PRÁCTICA 2

DESARROLLO DE JUEGOS CON INTELIGENCIA ARTIFICIAL

GRUPO H

Alba Haro Ballesteros

Jacobo Sánchez García

Luis Fernando Rodriguez Rivera

1. Introducción

En esta segunda práctica de Desarrollo de Juegos con Inteligencia Artificial, se ha desarrollado un agente inteligente que utiliza un algoritmo de aprendizaje por refuerzo basado en Q-Learning para evitar ser atrapado por un enemigo en un entorno controlado.

El objetivo principal de la práctica es dotar al agente de un comportamiento aprendido mediante técnicas de Machine Learning vistas en clase, permitiendo que este tome decisiones óptimas basadas en una tabla Q entrenada previamente.

Contexto

El entorno del juego consiste en un mapa cuadrícula, donde un agente inteligente debe desplazarse evitando a un enemigo que lo persigue. El mapa incluye:

- **Celdas transitables:** Donde el agente puede moverse.
- **Celdas no transitables:** Obstáculos que el agente debe evitar.
- **Un enemigo:** Persigue al agente utilizando el algoritmo A*.

El objetivo del agente es maximizar su supervivencia evitando al enemigo y realizando movimientos válidos en el mapa. El agente se evaluará en 10 escenarios diferentes de los proporcionados. En cada escenario se contarán los pasos que da el agente antes de ser atrapado o realizar una acción imposible.

2. Diseño de la tabla Q

Estado

La clase Estado se diseñó para representar un punto específico del entorno del agente. Este script encapsula la información clave que el agente necesita para evaluar su situación actual y decidir su próxima acción.

Cada estado combina:

- Un identificador único que permite indexarlo en la tabla Q.
- La posición relativa del enemigo con respecto al agente.
- Un array de booleanos que representa la caminabilidad en las direcciones cardinales (norte, sur, este, oeste).

Esta representación permite al agente interpretar rápidamente su contexto y tomar decisiones fundamentadas.

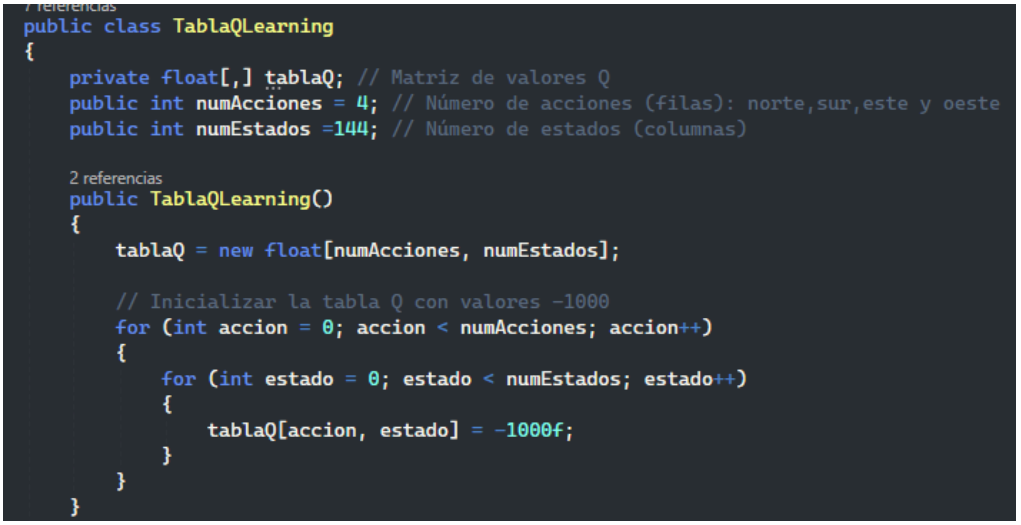
TablaQLearning

El script TablaQLearning define la clase que implementa la estructura de datos principal para almacenar y gestionar los valores Q del algoritmo de Q-Learning. Esta clase se encarga de representar la tabla Q como una matriz bidimensional y de ofrecer operaciones para inicializar, actualizar, consultar y cargar datos desde archivo.

Funcionalidades principales:

- ### Inicialización

  Crea una tabla Q con dimensiones fijas: 4 acciones (norte, sur, este y oeste) y 144 estados. Todos los valores se inicializan a **-1000**, lo cual representa un desconocimiento total y fuerza al agente a explorar. Esta inicialización evita que se asuman prematuramente caminos como válidos cuando no se han evaluado.



```
7 referencias
public class TablaQLearning
{
    private float[,] tablaQ; // Matriz de valores Q
    public int numAcciones = 4; // Número de acciones (filas): norte,sur,este y oeste
    public int numEstados =144; // Número de estados (columnas)

    2 referencias
    public TablaQLearning()
    {
        tablaQ = new float[numAcciones, numEstados];

        // Inicializar la tabla Q con valores -1000
        for (int accion = 0; accion < numAcciones; accion++)
        {
            for (int estado = 0; estado < numEstados; estado++)
            {
                tablaQ[accion, estado] = -1000f;
            }
        }
    }
```

- ### Inicialización desde archivo

  Alternativamente, permite crear la tabla cargando valores desde un archivo CSV. Cada línea del archivo contiene un trío de valores: estado, acción y valor Q. Se validan los datos antes de insertarlos y se omiten líneas mal formateadas. Esto permite que el aprendizaje sea persistente entre sesiones.

- ### Actualización de valores

  Dado un estado actual, una acción, una recompensa recibida y el nuevo estado alcanzado, se actualiza el valor Q usando la fórmula estándar del algoritmo Q-Learning: $Q(s, a) \leftarrow Q(s, a) + \alpha * [r + \gamma * \max(Q(s', a')) - Q(s, a)]$.

- ### Consulta

  Devuelve el valor Q correspondiente a una acción y un estado dados. Incluye validación de índices para evitar errores en tiempo de ejecución.

- ### Actualización directa

  Establece directamente un valor Q para una acción y un estado concretos, utilizado

tanto durante el entrenamiento como en la carga desde archivo.

- Selección de mejor acción
  Devuelve la acción con el valor Q más alto para un estado dado. Si hay empate, selecciona aleatoriamente entre las mejores acciones. Esto permite que el agente se comporte de forma óptima al tomar decisiones.

```
3 referencias
public int ObtenerMejorAccion(int estado)
{
    float mejorQ = float.MinValue;
    List<int> mejoresAcciones = new List<int>();

    // Encuentra las acciones con el mejor valor Q
    for (int accion = 0; accion < numAcciones; accion++)
    {
        float qValor = ObtenerQ(accion, estado);
        if (qValor > mejorQ)
        {
            mejorQ = qValor;
            mejoresAcciones.Clear();
            mejoresAcciones.Add(accion);
        }
        else if (qValor == mejorQ)
        {
            mejoresAcciones.Add(accion);
        }
    }

    // Selecciona una acción aleatoria entre las mejores
    return mejoresAcciones[UnityEngine.Random.Range(0, mejoresAcciones.Count)];
}
```

Esta clase encapsula por completo la gestión de la tabla Q, centralizando la lógica relacionada con el aprendizaje, la inferencia y la persistencia. Es una pieza fundamental tanto para el entrenamiento (QMindTrainer.cs) como para la inferencia (QMindTester.cs), y permite guardar y recuperar el conocimiento del agente entre ejecuciones del program

3. Algoritmos de entrenamiento

Movimiento

El script Movimiento gestiona la dinámica del entorno, permitiendo que el agente y el enemigo se desplacen dentro del mapa. Este script valida y ejecuta movimientos respetando las restricciones del entorno.

- El enemigo se mueve utilizando un algoritmo de navegación (como A*).
- El agente se desplaza según la acción elegida (norte, sur, este, oeste) y si la celda destino es transitable.

Funciones Principales:

- Validar si un movimiento es posible.
- Calcular la nueva posición del agente o del enemigo en función de sus objetivos.

QMindTrainer

QMindTrainer es el núcleo del entrenamiento del agente e implementa la interfaz IQMindTrainer, con las funciones Initialize() y DoStep().

```
3 referencias
public void DoStep(bool train)
{
    // Obtener el estado actual basado en la posición del agente y el enemigo
    int estadoActual = ObtenerEstado(AgentPosition, OtherPosition, mundo);

    // Elegir la acción (considerando si está en modo de entrenamiento o no)
    int accion = SeleccionarAccion(estadoActual, train);

    // Realizar la acción y obtener la nueva posición
    CellInfo nuevaPosicion = EjecutarAccion(accion);

    // Calcular el nuevo estado tras el movimiento
    int nuevoEstado = ObtenerEstado(nuevaPosicion, OtherPosition, mundo);


    Debug.Log($"DoStep - Estado actual: {estadoActual}, Acción elegida: {accion}," +
        $" Nueva posición: ({nuevaPosicion.x}, {nuevaPosicion.y}), Nuevo estado: {nuevoEstado}");

    // Calcular recompensa y actualizar tabla Q si está en modo entrenamiento
    if (train)
    {
        float recompensa = CalcularRecompensa(nuevaPosicion);
        recompensaTotal += recompensa;
        tablaQ.CalcularQ(estadoActual, accion, recompensa, nuevoEstado,parametros.alpha,parametros.gamma);
    }


    // Actualizar la posición del agente y del enemigo
    posicionAnteriorAgente = AgentPosition;
    accionAnterior = accion;
    AgentPosition = nuevaPosicion;

    // Mover al enemigo
    OtherPosition = GrupoH.Movimiento.MovimientoEnemigo(algoritmoNavegacion, OtherPosition, AgentPosition);

    // Verificar si el episodio debe terminar
    if (AgentPosition.Equals(OtherPosition) || CurrentStep >= parametros.maxSteps)
    {
        Debug.Log("DoStep - Episodio terminado");
        OnEpisodeFinished?.Invoke(this, EventArgs.Empty);
        ReiniciarEpisodio();
    }
    float nuevaDistancia = AgentPosition.Distance(OtherPosition, CellInfo.DistanceType.Manhattan);
    if (nuevaDistancia >= 10)
    {
        Debug.Log("Agente se ha alejado lo suficiente. Fin del episodio.");
        OnEpisodeFinished?.Invoke(this, EventArgs.Empty);
        ReiniciarEpisodio();
        return;
    }
    else
    {
        CurrentStep++;
    }
    Debug.Log($" De {AgentPosition} -> {nuevaPosicion} | Enemigo en " +
        $"{OtherPosition} | Distancia: {nuevaPosicion.Distance(OtherPosition, CellInfo.DistanceType.Manhattan)}");

}
```

Funcionamiento

El proceso de entrenamiento se organiza en episodios y pasos, siguiendo esta estructura:

1. **Inicialización:**
    - Configuración de parámetros (epsilon, alpha, gamma).
    - Creación o carga de la tabla Q.
    - Posicionamiento aleatorio del agente y el enemigo.

```
2 referencias
public void Initialize(QMindTrainerParams qMindTrainerParams, WorldInfo worldInfo, INavigationAlgorithm navigationAlgorithm)
{
    Debug.Log("QMindTrainer: initialized");
    // Inicialización
    parametros = qMindTrainerParams;
    mundo = worldInfo;
    algoritmoNavegacion = navigationAlgorithm;

    //carga la tabla si existe
    if (File.Exists(RUTA_TABLA))
    {
        tablaQ = new TablaQLearning(RUTA_TABLA);
        //CargarTablaQ();
        Debug.Log("Tabla Q cargada correctamente.");
    }
    //si no, crea una nueva
    else
    {
        tablaQ = new TablaQLearning();
        Debug.Log("Se ha creado una nueva tabla Q.");
    }

    AgentPosition = mundo.RandomCell();
    OtherPosition = mundo.RandomCell();
    CurrentEpisode = 0;
    CurrentStep = 0;
    ReturnAveraged = 0;

    OnEpisodeStarted?.Invoke(this, EventArgs.Empty);
}
```

2. **Selección de Acciones:**
    - Se aplica una estrategia epsilon-greedy.
    - Se elige una acción aleatoria o la mejor acción según los valores Q del estado actual.
3. **Ejecución de Acciones:**
    - El agente intenta moverse en la dirección elegida.
    - Si la celda no es válida, permanece en su posición actual.
4. **Cálculo de Recompensas:**
    - Recompensa +50 si se aleja significativamente del enemigo (>5 celdas).
    - Recompensa +20 si se aleja.
    - Penalización -5 si se acerca al enemigo.
    - Penalización -10 por intentar moverse a una celda no transitable.
    - Penalización -100 si es alcanzado por el enemigo.

```
private float CalcularRecompensa(CellInfo nuevaCelda)
{
    float distanciaActual = AgentPosition.Distance(OtherPosition, CellInfo.DistanceType.Manhattan);
    float nuevaDistancia = nuevaCelda.Distance(OtherPosition, CellInfo.DistanceType.Manhattan);

    if (!nuevaCelda.Walkable)
        return -10f;

    if (nuevaCelda.Equals(OtherPosition))
        return -100f;

    if (nuevaDistancia > distanciaActual)
    {
        if(nuevaDistancia> 5f)
        {
            return 50f;
        }
        return 20f;
    }



    if (nuevaDistancia < distanciaActual)
        return -5f;

    return 0;

}
```

5. **Actualización de la Tabla Q:**
    - Se aplica la fórmula de Q-Learning para actualizar el valor Q correspondiente.
6. **Reinicio de Episodios:**
    - Se reinicia si el agente es atrapado, alcanza la distancia máxima o el número máximo de pasos.
    - Se actualiza epsilon progresivamente para reducir la exploración.
    - Se guarda periódicamente la tabla Q en CSV.

Persistencia de Datos

- GuardarTablaQ() escribe los valores en un CSV.
- La tabla Q se reutiliza si existe en disco.

Visualización

Se usa OnGUI() para mostrar por pantalla información como episodio actual, paso actual y recompensas.

QMindTester

El archivo QMindTester implementa la lógica de inferencia del agente inteligente tras su fase de entrenamiento. Este script cumple con la interfaz IQMind y es el encargado de utilizar la tabla Q generada previamente para guiar al agente en tiempo real, sin seguir explorando ni modificando los valores aprendidos.

```
public void Initialize(WorldInfo worldInfo)
{
    Debug.Log("QMindTester: inicializando...");

    // Inicialización del mundo y tabla Q
    mundo = worldInfo;
    tablaQ = new TablaQLearning();

    // Cargar la tabla Q previamente entrenada
    CargarTablaQ();

    Debug.Log("QMindTester: inicialización completa.");
}
```

La función GetNextStep() es el método clave del script y se ejecuta en cada frame durante la ejecución en modo inferencia. Realiza las siguientes acciones:

- Calcula el estado actual del juego en función de la posición del agente, la del enemigo y las direcciones caminables desde la celda actual.
- Selecciona la **mejor acción posible** para ese estado, consultando directamente la tabla Q sin aplicar exploración aleatoria.
- Calcula la nueva posición del agente según la acción seleccionada y la devuelve.

```
2 referencias
public CellInfo GetNextStep(CellInfo currentPosition, CellInfo otherPosition)
{
    if (currentPosition == null || otherPosition == null)
    {
        Debug.LogError("Las posiciones proporcionadas son nulas.");
        return currentPosition; // Mantener la posición actual
    }

    // Calcular el estado actual
    int estadoActual = ObtenerEstado(currentPosition, otherPosition, mundo);
    Debug.Log($"Estado actual: {estadoActual}, Posición agente: ({currentPosition.x}, {currentPosition.y}), " +
              $"Posición enemigo: ({otherPosition.x}, {otherPosition.y})");

    // Calcular la distancia Manhattan al enemigo
    float distanciaEnemigo = currentPosition.Distance(otherPosition, CellInfo.DistanceType.Manhattan);
    Debug.Log($"Distancia al enemigo: {distanciaEnemigo}");

    // Seleccionar la mejor acción basada en la tabla Q
    int mejorAccion = tablaQ.ObtenerMejorAccion(estadoActual);
    Debug.Log($"Acción elegida: {mejorAccion}");

    // Mover el agente basado en la acción elegida
    CellInfo nuevaPosicionFinal = Movimiento.MovimientoAgente(mejorAccion, currentPosition, mundo);

    Debug.Log($"Nueva posición del agente: ({nuevaPosicionFinal.x}, {nuevaPosicionFinal.y})");
    return nuevaPosicionFinal;
}
```

Cálculo del estado

El estado se calcula combinando:

- Posición relativa del enemigo respecto al agente.
- Caminabilidad del entorno (norte, este, sur, oeste).

Esto se traduce en un identificador numérico único que se usa como índice en la tabla Q.

```
private int ObtenerEstado(CellInfo agente, CellInfo enemigo, WorldInfo mundo)
{
    int deltaX = Mathf.Clamp(enemigo.x - agente.x, -1, 1) + 1;
    int deltaY = Mathf.Clamp(enemigo.y - agente.y, -1, 1) + 1;
    int posicionRelativa = deltaX * 3 + deltaY;

    bool[] direcciones = new bool[4];
    direcciones[0] = agente.y + 1 < mundo.WorldSize.y && mundo[agente.x, agente.y + 1].Walkable;
    direcciones[1] = agente.x + 1 < mundo.WorldSize.x && mundo[agente.x + 1, agente.y].Walkable;
    direcciones[2] = agente.y - 1 >= 0 && mundo[agente.x, agente.y - 1].Walkable;
    direcciones[3] = agente.x - 1 >= 0 && mundo[agente.x - 1, agente.y].Walkable;

    int codAccesibilidad = 0;
    for (int i = 0; i < 4; i++)
        if (direcciones[i]) codAccesibilidad |= (1 << i);

    int estado = posicionRelativa * 16 + codAccesibilidad;
    Debug.Log($"ObtenerEstado - Relación enemigo: {posicionRelativa}, Caminabilidad: {codAccesibilidad}, Estado: {estado}");
    return estado;
}
```

El número total de estados se calcula a partir de las combinaciones de:

- Caminabilidad: 2^4 = 16 combinaciones.
- Posición relativa del enemigo: 3x3 = 9 combinaciones.

Total: 16 × 9 = 144 estados distintos.

4. Resultado final

Como resultado final se ha conseguido una implementación fiel al algoritmo de Q-Learning. El agente aprende a mantenerse alejado del enemigo a partir de recompensas y penalizaciones definidas.

La tabla Q generada queda almacenada en CSV para ser reutilizada por el sistema de inferencia (QMindTester) sin necesidad de reentrenar