guardará el índice y se modificará en la matriz de valores según la acción realizada el índice del estado.

En este punto, se actualizará la posición del agente a la próxima celda y la del enemigo a la siguiente celda que permita acercarse al agente, si la posición coincide, se iniciará un nuevo episodio, donde se situará tanto al agente como al enemigo en posiciones aleatorias y se reseateará el numero de pasos, además, se guardará la tabla Q en un .csv si han pasado un número determinado de episodios que el jugador podrá controlar en el inspector.

```
private void NuevoEpisodio(WorldInfo worldInfo)
{
    //reiniciamos posiciones
    AgentPosition = worldInfo.RandomCell();
    OtherPosition = worldInfo.RandomCell();
    //reinciamos numero de pasos
    counter = 0;
    CurrentStep = counter;
    numEpisode++;
    CurrentEpisode = numEpisode;
    //se guarda la tabla Q cada "episodesBetweenSaves"
    if (numEpisode % parameters.episodesBetweenSaves == 0)
    {
        GuardarTablaQ();
    }
    OnEpisodeStarted?.Invoke(this, EventArgs.Empty);
}
#endregion
```

MYTESTER

Esta clase será la encargada de ejecutar los movimientos necesarios según el aprendizaje que se ha llevado a cabo anteriormente. En primer lugar, como atributos esta clase tendrá: una matriz de floats que corresponderá a los valores de la tabla Q, una lista de estados, que se calculan de la misma forma que en el myTrainer y n filas y n columnas.

```
private WorldInfo worldInfo;

private float[,] _tablaQ; //Valores de la tabla Q
Grupo9.State[] states = new Grupo9.State[16 * 9]; //array de estados

//tamaño de la tabla Q
private int nRows = 4;
private int nCols = (4*4)* (3*3);

2 referencias
public void Initialize(WorldInfo worldInfo)
{
    this.worldInfo = worldInfo;
    InitializeStates(); //inicializamos array de estados
    LoadQTable();


}
```

Se han inicializado los estados tal y como se ha hecho en el myTrainer y el método donde se carga la tabla Q como .csv en la matriz de floats es el siguiente:

```
private void LoadQTable()
{
    string filePath = @"Assets/Scripts/Grupo9/tableQ.csv"; //Se busca la tablaQ
    StreamReader reader;
    if (File.Exists(filePath))
    {
        reader = new StreamReader(File.OpenRead(filePath));
        _tablaQ = new float[nRows, nCols];
        int contador = 0;
        while (!reader.EndOfStream && contador < nRows)
        {
            var line = reader.ReadLine();
            var values = line.Split(';');
            for (int i = 0; i < values.Length; i++)
            {
                _tablaQ[contador, i] = (float)Convert.ToDouble(values[i]);
            }
            contador++;
        }
    }
}
```

GetNextStep():

Este método calculará cual es el mejor paso que tiene que dar el agente según la tabla q. En primer lugar calculará el estado con el método CalculateState().

CalculateState():

Este seguirá la misma lógica de encontrar estados que se utilizó en el myTrainer, y se devolverá.

```
//Buscamos el estado que corresponda a la casilla vecina y posicion del enemigo relativa calculada
for (int i = 0; i < states.Length; i++)
{
    if (compararArraysBooleanos(tempWalkableArray, states[i].walkableNeighbours)
                                    &&
        compararArraysInt(tempEnemyRelativePosition, states[i].enemyRelativePosition))
    {
        return states[i]; //Se devuelve el estado
    }
}
return null;
```

Luego se calculará la mejor acción según los valores de Q, con el método GetAction().

GetAction():

Devuelve la mejor acción que puede realizar según el aprendizaje de la tabla Q y de sus valores:

```
private int GetAction(Grupo9.State state)
{
    //Se deuvel la mejor acción según el valor de Q
    int bestQaction = 0;
    float bestQ = -1000.0f;
    for (int i = 0; i < nRows; i++)
    {
        if (_tablaQ[i, state.id] >= bestQ)
        {
            bestQ = _tablaQ[i, state.id];
            bestQaction = i;
        }
    }
    return bestQaction;
}
```

Por último, se actualizará la posición del agente y se devolverá

CONCLUSIONES

Respecto a la anterior entrega, se ha pensado como calcular los estados de forma más optima y esto ha hecho que tengamos un número de estados bastante pequeño (144). Esto ha hecho que su implementación sea mas sencilla ya que se han creado clases para que tengan cada una su propia funcionalidad y que sea mucho mas simple y lógica todo el código. Estos scripts se han probado en diferentes escenarios y no se han encontrado fallos, y siempre suele llegar a un número alto de pasos en el MyTester. Cabe resaltar de que cuando se ha entrenado alrededor de mil episodios, a la hora de testear el agente tendía a moverse al borde derecho del mapa y se quedaba en bucle ahí huyendo del enemigo, en la tabla proporcionada se ha realizado un entrenamiento de unos 8000 pasos y esto hace que el movimiento sea mucho más diverso. Respecto a la entrega anterior, se ha modifcado como se otorga la recompensa para que penalice si el agente busca una casilla no transitable, con el objetivo de que nunca vaya a por esa opción y no tengamos que estar controlando mediante el script que busque otras opciones si se va a una casilla no transitable.