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.

Esta clase, combina la posición relativa del enemigo respecto al agente y las condiciones de accesibilidad en las direcciones cardinales (norte, sur, este y oeste). Esto permite que el agente interprete rápidamente su contexto y tome decisiones fundamentadas.

Contenido:

- **Id:** Un identificador único que permite indexar este estado en la tabla Q.
- **PosicionEnemigo:** Un vector que indica la posición relativa del enemigo.
- **Direcciones:** Un array booleano que señala si cada dirección cardinal es transitable.

TablaQLearning

El script `TablaQLearning` centraliza el almacenamiento y gestión de los valores Q. Este componente es esencial para la implementación del aprendizaje por refuerzo.

Este script fue diseñado para manejar la matriz que contiene los valores Q, permitiendo operaciones como actualizar valores, consultar las mejores acciones y guardar los datos en el archivo CSV.

Funciones Principales:

- Obtener y actualizar valores Q asociados a estados y acciones.
- Seleccionar la mejor acción para un estado dado.
- Guardar y cargar la tabla en formato CSV para persistir los resultados del entrenamiento.

GeneradorEstados

En esta clase, lo que se busca es considerar todas las combinaciones posibles de estados y preparar la tabla Q para su inicialización.

Se garantiza una cobertura total de todos los estados posibles combinando posiciones relativas del enemigo (x, y) con configuraciones de accesibilidad (Norte, Sur, Este, Oeste).

Las acciones posibles, van a venir definidas por el movimiento del agente en los ejes X e Y, siguiendo este orden: arriba (Norte), derecha(Este), abajo(Sur) e izquierda(Oeste), lo que implica un total de **4 acciones distintas**.

Por otro lado, el estado del agente se basa en dos factores: las **acciones posibles** para moverse a una celda vecina accesible y la **posición relativa** del oponente en los ejes X e Y.

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

- Cada una de las 4 acciones posibles puede estar habilitada o no, lo que genera 2^4=**16 combinaciones.**
- La posición relativa del oponente en cada uno de los dos ejes puede ser menor, igual o mayor que la del agente, lo que da lugar a 3^2 = **9 combinaciones.**

Combinando ambos factores, se obtiene un total de 16×9= **144 estados distintos**.

El generador por tanto, itera sobre las posiciones relativas posibles del enemigo y todas las configuraciones de accesibilidad, creando una lista de instancias de Estado con un identificador único para cada uno. Esto asegura que ningún escenario relevante quede fuera del modelo.

```
public class GeneradorEstados
{
    public List<Estado> Estados { get; } = new List<Estado>(); // Lista de todos los es

    public void GenerarEstados()
    {
        int id = 0;

        // Generar todas las combinaciones de los posibles movimientos
        List<bool[]> combinacionesMovimiento = GenerarCombinacionesMovimiento();

        // Generar todas las combinaciones de posición relativa
        for (int x = -1; x <= 1; x++) // Posición relativa del enemigo en X
        {
            for (int y = -1; y <= 1; y++) // Posición relativa del enemigo en Y
            {
                Vector2 posicionEnemigo = new Vector2(x, y);

                // Combinar con todas las combinaciones de accesibilidad
                foreach (bool[] accesibilidad in combinacionesMovimiento)
                {
                    // Crear un nuevo estado
                    Estado estado = new Estado(id, posicionEnemigo, accesibilidad);
                    Estados.Add(estado);

                    id++; // Incrementar el identificador único
                }
            }
        }
    }

    // Método para generar todas las combinaciones de true/false para las direcciones
    private List<bool[]> GenerarCombinacionesMovimiento()
    {
        List<bool[]> combinaciones = new List<bool[]>();

        // 16 combinaciones de true/false
        for (int i = 0; i < 16; i++)
        {
            // Almacenamos los numeros en binario
            combinaciones.Add(new bool[]
            // Convertir el número a combinaciones de true/false directamente
            {
                (i % 2) == 1,         // Norte
                ((i / 2) % 2) == 1,   // Este
                ((i / 4) % 2) == 1,   // Sur
                ((i / 8) % 2) == 1    // Oeste
            });
        }

        return combinaciones;
    }
```

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 mientras respeta las restricciones del entorno.

La clase garantiza que las acciones realizadas sean coherentes con las reglas del juego, evitando movimientos no válidos como atravesar obstáculos. También incluye un algoritmo de navegación para que el enemigo pueda calcular el camino más corto hacia el agente.

Funciones Principales:

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

QMindTrainer

El `QMindTrainer` es el núcleo del entrenamiento del agente, que implementa la interfaz IQMindTrainer, por lo que cuenta con las funciones Initialize() y DoStep().

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

    // 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);

    // Logs detallados para depuración
    Debug.Log($"DoStep - Estado actual: {estadoActual}, Acción elegida: {accion}, Nueva posición: ({nuevaPosicion.x}, {nuevaP

    // Calcular recompensa y actualizar tabla Q si está en modo entrenamiento
    if (train)
    {
        float recompensa = CalcularRecompensa(nuevaPosicion);
        ActualizarQ(estadoActual, accion, recompensa, nuevoEstado);

        Debug.Log($"DoStep - Recompensa obtenida: {recompensa}, Estado previo: {estadoActual}, Estado nuevo: {nuevoEstado}");
    }

    // 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();
    }
    else
    {
        CurrentStep++;
    }
}
```

Este script es responsable de implementar el algoritmo de Q-Learning y gestionar todo el proceso de aprendizaje, desde la inicialización hasta el ajuste de los valores de la tabla Q en función de las recompensas obtenidas.

La clase fue diseñada para permitir un entrenamiento iterativo en el que el agente aprenda a maximizar sus recompensas mediante la interacción con su entorno.

Funcionamiento

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

1. **Inicialización**:
   La función `Initialize` configura los parámetros iniciales, crea la tabla Q y posiciona al agente y al enemigo en celdas aleatorias. También permite cargar la tabla Q desde un archivo CSV para reutilizar datos previos.

```
public void Initialize(QMindTrainerParams qMindTrainerParams, WorldInfo worldInfo, INavigationAlgorithm navigationAlgorithm)
{
    Debug.Log("QMindTrainer: initialized");

    // Inicialización
    parametros = qMindTrainerParams;
    parametros.epsilon = Mathf.Lerp(1f, 0.1f, (float)CurrentEpisode / parametros.episodes); // Ajuste dinámico de epsilon

    mundo = worldInfo;
    algoritmoNavegacion = navigationAlgorithm;

    // Crear tabla Q
    tablaQ = new TablaQLearning(numAcciones, numEstados);

    // Cargar tabla Q
    if (File.Exists(RUTA_TABLA))
    {
        CargarTablaQ();
    }

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

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

2. **Selección de Acciones**:
   La función `SeleccionarAccion` utiliza un enfoque de exploración-explotación. Basado en el valor de `epsilon`, decide si explorar (seleccionando una acción aleatoria) o explotar (eligiendo la acción con el mayor valor Q para el estado actual).

```
private int SeleccionarAccion(int estado, bool explorar)
{
    float probabilidad = UnityEngine.Random.Range(0f, 1f);

    if (explorar && probabilidad < parametros.epsilon)
    {
        // Acción aleatoria
        int accionAleatoria = UnityEngine.Random.Range(0, 4);
        Debug.Log($"SeleccionarAccion - Explorando. Acción aleatoria: {accionAleatoria}");
        return accionAleatoria;
    }
    else
    {
        // Mejor acción según tabla Q
        int mejorAccion = tablaQ.ObtenerMejorAccion(estado);
        Debug.Log($"SeleccionarAccion - Explotando. Mejor acción: {mejorAccion} para estado: {estado}");
        return mejorAccion;
    }
}
```

3. **Ejecución de Acciones**:
   La función `EjecutarAccion` mueve al agente en la dirección elegida, validando si la nueva posición es accesible. Si la celda no es válida, el agente permanece en su posición actual.

```
private CellInfo EjecutarAccion(int accion)
{
    CellInfo nuevaPosicion = GrupoH.Movimiento.MovimientoAgente(accion, AgentPosition, mundo);

    if (!nuevaPosicion.Walkable)
    {
        Debug.LogWarning($"Movimiento inválido detectado. Acción: {accion}. Manteniendo posición...");
        return AgentPosition; // Mantener posición si no es válida
    }

    Debug.Log($"EjecutarAccion - Acción: {accion}, Nueva posición: ({nuevaPosicion.x}, {nuevaPosicion.y})");
    return nuevaPosicion;
}
```

4. **Cálculo de Recompensas**:
   La función `CalcularRecompensa` asigna valores positivos o negativos según diversos factores:
   - **Penalizaciones**: Por moverse a celdas no válidas, acercarse al enemigo(distancia Manhattan) o quedarse inmóvil.
   - **Recompensas**: Por alejarse del enemigo, mantenerse lejos del borde del mapa y moverse hacia celdas válidas.

```
{
    // Constantes ajustadas para recompensas y penalizaciones
    const float RECOMPENSA_BAJA = 10f;     // Recompensa por alejarse ligeramente
    const float RECOMPENSA_MEDIA = 50f;    // Recompensa por una distancia significativa
    const float RECOMPENSA_ALTA = 100f;    // Recompensa máxima (objetivo ideal)
    const float PENALIZACION_BAJA = -10f; // Penalización por acercarse ligeramente
    const float PENALIZACION_MEDIA = -50f; // Penalización por estar cerca
    const float PENALIZACION_ALTA = -100f; // Penalización severa (ser atrapado)

    float recompensa = 0f;

    // Penalización alta si intenta moverse a una celda no válida
    if (!celda.Walkable)
    {
        Debug.LogWarning("Movimiento hacia celda no válida. Aplicando penalización.");
        return PENALIZACION_ALTA; // Penalización moderada por celda no válida
    }

    // Penalización severa si el agente es atrapado por el enemigo
    if (celda.Equals(OtherPosition))
    {
        Debug.Log("El agente fue atrapado por el enemigo. Penalización severa.");
        return PENALIZACION_ALTA; // Penalización máxima por ser atrapado
    }
    if (celda.Walkable && !celda.Equals(OtherPosition))
    {
        recompensa += 5f; // Pequeña recompensa por moverse hacia una celda válida
    }

    int distanciaAlBorde = Mathf.Min(celda.x, mundo.WorldSize.x - celda.x - 1,
                          celda.y, mundo.WorldSize.y - celda.y - 1);

    if (distanciaAlBorde <= 1) // Muy cerca del borde
    {
        recompensa -= 50f; // Penalización por estar junto al borde
    }

    if (distanciaAlBorde >= 3) // Lejos del borde
    {
        recompensa += 10f; // Recompensa por mantenerse lejos del borde
    }
    if (celda.Equals(AgentPosition))
    {
        recompensa -= 20f; // Penalización por quedarse en el mismo lugar
        Debug.Log("Penalización por quedarse quieto.");
    }

    float distanciaEnemigo = celda.Distance(OtherPosition, CellInfo.DistanceType.Manhattan);

    if (distanciaEnemigo <= 3) // Cercanía peligrosa
    {
        recompensa -= 30f; // Penalización por estar demasiado cerca del enemigo
        Debug.Log("Penalización por estar cerca del enemigo.");
    }

    // Cálculo de distancias Manhattan (actual y nueva)
    float distanciaActual = AgentPosition.Distance(OtherPosition, CellInfo.DistanceType.Manhattan);
    float nuevaDistancia = celda.Distance(OtherPosition, CellInfo.DistanceType.Manhattan);

    // Recompensas y penalizaciones basadas en la distancia al enemigo
    if (nuevaDistancia > distanciaActual)
    {
        recompensa += 20f; // Recompensa base por alejarse
        if (nuevaDistancia >= 10)
        {
            recompensa += 30f; // Recompensa adicional por gran distancia
        }
    }
    else if (nuevaDistancia < distanciaActual)
    {
        recompensa -= 15f; // Penalización base por acercarse
        if (nuevaDistancia <= 3)
        {
            recompensa -= 30f; // Penalización adicional por proximidad peligrosa
        }
```

5. **Actualización de la Tabla Q**:
   Utilizando la fórmula de Q-Learning, se actualizan los valores Q en función de la recompensa obtenida y el mejor valor Q del nuevo estado alcanzado.

```
private void ActualizarQ(int estadoActual, int accion, float recompensa, int nuevoEstado)
{

    float qActual = tablaQ.ObtenerQ(accion, estadoActual);
    float mejorQ = tablaQ.ObtenerMejorAccion(nuevoEstado);

    // Fórmula de actualización Q-Learning
    float nuevoQ = qActual + parametros.alpha * (recompensa + parametros.gamma * mejorQ - qActual);
    tablaQ.ActualizarQ(accion, estadoActual, nuevoQ);
    Debug.Log($"Estado: {estadoActual}, Acción: {accion}, Q antes: {qActual}, Q después: {nuevoQ}");

}
```

6. **Reinicio de Episodios**:
   Cuando el agente es atrapado por el enemigo o se alcanza el límite de pasos, se reinician las posiciones y se actualiza el promedio de recompensas acumuladas.

```
private void ReiniciarEpisodio()
{
    // Incrementar el contador de episodios
    episodiosTotales++;

    // Cálculo del promedio usando media ponderada exponencial
    ReturnAveraged = Mathf.Round((ReturnAveraged * 0.9f + Return * 0.1f) * 100) / 100;

    // También puedes calcular el promedio acumulado (opcional)
    sumaDeRetornos += Return;
    float promedioGlobal = Mathf.Round((sumaDeRetornos / episodiosTotales) * 100) / 100;

    Debug.Log($"Episodio {CurrentEpisode} finalizado: Total Reward = {Return}, Averaged Reward = {ReturnAveraged}

    // Actualizar epsilon dinámicamente para reducir exploración con el tiempo
    parametros.epsilon = Mathf.Lerp(1f, 0.1f, (float)CurrentEpisode / parametros.episodes);
    Debug.Log($"Epsilon actualizado: {parametros.epsilon}");

    // Reiniciar variables del episodio
    recompensaTotal = 0f; // Reiniciar acumulador de recompensa
    CurrentStep = 0;
    CurrentEpisode++;

    // Actualizar posiciones iniciales del agente y el enemigo
    AgentPosition = mundo.RandomCell();
    OtherPosition = mundo.RandomCell();

    // Guardar la tabla Q periódicamente
    if (CurrentEpisode % parametros.episodesBetweenSaves == 0)
    {
        GuardarTablaQ();
    }

    // Disparar evento para iniciar un nuevo episodio
    OnEpisodeStarted?.Invoke(this, EventArgs.Empty);
}
```

Persistencia de Datos

Para garantizar la continuidad del entrenamiento:

- La tabla Q se guarda periódicamente en un archivo CSV mediante la función `GuardarTablaQ`.
- La función `CargarTablaQ` permite recuperar valores Q previamente almacenados.

```
public void GuardarTablaQ()
{
    Debug.Log("Tabla Q guardada en archivo CSV.");
    try
    {
        using (StreamWriter writer = new StreamWriter(RUTA_TABLA))
        {
            // Escribir encabezados (opcional)
            writer.WriteLine("Estado,Accion,Q-Valor");

            // Recorrer todos los estados y acciones de la tabla Q
            for (int estado = 0; estado < tablaQ.numEstados; estado++)
            {
                for (int accion = 0; accion < tablaQ.numAcciones; accion++)
                {
                    float qValor = tablaQ.ObtenerQ(accion, estado);
                    // Usar punto como separador decimal
                    writer.WriteLine($"{estado},{accion},{qValor.ToString(CultureInfo.InvariantCulture)}");
                }
            }
        }
        Debug.Log($"Tabla Q guardada exitosamente en {RUTA_TABLA}");
    }
    catch (Exception ex)
    {
        Debug.LogError($"Error al guardar la tabla Q en el archivo CSV: {ex.Message}");
    }
}


public void CargarTablaQ()
{
    try
    {
        using (StreamReader reader = new StreamReader(RUTA_TABLA))
        {
            string line;
            while ((line = reader.ReadLine()) != null)
            {
                var parts = line.Split(',');
                int estado = int.Parse(parts[0]);
                int accion = int.Parse(parts[1]);
                float qValor = float.Parse(parts[2]);
                tablaQ.ActualizarQ(accion, estado, qValor);
            }
        }
        Debug.Log("Tabla Q cargada exitosamente.");
    }
    catch (Exception ex)
    {
        //Debug.LogError($"Error al cargar la tabla Q: {ex.Message}");
    }
}
```

Visualización

Se utiliza `OnGUI` para mostrar en pantalla información relevante sobre el entrenamiento, como el episodio actual, el paso dentro del episodio y las recompensas acumuladas.

QMindTester

La clase `QMindTester` implementa un sistema para que el agente tome decisiones basadas en la tabla Q entrenada previamente utilizando el algoritmo de Q-Learning. Esta clase permite que el agente actúe dentro de un entorno, optimizando sus movimientos para evitar al enemigo y tomar decisiones informadas basadas en la tabla Q.

Funcionamiento

1. **Inicialización**:
    - Se configura el entorno (`mundo`) y se inicializa la tabla Q.
    - Se utiliza la función `CargarTablaQ` para cargar los valores entrenados desde un archivo CSV. Este archivo debe estar previamente generado por el sistema de entrenamiento.

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

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

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

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

2. **Selección de la Próxima Acción**:
    - La función `GetNextStep` determina la siguiente acción que debe realizar el agente, basándose en la tabla Q y la lógica de escape si el enemigo está cerca.
    - Si el enemigo está a una distancia Manhattan de 3 o menos, el agente prioriza un movimiento que aumente la distancia al enemigo, evaluando todas las acciones posibles.
    - En situaciones normales, el agente selecciona la mejor acción según los valores Q almacenados.

```
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);
    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}");

    // Forzar escape si el enemigo está demasiado cerca
    if (distanciaEnemigo <= 3)
    {
        Debug.Log("El enemigo está cerca. Forzando movimiento de escape.");

        // Buscar una acción que aumente la distancia al enemigo
        int mejorAccionEscape = -1;
        float mejorDistancia = float.MinValue;

        for (int accion = 0; accion < numAcciones; accion++)
        {
            CellInfo nuevaPosicion = Movimiento.MovimientoAgente(accion, currentPosition, mundo);

            if (nuevaPosicion.Walkable)
            {
                float nuevaDistancia = nuevaPosicion.Distance(otherPosition, CellInfo.DistanceType.Manhat
                if (nuevaDistancia > mejorDistancia)
                {
                    mejorDistancia = nuevaDistancia;
                    mejorAccionEscape = accion;
                }
            }
        }

        if (mejorAccionEscape != -1)
        {
            Debug.Log($"Acción de escape elegida: {mejorAccionEscape}");
            return Movimiento.MovimientoAgente(mejorAccionEscape, currentPosition, mundo);
        }
    }

    // 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;
```

3. **Cálculo del Estado**:
    - La función `ObtenerEstado` calcula el estado actual combinando la posición relativa del enemigo respecto al agente y la celda específica del agente dentro del mapa. Esto permite seleccionar las acciones apropiadas de la tabla Q para el estado actual.

```
private int ObtenerEstado(CellInfo posicionAgente, CellInfo posicionEnemigo)
{
    // Calcular la posición relativa del oponente respecto al agente
    int deltaX = Mathf.Clamp(posicionEnemigo.x - posicionAgente.x, -1, 1) + 1; // Rango [0, 2]
    int deltaY = Mathf.Clamp(posicionEnemigo.y - posicionAgente.y, -1, 1) + 1; // Rango [0, 2]
    int posicionRelativa = deltaX * 3 + deltaY; // Combinar en un rango [0, 8]

    // Identificar celda única del agente
    int celdaAgente = posicionAgente.x * mundo.WorldSize.x + posicionAgente.y;

    // Combinar la posición del agente con la posición relativa del oponente
    int estado = (celdaAgente * 9 + posicionRelativa) % numEstados;

    // Depuración
    Debug.Log($"ObtenerEstado - Agente: ({posicionAgente.x}, {posicionAgente.y}), Enemigo: ({posicionEnem

    return estado;
}
```

4. **Reacción al Peligro**:
    - Si el enemigo está demasiado cerca, el agente utiliza un algoritmo de escape que evalúa todas las acciones posibles y selecciona aquella que maximiza la distancia al enemigo. Este comportamiento asegura que el agente evite situaciones de alto riesgo.

Persistencia de Datos

La función `CargarTablaQ` lee un archivo CSV que contiene los valores Q entrenados, procesando cada línea para actualizar la tabla Q interna. Incluye validaciones para evitar errores, como líneas vacías o formatos incorrectos.

```
private void CargarTablaQ()
{
    string rutaTabla = "Assets/Scripts/GrupoH/TablaQ.csv";
    if (!File.Exists(rutaTabla))
    {
        Debug.LogError("No se encontró la tabla Q entrenada en la ruta especificada.");
        return;
    }

    try
    {
        using (StreamReader reader = new StreamReader(rutaTabla))
        {
            string line;
            while ((line = reader.ReadLine()) != null)
            {
                // Ignorar encabezados o líneas vacías
                if (string.IsNullOrWhiteSpace(line) || line.StartsWith("Estado")) continue;

                // Separar por comas
                var parts = line.Split(',');
                if (parts.Length != 3)
                {
                    Debug.LogWarning($"Línea inválida: {line}. Formato esperado: estado,acción,qValor");
                    continue;
                }

                // Validar y convertir datos
                if (int.TryParse(parts[0], out int estado) &&
                    int.TryParse(parts[1], out int accion) &&
                    float.TryParse(parts[2], NumberStyles.Float, CultureInfo.InvariantCulture, out float
                {
                    tablaQ.ActualizarQ(accion, estado, qValor);
                }
                else
                {
                    Debug.LogWarning($"Error al procesar la línea: {line}. No se pudieron parsear los val
                }
            }
        }
        Debug.Log("Tabla Q cargada exitosamente.");
    }
    catch (Exception ex)
    {
        Debug.LogError($"Error al cargar la tabla Q: {ex.Message}");
    }
}
```

4. Resultado final

Como resultado final se ha logrado que el agente aguante más de 5000 pasos sin ser capturado.