Universidad
Rey Juan Carlos

Práctica 2

Desarrollo de juegos con Inteligencia Artificial - Grado en Diseño y Desarrollo de Videojuegos

Grupo 8

Jose Mª Soriano Villalba
Juan Coronado Gómez

ÍNDICE

Introducción

El objetivo de esta práctica es implementar un algoritmo de aprendizaje por refuerzo basado en Q-Learning que enseñe al agente inteligente Agent a escapar el mayor tiempo posible del Player. Con este documento, se procede a explicar el desarrollo de dicho algoritmo de aprendizaje implementado junto con el código de sus respectivas clases e interfaces.

Clase Q-State

La primera clase que se implementa para solucionar el problema planteado es QState. Dentro de ella, se encuentran los aspectos del entorno que se encargan de caracterizarlo y de diferenciarlo entre una situación u otra.
Tras múltiples pruebas, observando el rendimiento del aprendizaje en cada una de ellas (guardar las posiciones del agente y del enemigo, agrupar celdas en regiones…), se llega a una abstracción eficiente:

- Primero se almacenan los vecinos accesibles desde una casilla (nWalkable, sWalkable, eWalkable, wWalkable).
- Además, se tienen otras dos variables que sirven para caracterizar la posición del enemigo (player) respecto al agente. Con ellas se almacena si el player está a la derecha, izquierda o misma línea que el agente, y si está arriba, abajo o en la misma línea también.
- Por último, se brinda un índice que caracteriza a cada estado.

Clase Q-Table

La segunda clase que se utiliza para la resolución del problema que se plantea es la clase Q-Table, en ella, se crea la tabla Q con sus respectivas filas y columnas, además, también se inicializa mediante una función una lista de estados creando así todos las posibles combinaciones que puede tomar el entorno. Esto se consigue mediante la creación de una tabla de verdad de atributos booleanos sumando todas las combinaciones de atributos numéricos. Se cuenta por lo tanto con 2^4*3*3 (un total de 144) estados distintos. Entonces, la tabla Q, se compone de 144 columnas y 4 filas.

Además de para almacenar la tabla y los estados, la clase cuenta con los métodos necesarios para ejecutar el algoritmo de Q Learning. Se cuenta con una función que devuelve la mejor acción para cada estado, otra para devolver la Q dados un estado y una acción, para actualizar la Q después de aplicar la regla de aprendizaje…

La lógica dentro de estas funciones es muy similar. Se les pasa como parámetros los atributos del entorno en ese momento, para averiguar en qué estado se encuentra, recorriendo la lista y comparando mediante un bucle. Una vez se averigua el índice del estado en el que está el entorno, se devuelve su Q, su mejor acción, o lo que proceda según la función.

```
1 referencia
public int DevolverMejorAccion(bool n, bool s, bool e, bool w, int up, int right)
{
    int indice = 0;
    // Primero se recorre la lista de estados, para identificar de cuál se trata, en base a las posiciones pasadas como parámetro
    for (int i = 0; i < _listaEstados.Length; i++)
    {
        if (n == _listaEstados[i]._nWalkable &&
            s == _listaEstados[i]._sWalkable &&
            w == _listaEstados[i]._wWalkable &&
            e == _listaEstados[i]._eWalkable &&
            up == _listaEstados[i]._playerUp &&
            right == _listaEstados[i]._playerRight)
        {
            // Se guarda el índice del estado
            indice = _listaEstados[i]._idState;
        }
    }

    int mejorAccion = 0;
    float mejorQ = -1000f;

    for(int i=0; i< _numRows; i++)
    {
        if (_tablaQ[i,indice]>mejorQ)
        {
            mejorAccion = i;
            mejorQ = _tablaQ[i, indice];
        }
    }

    return mejorAccion;
}
```

Clase MyTrainer

Esta clase implementa la interfaz IQMindTrainer, por lo que cuenta con las funciones Initialize() y DoStep().
La primera se encarga de crear la tabla Q, y albergarla como parámetro, además de inicializar la lista de estados.
Después, la función DoStep() se encarga de controlar todo el algoritmo de aprendizaje. Para ello, se emplean los parámetros dados en la clase QMindTrainerParams.

1. Se genera un número aleatorio. Dependiendo de si es mayor o menor que el valor de épsilon declarado en los parámetros dados, la acción que realiza en el aprendizaje es al azar o no.
2. Si no es una acción al azar, se busca aquella que ofrezca mayor Q, gracias a la función MejorAccion(). En ella, se calculan los parámetros necesarios para caracterizar los estados del entorno (si los vecinos son caminables, la posición relativa del enemigo respecto del agente).

Una vez hecho esto, se pasan todos esos atributos a la función que contiene la tabla Q anteriormente mencionada, que devuelve la mejor acción para realizar.

```
// Para escoger la mejor acción, primero debemos averiguar el estado en el que nos encontramos actualmente
// De esta forma, se escoge la acción que aporte más Q

int posX = AgentPosition.x;
int posY = AgentPosition.y;

CellInfo north = QMind.Utils.MoveAgent(0, AgentPosition, _worldInfo);
bool n = north.Walkable;

CellInfo south = QMind.Utils.MoveAgent(2, AgentPosition, _worldInfo);
bool s = south.Walkable;

CellInfo east = QMind.Utils.MoveAgent(1, AgentPosition, _worldInfo);
bool e = east.Walkable;

CellInfo west = QMind.Utils.MoveAgent(3, AgentPosition, _worldInfo);
bool w = west.Walkable;

int up = 0, right = 0;
if(OtherPosition.x > AgentPosition.x)
{
    right = 0;
}
if(OtherPosition.x < AgentPosition.x)
{
    right = 1;
}
if(OtherPosition.x == AgentPosition.x)
{
    right = 2;
}

if (OtherPosition.y > AgentPosition.y)
{
    up = 0;
}
if (OtherPosition.y < AgentPosition.y)
{
    up = 1;
}
if (OtherPosition.y == AgentPosition.y)
{
    up = 2;
}

return _QTable.DevolverMejorAccion(n,s,e,w,up,right);
```

3. Una vez escogida la acción, se comprueba que no lleve a una casilla inalcanzable, para evitar que salte un error y se trunque el aprendizaje. En caso de que esto suceda, se penaliza fuertemente la acción, para intentar que no vuelva a suceder en el futuro. Para ello, se le brinda una recompensa negativa muy alta.
4. Después de comprobar que la acción se puede realizar, se comienza a aplicar la regla de aprendizaje:

$$Q'^{(s,a)} = (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{\forall a'} Q(s',a'))$$

Por lo tanto, se necesita calcular la Q del estado actual con la acción escogida, la recompensa asociada y la mejor Q de todas las posibles en el estado siguiente. Se necesita identificar en qué estado se encuentra el entorno, por lo que se aplica la misma lógica que para el caso de escoger la mejor acción, plasmado en la parte superior.

5. Para determinar la recompensa asociada, se tienen en cuenta varios factores:
    - Distancia Manhattan con el enemigo. Si se aumenta la distancia respecto al enemigo con la acción, se brinda una recompensa de 100. En cambio, si se acerca, se penaliza con -10. Si además, nos encontramos al lado del enemigo y realizamos una mala acción, se penaliza con -100 más.
    - Agente capturado. Si con la acción el enemigo nos captura, se penaliza con -100 directamente.
    - Agente en las esquinas. Se ha tratado de evitar en la medida de lo posible, que el agente vaya hacia las esquinas, ya que ahí el enemigo lo captura con facilidad. Por lo tanto, si se encuentra en una, se penaliza con -1000.
6. Por último, se calcula el nuevo valor de Q aplicando todos los factores, y se coloca en la tabla.

Esto se repite para cada “step”, hasta que se alcanza el número máximo de “steps” por episodio. Una vez se finaliza el episodio, comienza uno nuevo, con posiciones aleatorias, explorando de esta forma nuevas casillas. Alcanzado el máximo de episodios, se finaliza el entrenamiento.

Además, dentro de esta clase, se encuentra implementada la función GuardarTablaQ(), que se encarga de escribir sus datos en un fichero .csv, para poder cargarlos después en la escena de testing.

```
1 referencia
private void GuardarTablaQ()
{
    File.WriteAllLines(@"Assets/Scripts/Grupo8/TablaQ.csv",
        ToCsv(_QTable._tablaQ));
}

1 referencia
private static IEnumerable<String> ToCsv<T>(T[,] data, string separator = ";")
{
    for (int i = 0; i < data.GetLength(0); ++i)
        yield return string.Join(separator, Enumerable
          .Range(0, data.GetLength(1))
          .Select(j => data[i, j]));
}
```

Clase MyTester

Por último, se implementa la clase MyTester, donde se carga la Q-Table y se inicializan sus estados de la misma manera que al inicio en la clase Q-Table. Mediante el fichero .csv, se leen los valores de la tabla Q y se pasa cada uno a una casilla distinta de dicha tabla. Esto se consigue con una función llamada Convert.ToDouble que pasa el valor String de la tabla a Double y de Double a Float, mediante un casting.
Una vez cargada la tabla, la función principal de esta clase es brindar la mejor acción al agente, para conseguir escapar del enemigo. Para ello, se cuenta con la función GetNextStep():

1. Se percibe el entorno y se clasifica el estado en el que se encuentra, de la misma forma que durante el aprendizaje.
2. Se obtiene la acción con mayor Q para el estado percibido y se devuelve.

Resultado final

Después de aplicar el algoritmo de aprendizaje, se ha logrado que el agente aguante más de 5000 pasos sin que el enemigo lo alcance.