MEMORIA:
PRACTICA 2

GRUPO G

DESARROLLO DE JUEGOS CON INTELIGENCIA ARTIFICIAL

13 de enero de 2024

Jose Ignacio González Vicente,
Cristina González de Lope
& Pablo García García

ÍNDICE

1. INTRODUCCIÓN Y CONTEXTO.

En la práctica planteada se parte con una escena en el motor de juego "***Unity***", concretamente en la versión: 2022.3.31f1. Dicha escena se encuentra compuesta por un escenario con muros, un enemigo y el agente principal con el que se trabajará en este proyecto.

1.1. PROBLEMA PLANTEADO.

El objetivo principal consta de **entrenar al agente** mediante un algoritmo de **Q-Learning** para que sea capaz de huir del enemigo, que le irá persiguiendo por el mapa, aguantando un número estimado de pasos. El agente, además de huir de él, deberá reconocer las zonas por las que no se puede avanzar, como los muros o las zonas límite, para no caerse del mapa o entorpecer su camino de huida.

Dicho algoritmo de Q-Learning deberá de implementarse junto con una **Tabla Q**, donde se recogerán las acciones y estados del entorno. Por último, habrá que entrenar al agente para que sea capaz de adaptarse a mapas con distinta disposición de los muros y localización inicial del enemigo.

2. ALGORITMO IMPLEMENTADO. CONCEPTOS.

Para desarrollar el algoritmo se ha hecho uso de tres clases: la clase "***QMindTrainer***", la clase "***QMindTester***" y la clase "***State***". Además, se ha utilizado un archivo denominado "**TablaQ.csv**", en el que poder guardar los datos del entrenamiento del agente.

Clase "*QMindTrainer*"

Esta clase es la encargada de implementar el algoritmo de Q-Learning, en ella, se encuentran las funciones detalladas a continuación:

- **Función "*void Initialize*"** → encargada de inicializar el mapa, los algoritmos y parámetros, así como la tabla Q. En el caso de ser la primera vez que se ejecuta, se creará la Tabla Q vacía, sino, se creará y se cargará la Tabla Q guardada en el archivo .csv.

- **Función "*void DoStep*"** → encargada de realizar los pasos que seguirá el agente. Si el agente es pillado por el enemigo, cruza a una zona inaccesible (como un muro o el vacío) o se lleva a más de 1000 pasos se acabará el primer episodio y se calculará la recompensa media que ha obtenido el agente en dicho margen de pasos. Además, se reseteará el mapa y se guardará la Tabla Q en el archivo .csv.

  Mientras no ocurra ninguna de las tres opciones mencionadas, el agente actualizará su posición constantemente (sumando pasos), calculando la recompensa asociada a cada decisión y guardando los datos de los estados, acciones tomadas, recompensa y próximo estado en la Tabla Q.

- **Función "*int SelectAction*"** → encargada de seleccionar la acción más adecuada que realizará el agente basándose en el valor del "***Epsilon***". Su valor de retorno se toma según el estado recibido, haciendo uso de la función "***GetBestAction***". El valor "***Epsilon***" afectará la probabilidad de que el agente tome una acción aleatoria (exploración).

- **Función "*int GetBestAction*"** → encargada de calcular la mejor acción que podría realizar el agente según el estado recibido. Concretamente, recorre las cuatro opciones disponibles para el agente y toma la acción con el valor Q más alto.

- **Función "*float GetQValue*"** → encargada de acceder a la Tabla Q y devolver el **valor Q** asociado a una acción y un estado concretos.

- **Función "*void UpdateQTable*"** → encargada de actualizar la Tabla Q según la acción tomada desde el estado actual del jugador, añadiendo la recompensa obtenida y su estado resultante.

- **Función "*void SaveQTableToCsv*"** → encargada de guardar la Tabla Q sobre la que se ha trabajado en el archivo .csv. Se recorre el diccionario y se asocian los estados y acciones a los valores Q. Cabe destacar que los estados se representan con un "Id".

- **Función "*Dictionary<(State, int), float> LoadQTable*"** → encargada de cargar los valores asociados a la Tabla Q del archivo .csv. Procesa y divide los datos del archivo en columnas (estado, acción, valor de Q). Por último, devuelve la tabla cargada.

- **Función "*State ParseStateFromId*"** → encargada de convertir los "id" asociados a los estados en una cadena de carácteres de la clase State. Por ejemplo, en el caso de una pared al sur, si existe el id se convertirá en un "1" y si no existe se convertirá en un "0".

- **Función "*float CalculateReward*"** → encargada de generar un valor de recompensa, basándose en la posición del agente y del enemigo, así como si choca con un muro, sale del mapa o es alcanzado por el enemigo. La mayor penalización se genera cuando le alcanza el enemigo. Concretamente, en el caso de que el enemigo alcance al agente o atraviese una celda inaccesible (como un muro o el vacío), recibirá una penalización de -100. Así mismo, si se acerca al enemigo recibirá otra penalización de -10, mientras que si se aleja, la recompensa será de +10 (por cada paso).

- **Función "*void ResetEnvironment*"** → encargada de reiniciar el entorno, el agente y el enemigo en posiciones aleatorias, además de restablecer todas los parámetros antes de comenzar un nuevo episodio.

- **Función "*(CellInfo, CellInfo) UpdateEnvironment*"** → encargada de actualizar el entorno del agente y del enemigo teniendo en cuenta la acción realizada por el agente. Tiene como finalidad devolver las nuevas posiciones de ambos.

Clase "*QMindTester*"

Esta clase está diseñada para simular cómo actuaría el agente dentro de un entorno definido por ***WorldInfo*** y haciendo uso de la Tabla Q generada en la clase "***QMindTrainer***". El agente tomará decisiones sobre sus acciones a realizar según el entrenamiento por refuerzo realizado (reflejado en los valores de la Tabla Q).

Se hace uso de la función "***LoadQTable***" para cargar los valores de la Tabla Q desde el archivo QTable.csv. Se inicializa el mundo con las celdas y sus muros, así como las posiciones aleatorias del agente y del enemigo. El agente toma acciones haciendo uso de la función "***GetBestAction***", la cual tiene en cuenta su posición y la del enemigo (además de las celdas inaccesibles). Una vez seleccionada la mejor acción el agente se mueve en función de esta, finalizando en su nuevo estado y recalculando la mejor acción desde la nueva localización.

Clase "*State*"

Esta clase ha sido creada especialmente para manejar los distintos estados en los que se puede encontrar el agente. Se encarga de almacenar la información sobre el entorno y las posiciones del agente y del enemigo.

Almacena la información de las celdas inaccesibles, como es el caso de los muros, según sus direcciones cardinales (Norte, Sur, Este y Oeste). Además de guardar la posición del enemigo, es decir, si se encuentra por encima o a la derecha (si alguno resulta negativo significa que el resultado es su opuesto). Por último, incluye la distancia que separa al agente y a su enemigo.

Tiene métodos como "***Equals***" o "***GetHashCode***", los cuales permiten usar objetos de la clase "***State***" en el diccionario (estructura de datos) y compararlos entre sí. Por otro lado, existe el método "***StateId***", encargado de generar el id que representa cada estado de forma única para diferenciarlo a la hora de guardarlo en la Tabla Q.

3. DETALLES DEL ENTRENAMIENTO.

En cuanto al entrenamiento del agente, se ha optado por lo siguiente:

- **Cambio de estructura del nivel**: durante el entrenamiento se ha variado la disposición del mapa, de esta forma se consigue mejorar la robustez del agente frente a la posible variabilidad de su entorno. Al cambiar el mapa se obliga al agente a generalizar sus estrategias para que sean válidas en cualquier tipo de mapa y a evitar adaptarse a un tipo de disposición concreta. Además, en general, esto hace que todo el entrenamiento sea más flexible, ya que así se permite que el agente tome decisiones óptimas incluso en contextos nuevos o inesperados. Se considera que esto es especialmente importante teniendo en cuenta el método de evaluación de esta práctica.

- **Parametrización**: teniendo en cuenta que el valor de "***Alpha***" controla la velocidad a la que aprende el agente. Mantener este valor constante asegura que el ritmo de aprendizaje sea estable, evitando cambios drásticos. Por otro lado, el parámetro "***Gamma***" determina cuánto peso se asigna a las recompensas futuras frente a las inmediatas. Un valor constante en él permite que el agente mantenga una visión aún más consistente sobre la importancia de planificar sus acciones a largo plazo. Inicialmente, el valor del parámetro "***Epsilon***" era muy alto, incentivando la exploración. Por cada episodio, este valor se ha ido reduciendo (en cantidades muy pequeñas). Esto es debido a que la necesidad de explorar se reduce, y el agente puede confiar más en sus experiencias previas.

4. CÓDIGO IMPLEMENTADO.

Clase “*QMindTrainer*”

```
public class QMindTrainer : IQMindTrainer
{
    public int CurrentEpisode { get; private set; }
    public int CurrentStep { get; private set; }
    public CellInfo AgentPosition { get; private set; }
    public CellInfo OtherPosition { get; private set; }
    public float Return { get; private set; }
    public float ReturnAveraged { get; private set; }
    public event EventHandler OnEpisodeStarted;
    public event EventHandler OnEpisodeFinished;
    private QMindTrainerParams _qMindTrainerParams;
    private WorldInfo _worldInfo;
    private INavigationAlgorithm _navigationAlgorithm;
    private Dictionary<(State, int), float> QTable;
    float total_reward = 0;
    string filePath = "Assets/Scripts/Grupo G/TablaQ.csv";
    int saveRate = 0;


    public void Initialize(QMindTrainerParams qMindTrainerParams, WorldInfo worldInfo,
    INavigationAlgorithm navigationAlgorithm)
    {
        _worldInfo = worldInfo;
        _navigationAlgorithm = navigationAlgorithm;
        _navigationAlgorithm.Initialize(_worldInfo);
        _qMindTrainerParams = qMindTrainerParams;

        QTable = new Dictionary<(State, int), float>();
        QTable = LoadQTable(filePath);
        ResetEnvironment();
    }


    public void DoStep(bool train)
    {
        if (AgentPosition == OtherPosition || !AgentPosition.Walkable || CurrentStep >= 1000)
        {
            ReturnAveraged = Mathf.Round((ReturnAveraged * 0.9f + Return * 0.1f) * 100) / 100;
            OnEpisodeFinished?.Invoke(this, EventArgs.Empty);
            saveRate += 1;

            if (saveRate % _qMindTrainerParams.episodesBetweenSaves == 0)
            {
                SaveQTableToCsv(filePath);
            }

            _qMindTrainerParams.epsilon = Mathf.Max(0.01f, _qMindTrainerParams.epsilon
```

```
            * 0.999f);
            ResetEnvironment();
            return;
        }

        State currentState = new State(AgentPosition, OtherPosition, _worldInfo);
        int action = SelectAction(currentState);
        (CellInfo newAgentPosition, CellInfo newOtherPosition) = UpdateEnvironment(action);
        State nextState = new State(newAgentPosition, newOtherPosition, _worldInfo);
        float reward = CalculateReward(AgentPosition, OtherPosition, newAgentPosition,
        newOtherPosition);
        total_reward += reward;
        Return = Mathf.Round(total_reward * 10) / 10;

        if (train)
        {
            UpdateQTable(currentState, action, reward, nextState);
        }

        AgentPosition = newAgentPosition;
        OtherPosition = newOtherPosition;
        CurrentStep++;
    }


    private int SelectAction(State state)
    {
        if (Random.Range(0f, 1f) < _qMindTrainerParams.epsilon)
        {
            return Random.Range(0, 5);
        }

        return GetBestAction(state);
    }


    private int GetBestAction(State state)
    {
        float maxQValue = float.MinValue;
        int bestAction = 0;

        for (int action = 0; action < 5; action++)
        {
            float qValue = GetQValue(state, action);
            if (qValue > maxQValue)
            {
                maxQValue = qValue;
                bestAction = action;
            }
        }

        return bestAction;
    }
```

```
private float GetQValue(State state, int action)
 {
    return QTable.TryGetValue((state, action), out float value) ? value : 0f;
 }


 private void UpdateQTable(State state, int action, float reward, State nextState)
 {
    float currentQ = GetQValue(state, action);
    float maxNextQ = float.MinValue;

    for (int nextAction = 0; nextAction < 5; nextAction++)
    {
       float nextQ = GetQValue(nextState, nextAction);
       maxNextQ = MathF.Max(maxNextQ, nextQ);
    }

    float updatedQ = currentQ + _qMindTrainerParams.alpha * (reward +
    _qMindTrainerParams.gamma * maxNextQ - currentQ);
    QTable[(state, action)] = updatedQ;
 }


 public void SaveQTableToCsv(string filePath)
 {
    using (StreamWriter writer = new StreamWriter(filePath))
    {
       writer.WriteLine("State Action QValue");

       foreach (var entry in QTable)
       {
          string stateId = entry.Key.Item1.StateId();
          int action = entry.Key.Item2;
          float qValue = entry.Value;
          writer.WriteLine($"{stateId} {action} {qValue}");
       }
    }

    Debug.Log($"QTable saved successfully to {filePath}");
 }


 public Dictionary<(State, int), float> LoadQTable(string filePath)
 {
    using (StreamReader reader = new StreamReader(filePath))
    {
       string header = reader.ReadLine();

       while (!reader.EndOfStream)
       {
          string line = reader.ReadLine();
          string[] parts = line.Split(' ');

          if (parts.Length == 3)
          {
             string stateId = parts[0];
             int action = int.Parse(parts[1]);
```

```
                float qValue = float.Parse(parts[2]);
                State state = ParseStateFromId(stateId);
                QTable[(state, action)] = qValue;
            }
        }

        return QTable;
    }
}


private State ParseStateFromId(string stateId)
{
    bool NWall = stateId[0] == '1';
    bool SWall = stateId[1] == '1';
    bool EWall = stateId[2] == '1';
    bool OWall = stateId[3] == '1';

    bool playerAbove = stateId[4] == '1';
    bool playerRight = stateId[5] == '1';

    int playerDistance = int.Parse(stateId[6].ToString());

    return new State(null, null, null)
    {
        NWall = NWall,
        SWall = SWall,
        EWall = EWall,
        OWall = OWall,
        playerAbove = playerAbove,
        playerRight = playerRight,
        playerDistance = playerDistance
    };
}


private float CalculateReward(CellInfo AgentPosition, CellInfo OtherPosition, CellInfo
newAgentPosition, CellInfo newOtherPosition)
{
    float distance = AgentPosition.Distance(OtherPosition, CellInfo.DistanceType.Euclidean);
    float newDistance = newAgentPosition.Distance(newOtherPosition,
CellInfo.DistanceType.Euclidean);

    if(newAgentPosition == newOtherPosition || !newAgentPosition.Walkable)
    {
        return -100f;
    }

    if (newDistance < distance)
    {
        return -10f;
    }
    else
    {
        return 10f;
    }
}
```

```
    private void ResetEnvironment()
    {
        AgentPosition = _worldInfo.RandomCell();
        OtherPosition = _worldInfo.RandomCell();
        CurrentEpisode++;
        CurrentStep = 0;
        Return = 0f;
        total_reward = 0f;
        OnEpisodeStarted?.Invoke(this, EventArgs.Empty);
        Debug.Log("Reseting environment");
    }


    private (CellInfo, CellInfo) UpdateEnvironment(int action)
    {
        CellInfo newAgentPosition = _worldInfo.NextCell(AgentPosition,

         if (action < 4)
         {
               newAgentPosition = _worldInfo.NextCell(AgentPosition,
               _worldInfo.AllowedMovements.FromIntValue(action));
         }

        CellInfo[] path = _navigationAlgorithm.GetPath(OtherPosition, AgentPosition, 1);
        CellInfo newOtherPosition = path.Length > 0 ? path[0] : OtherPosition;

        return (newAgentPosition, newOtherPosition);
    }
}
```

[Figura 1] Clase "*QMindTrainer*"

Clase “QMindTester”

```
public class QMindTester : IQMind
{
    private Dictionary<(State, int), float> QTable;
    private WorldInfo _worldInfo;
    string filePath = "Assets/Scripts/Grupo G/TablaQ.csv";

    public void Initialize(WorldInfo worldInfo)
    {
        Debug.Log("QMindDummy: initialized");
        _worldInfo = worldInfo;
        QTable = new Dictionary<(State, int), float>();
        QTable = LoadQTable(filePath);
    }


public CellInfo GetNextStep(CellInfo currentPosition, CellInfo otherPosition)
    {
        Debug.Log("QMindDummy: GetNextStep");
        State state = new State(currentPosition, otherPosition, _worldInfo);
        int action = GetBestAction(state);

        if (action < 4)
        {
            return _worldInfo.NextCell(currentPosition,
_worldInfo.AllowedMovements.FromIntValue(action));
        }

        return currentPosition;

    }

    private int GetBestAction(State state)
    {
        float maxQValue = float.MinValue;
        int bestAction = 0;

        for (int action = 0; action < 5; action++)
        {
            float qValue = GetQValue(state, action);
            if (qValue > maxQValue)
            {
                maxQValue = qValue;
                bestAction = action;
            }
        }

        return bestAction;
    }
```

```
    private float GetQValue(State state, int action)
    {
        return QTable.TryGetValue((state, action), out float value) ? value : 0f;
    }

    public Dictionary<(State, int), float> LoadQTable(string filePath)
    {
        using (StreamReader reader = new StreamReader(filePath))
        {
            string header = reader.ReadLine();

            while (!reader.EndOfStream)
            {
                string line = reader.ReadLine();
                string[] parts = line.Split(' ');

                if (parts.Length == 3)
                {
                    string stateId = parts[0];
                    int action = int.Parse(parts[1]);
                    float qValue = float.Parse(parts[2]);

                    State state = ParseStateFromId(stateId);

                    QTable[(state, action)] = qValue;
                }
            }

            return QTable;
        }
    }

    private State ParseStateFromId(string stateId)
    {
        bool NWall = stateId[0] == '1';
        bool SWall = stateId[1] == '1';
        bool EWall = stateId[2] == '1';
        bool OWall = stateId[3] == '1';

        bool playerAbove = stateId[4] == '1';
        bool playerRight = stateId[5] == '1';

        int playerDistance = int.Parse(stateId[6].ToString());

        return new State(null, null, null)
        {
            NWall = NWall,
            SWall = SWall,
            EWall = EWall,
            OWall = OWall,
            playerAbove = playerAbove,
            playerRight = playerRight,
            playerDistance = playerDistance
        };
    }
}
```

[Figura 2] Clase "*QMindTester*"

Clase "*State*"

```
public class State
{
    public bool NWall;
    public bool SWall;
    public bool EWall;
    public bool OWall;

    public bool playerAbove;
    public bool playerRight;

    public int playerDistance;


    public State(CellInfo AgentPos, CellInfo OtherPos, WorldInfo worldInfo)
    {
        if (AgentPos != null && OtherPos != null && worldInfo != null)
        {
            NWall = !worldInfo.NextCell(AgentPos, Directions.Up).Walkable;
            SWall = !worldInfo.NextCell(AgentPos, Directions.Down).Walkable;
            EWall = !worldInfo.NextCell(AgentPos, Directions.Right).Walkable;
            OWall = !worldInfo.NextCell(AgentPos, Directions.Left).Walkable;

            playerAbove = OtherPos.y > AgentPos.y;
            playerRight = OtherPos.x > AgentPos.x;
            playerDistance = BinDistance(AgentPos.Distance(OtherPos,
            CellInfo.DistanceType.Euclidean));
        }
    }


    private int BinDistance(float distance)
    {
        if (distance < 3f) return 0; //Cerca
        else if (distance < 6f) return 1;   //Normal
        return 2; //Lejos
    }


    public override bool Equals(object obj)
    {
        return obj is State state &&
               NWall == state.NWall &&
               SWall == state.SWall &&
               EWall == state.EWall &&
               OWall == state.OWall &&
               playerAbove == state.playerAbove &&
               playerRight == state.playerRight &&
               playerDistance == state.playerDistance;
    }
```

```
    public override int GetHashCode()
    {
        HashCode hash = new HashCode();
        hash.Add(NWall);
        hash.Add(SWall);
        hash.Add(EWall);
        hash.Add(OWall);
        hash.Add(playerAbove);
        hash.Add(playerRight);
        hash.Add(playerDistance);
        return hash.ToHashCode();
    }


    public string StateId()
    {
        string id = "";

        id += NWall ? 1 : 0;
        id += SWall ? 1 : 0;
        id += EWall ? 1 : 0;
        id += OWall ? 1 : 0;

        id += playerAbove ? 1 : 0;
        id += playerRight ? 1 : 0;

        id += playerDistance;

        return id;
    }
}
```

[Figura 3] Clase "*State*"