Algoritmo de búsqueda

MEMORIA
PRÁCTICA 1

Javier Álvarez Pérez
Ana Ordóñez Gragera
Fabio Elías Rengifo García

DESARROLLO DE JUEGOS CON INTELIGENCIA ARTIFICIAL
2024

Índice

Introducción

El principal objetivo de este proyecto ha sido emplear los conocimientos adquiridos sobre algoritmos de búsqueda para la resolución de una serie de problemas planteados en un entorno de **Unity3D**, haciendo uso del lenguaje de programación **C#**.

Estos problemas se basan en lograr que un **personaje encuentre** el camino adecuado para alcanzar una serie de **objetivos** distribuidos por un **mapa** dividido en **celdas**.

Para abordar y resolver los desafíos propuestos, que se tratarán con mayor detalle en este documento, se decidió implementar el algoritmo **A*** .Para el correcto desarrollo de los ejercicios se llevó a cabo la modificación y creación de varios scripts: **NodeClass**, que se encarga de almacenar toda la información necesaria de las celdas del mapa para su uso en el algoritmo; **PathFinding,** donde se implementa el propio algoritmo de búsqueda A* ; y **BaseSearchAgent**, que se encarga de la gestiónlos diferentes objetivos que hay que alcanzar.

Ejercicio 1:
La salida

Este primer desafío consistía en la implementación de un algoritmo de búsqueda adecuado para conseguir que el **personaje** encontrase el camino adecuado para alcanzar la **salida**.

Para ello, el primer paso era seleccionar el algoritmo de búsqueda que sería más adecuado implementar. Teniendo en cuenta los diferentes algoritmos vistos en clase, se optó por hacer uso de **A***, ya que este utiliza heurísticas fuertes y es el más informado, lo que le permite ser más eficiente encontrando el camino óptimo al priorizar las opciones más prometedoras.

A* hace uso de una **función de evaluación** para explorar los caminos más prometedores en búsqueda del óptimo. Esta función de evaluación es una combinación del coste acumulado (g(n)) y una estimación heurística del coste restante hasta el objetivo (h(n)).

$$f(n) = g(n) + h(n)$$

g(n): coste acumulado h(n): estimación heurística

NodeClass

Para la correcta implementación del algoritmo de búsqueda fue necesaria la creación de la clase **Node**. Su función es representar un nodo en el espacio de búsqueda, albergando la información necesaria para la evaluación y reconstrucción del camino óptimo por A.

Por ello la clase tiene los siguientes atributos:

- **Cell**: objeto de tipo **CellInfo** que contiene la información necesaria de la celda, como sus coordenadas o si es o no caminable.
- **Parent**: objeto de tipo **Node**, es una referencia al nodo **predecesor** en el camino, para permitir su reconstrucción y el cálculo del coste acumulado.
- **GCost**: atributo de tipo integer, es el **coste acumulado** desde el inicio hasta el nodo.
- **HCost**: atributo de tipo integer, la **heurística** hasta el objetivo.
- **FCost**: atributo de tipo integer, **coste total** del nodo.

```
1 referencias
public class Node
{
    7 referencias
    public CellInfo Cell { get; } //celda del nodo
    3 referencias
    public Node Parent { get; } //nodo padre de el actual en la ruta
    5 referencias
    public int GCost { get; } // Coste acumulado desde el inicio hasta este nodo
    3 referencias
    public int HCost { get; } // Heurística hasta el objetivo (distancia de Manhattan)
    2 referencias
    public int FCost => GCost + HCost; // GCost + HCost (coste total del nodo)
```

PathFinding

Este script alberga la clase **AStarAlgorithm**, encargada de construir el camino al objetivo, donde se implementa el algoritmo de búsqueda **A***. Para este código se uso como base y se modificó el script **RandomMovement**.

El método principal de esta clase es **GetPath()**, el cual es llamado por el agente para poder encontrar el camino al objetivo. Este método recibe dos CellInfo, el punto de inicio y el objetivo y devuelve un array de CellInfo con el camino a seguir.
Para ello, se implementa el algoritmo ya mencionado, el cual hace uso de las listas de nodos **openList** y **closedList** para gestionar los nodos evaluados y los ya visitados. Desde el nodo inicial va evaluando las celdas vecinas, si no han sido ya visitadas y son caminables, crea un nodo para cada una de ellas, calculando su **coste** y asignándoles un **predecesor**. Si están ya en la openList, **actualiza** su coste en caso de ser menor, y si no esta se inserta.
Se repite el proceso visitando cada vez el nodo con menos coste de la openList hasta llegar al nodo objetivo y devolver el camino **reconstruido**.

PathFinding

```
referencias
ublic CellInfo[] GetPath(CellInfo startNode, CellInfo targetNode)

    var openList = new List<Node>(); // Lista abierta de nodos pendientes por visitar
    var closedList = new HashSet<CellInfo>(); // Lista cerrada de nodos ya visitados

    // Nodo inicial con coste inicial y heurística
    Node start = new Node(startNode, 0, ManhattanDistance(startNode, targetNode));
    openList.Add(start);

    while (openList.Count > 0)//mientras haya nodos por explorar itera
    {
        openList.Sort((a, b) => a.FCost.CompareTo(b.FCost)); //ordena por coste F menor
        Node currentNode = openList[0]; // cge el nodo con F menor
        openList.RemoveAt(0); // lo quita de la lista

        if (currentNode.Cell == targetNode) //si se ha alcanzado el objetivo, devuelve ruta
            return ReconstructPath(currentNode);

        closedList.Add(currentNode.Cell); //y añadimos el nodo a la lista de nodos visitados

        foreach (var neighbor in GetNeighbors(currentNode.Cell)) //obtiene los nodos vecinos de l
        {
            if (closedList.Contains(neighbor) || !neighbor.Walkable) continue; //ignorar celdas v

            int tentativeGCost = currentNode.GCost + 1; //coste a cada casilla vecina es uno
            int hCost = ManhattanDistance(neighbor, targetNode);// heuristuca hasta el objetivo
            Node neighborNode = new Node(neighbor, tentativeGCost, hCost, currentNode); //crea el

            if (openList.Exists(n => n.Cell == neighbor && tentativeGCost >= n.GCost)) continue;

            openList.Add(neighborNode); // Añade el vecino a la lista
        }
    }

    return null; // Ruta no encontrada
```

PathFinding

Esta clase también cuenta con diferentes **métodos auxiliares** necesarios para el correcto funcionamiento de GetPath():

- **ManhattanDistance()**: recibe el CellInfo que se está evaluando y el objetivo, calcula la **heurística** según la **distancia Manhattan** y devuelve el resultado.

```
2 referencias
private int ManhattanDistance(CellInfo a, CellInfo b)
{
    return Mathf.Abs(a.x - b.x) + Mathf.Abs(a.y - b.y);
}
```

- **ReconstructPath():** recibe un nodo y devuelve un array de CellInfo que componen el camino final a recorrer. Para ello va insertando los CellInfo de los nodos en una lista, recorriéndolos a través de su atributo Parent, hasta llegar al nodo de inicio que no tiene predecesor. Por último, invierte la lista para que esté en el orden correcto y se pasa a un array.

```
private CellInfo[] ReconstructPath(Node currentNode)
{
    var path = new List<CellInfo>();
    while (currentNode != null)// Recorre los nodos padres desde el nodo final
    {
        path.Add(currentNode.Cell);// Añade la celda del nodo actual a la ruta
        currentNode = currentNode.Parent;// Avanza al nodo padre
    }
    path.Reverse();// Invierte la ruta para que vaya del inicio al final
    return path.ToArray();// Devuelve la ruta como un array
}
```

PathFinding

- **GetNeighbours()**: recibe el CellInfo que se está evaluando y devuelve una **lista** de CellInfo con los **vecinos**. Estos son calculados gracias a las **coordenadas** del CellInfo actual y la infromación del mundo, **_worldInfo**, esta se inicializa con el método **Initialize()** llamado por el agente y contiene entre otras cosas, todas las celdas del mapa.

```
1 referencia
private IEnumerable<CellInfo> GetNeighbors(CellInfo cell)
{
    int x = cell.x;
    int y = cell.y;
    List<CellInfo> neighbors = new List<CellInfo>();

    // Vecinos (arriba, abajo, izquierda, derecha)
    if (x > 0) neighbors.Add(_worldInfo[x - 1, y]);
    if (x < _worldInfo.WorldSize.x - 1) neighbors.Add(_worldInfo[x + 1, y]);
    if (y > 0) neighbors.Add(_worldInfo[x, y - 1]);
    if (y < _worldInfo.WorldSize.y - 1) neighbors.Add(_worldInfo[x, y + 1]);

    return neighbors;
}
```

Ejercicio 2:
La búsqueda del tesoro

La segunda tarea consiste en la recogida de tesoros. Esto implica que el agente sea capaz de identificar, **recoger** y almacenar cofres de tesoros dispersos por la escena, comenzando **desde cualquier posición** inicial y completando el recorrido hasta alcanzar un punto de meta determinado.

Este tipo de desafío nos servirá para medir la habilidad de nuestro agente para encontrar y recoger objetos en un entorno dinámico, además de su competencia en la toma de decisiones óptimas y la adaptación a distintos escenarios.

Para este apartado, se ha optado por crear dos listas. Una la **lista de zombies**, de la que se hablará más adelante, y una **lista de tesoros**, que desarrollaremos a continuación. La segunda mencionada tendrá prioridad ante la llegada a la meta, es decir, si existe algún tesoro en la escena, nuestro personaje irá siempre primero a por ellos antes de llegar a la salida. En caso de que no haya ninguno, la prioridad será la bandera. Por lo tanto, El orden quedaría:

Ejecución ⟶ Tesoros ⟶ Salida

Empezamos en la clase **BaseSearchAgent** creando las siguientes variables:

```
private List<CellInfo> _treasures; // Lista de cofres en el mundo
3 referencias
public int NumberOfDestinations => _zombies.Count + _treasures.Count; // N
```

- **_treasures:** es una lista que contiene todos los tesoros del mundo.
- **NumberOfDestinations**: es un entero que almacena el número total de destinos, incluyendo tanto zombies como tesoros, aunque ahora nos centraremos en los tesoros.

En el método **Initialize()** añadimos lo siguiente:

```
_treasures = _worldInfo.Targets.ToList();
SetClosestObjective(_worldInfo.FromVector3(Vector3.zero));
```

Esto nos añade los cofres del mundo dentro de nuestra variable _treasures. Además, la función **SetClosesObjective()** inicializa el objetivo con aquel que es más cercano:

```
private void SetClosestObjective(CellInfo currentPosition)
{
    float minDistance = float.MaxValue;
    CellInfo closestObjective = null;

    // Prioridad 1: Si hay zombies, selecciona el más cercano
    if (_zombies.Count > 0)
    {
        foreach (var zombie in _zombies)
        {
            float distance = CalculateEuclideanDistance(currentPosition, zombie);
            if (distance < minDistance)
            {
                minDistance = distance;
                closestObjective = zombie;
            }
        }
    }
```

```
//Prioridad 2: Cuando no haya cofres pero si tesoros, selecciona el más cercano
else if (_treasures.Count > 0)
{
    foreach (var treasure in _treasures)
    {
        float distance = CalculateEuclideanDistance(currentPosition, treasure);
        if (distance < minDistance)
        {
            minDistance = distance;
            closestObjective = treasure;
        }
    }
}
//Prioridad 3: no hay zombies ni tesoros objetivo = meta
CurrentObjective = closestObjective ?? _worldInfo.Exit;
```

Lo que esta función lleva a cabo es seleccionar aquel objetivo que, por prioridad, es el más cercano. La máxima prioridad, ignorando a los zombies, son los **tesoros**, por lo que siempre que haya un número superior a 0 de tesoros en la lista de tesoros, el personaje los recogerá antes de llegar a la salida.

Para calcular aquel objeto con menor distancia al personaje, llamamos a nuestro método **CalculateEuclideanDistance()**, que devuelve la distancia uclidea entre los objetos y el agente.

```
private float CalculateEuclideanDistance(CellInfo a, CellInfo b)
{
    return Mathf.Sqrt(Mathf.Pow(a.x - b.x, 2) + Mathf.Pow(a.y - b.y, 2));
}
```

A continuación, nos adentramos en el siguiente método a ejecutar: **GetNextDestionation()**.

```
// Procesar colisiones con objetos en el camino
HandleCollisions(currentPosition);

// Si alcanzó el objetivo actual se la ruta = null para recalcularla
if (CurrentObjective != null && currentPosition == CurrentObjective)
{
    OnCollisionWithObjective(CurrentObjective);
    SetClosestObjective(currentPosition);
    _path = null;
}
```

Dentro de la función, añadimos una comprobación para ver si el agente ha **alcanzado el objetivo actual** y una comprobación para **manejar las capturas de objetos indeseados (que no son el CurrentObjetive).**

```
private void HandleCollisions(CellInfo currentPosition)
{
    // Si se encuentra con un objetivo aunque no sea el objetivo lo manejar
    //Eliminandolo de la lista correspondiente
    if (_zombies.Contains(currentPosition))
    {
        OnCollisionWithObjective(currentPosition);
    }
    else if (_treasures.Contains(currentPosition))
    {
        OnCollisionWithObjective(currentPosition);
    }
}
```

Este método se llama en cada paso de movimiento para verificar si el agente ha colisionado (es decir, "pasado por encima") con un objetivo. Si encuentra un zombie o un cofre en la posición actual, llama al método **OnCollisionWithObjective** para procesarlo.

Esto permite que el agente interactúe con cualquier objetivo en su camino, incluso si no es el objetivo principal que estaba persiguiendo.

Una vez alcanzado el objetivo actual, llamamos a **OnCollisionWithObjective()**, función que actualiza la lista de objetivos tras la colisión. Si ya no quedan tesoros, se cambiará el objetivo a la salida.

```
3 referencias
public void OnCollisionWithObjective(CellInfo objective)
{
    Debug.Log($"Colisión detectada con: {objective}");

    if (_zombies.Contains(objective))
    {
        //Eliminar zombie de la lista
        _zombies.Remove(objective);
        Debug.Log($"Zombie {objective} atrapado.");
    }
    else if (_treasures.Contains(objective))
    {
        //Eliminar cofre de la lista
        _treasures.Remove(objective);
        Debug.Log($"Cofre {objective} recogido.");
    }
    else
    {
        Debug.Log($"Objetivo no reconocido: {objective.Type}");
    }
    //Actualiza el objetivo al más cercano desde la posición actual
    //(pos del objeto recien capturado)
    SetClosestObjective(objective);
}
```

Finalmente, se limpia la ruta para recalcularla.

```
    _path = null; // Limpiar la ruta para recalcularla
```

Así se ve **GetNextDestination()** completo. Si la ruta es nula se calculará utilizando nuestro algoritmo en **_navigationAlgorithm.**

Si la ruta contiene elementos se descolará el primer elemento de esta que será la siguiente celda en la ruta del agente. Se devuelve este destino al final del método.

```
public Vector3? GetNextDestination(Vector3 position)
{
    CellInfo currentPosition = _worldInfo.FromVector3(position);

    if (_zombies.Count > 0)
    {
        UpdateZombiesList();
    }

    // Procesar colisiones con objetos en el camino al CurrentObjetive
    HandleCollisions(currentPosition);

    // Si alcanzó el objetivo actual se busca el siguiente objetivo más cercano
    // y se iguala la ruta a null para recalcularla
    if (CurrentObjective != null && currentPosition == CurrentObjective)
    {
        OnCollisionWithObjective(CurrentObjective);
        SetClosestObjective(currentPosition);
        _path = null;
    }

    // Recalcular la ruta al nuevo CurrentObjetive
    if (_path == null || _path.Count == 0)
    {
        CellInfo[] path = _navigationAlgorithm.GetPath(currentPosition, CurrentO
        if (path == null) return null;
        _path = new Queue<CellInfo>(path);
    }
    // si la cola _path tiene elementos
    // (si hay puntos pendientes en la ruta hacia el objetivo)
    if (_path.Count > 0)
    {
        //el destino es el primer elemento de la cola
        CellInfo destination = _path.Dequeue();
        //(siguiente celda en la ruta del agente hacia el CurrenteObjetive)
        CurrentDestination = _worldInfo.ToWorldPosition(destination);
    }

    return CurrentDestination;
}
```

Ejercicio 3:
La caza del zombie

El último ejercicio consiste en la caza de zombies. Esto implica que el agente sea capaz perseguir a los enemigos por la escena con el objetivo de alcanzarlos, comenzando **desde cualquier posición inicial**.

Este tipo de desafío nos servirá para medir la habilidad de nuestro agente para encontrar y recoger objetos en un entorno dinámico y en tiempo de ejecución.

Como antes se ha mencionado, se ha creado una **lista para los zombies**. Esta tendrá la mayor prioridad a la hora de ejecutar el programa. Por lo tanto, El orden quedaría:

Ejecución ⟶ Zombies ⟶ Tesoros ⟶ Salida

En la clase **BaseSearchAgent** se añade la variable **_zombies**, que recoge la cantidad de zombies que hay en el mundo.

```
private List<CellInfo> _zombies; // Lista de zombies en el mundo
private List<CellInfo> _treasures; // Lista de cofres en el mundo
3 referencias
public int NumberOfDestinations => _zombies.Count + _treasures.Count;
```

Como antes se mencionó, **NumberOfDestinations** almacena el número total de destinos, incluyendo tanto zombies como tesoros.

En el método **Initialize()** añadimos a _zombies los zombies de nuestro mundo.

```
_zombies = _worldInfo.Enemies.ToList();
_treasures = _worldInfo.Targets.ToList();
SetClosestObjective(_worldInfo.FromVector3(Vector3.zero));
```

En **SetClosesObjective()** tenemos la máxima prioridad en los zombies, por lo que siempre que exista uno, el agente irá a por ellos primero.

```
private void SetClosestObjective(CellInfo currentPosition)
{
    float minDistance = float.MaxValue;
    CellInfo closestObjective = null;

    // Prioridad 1: Si hay zombies, selecciona el más cercano
    if (_zombies.Count > 0)
    {
        foreach (var zombie in _zombies)
        {
            float distance = CalculateEuclideanDistance(currentPosition, zombie);
            if (distance < minDistance)
            {
                minDistance = distance;
                closestObjective = zombie;
            }
        }
    }
    // Prioridad 2: Si no quedan zombies, busca el cofre más cercano
    else if (_treasures.Count > 0)
    {
        foreach (var treasure in _treasures)
        {
            float distance = CalculateEuclideanDistance(currentPosition, treasure);
            if (distance < minDistance)
            {
                minDistance = distance;
                closestObjective = treasure;
            }
        }
    }
```

En **GetNextDestionation()** se añade lo siguiente:

```
if (_zombies.Count > 0)
{
    UpdateZombiesList();
}
```

UpdateZombieList() tiene la tarea de actualizar y mantener una lista válida de los zombies activos en el mundo. Para ello realiza un filtrado de la lista de enemigos en el que se asegura de que:

- Cada enemigo no sea null.
- Cada enemigo tenga un GameObject asignado.
- El GameObject esté activo (activeSelf == true).

Solo los enemigos que cumplen estos criterios se agregan a la lista **_zombies**.

De esta forma el agente siempre tendrá una lista actualizada de los zombies activos, filtrando a los que han sido eliminados o desactivados. Esto ayuda a que el agente evite tratar de interactuar o dirigirse hacia enemigos que ya no están activos.

A su vez, el agente obtiene las celdas actuales de los zombies mediante **_worldInfo.Enemies**. Esta propiedad retorna un array de objetos CellInfo, que contiene información sobre la posición y el tipo de cada celda en el mundo.

```
private void UpdateZombiesList()
{
    //Gestionar que ni worldInfo ni la lista de enemigos sea nula
    if (_worldInfo == null || _worldInfo.Enemies == null)
    {
        return;
    }
    //Filtrar enemigos que no sean null y que tenga un gameObject activo
    _zombies = _worldInfo.Enemies
        .Where(enemy => enemy != null && enemy.GameObject != null && enemy.GameObject.active
        .ToList();

    Debug.Log($"Zombies restantes: {_zombies.Count}");
}
```

Conclusión

En esta práctica grupal, implementamos A* para la búsqueda de tesoros y eliminación de zombies. Un reto clave fue manejar las posiciones dinámicas de los zombies, ya que su movimiento constante requería actualizar su ubicación en cada ciclo. Para resolverlo, recalculamos las rutas del agente hacia el zombie más cercano, actualizando su posición con cada iteración. Cuando el agente capturaba un zombie, este se eliminaba de la lista, permitiendo que el agente se dirigiera al siguiente objetivo.

A su vez a otra complicación a tener en cuenta ha sido gestionar que se eliminaran como targets los elementos que el agente iba cogiendo por el camino a atrapar a su CurrentObjetive. Tras varios intentos implementamos el método HandleCollisions para comprobar los objetos con los que el agente iba colisionando y eliminarlos de sus respectivas listas.

A pesar de estos desafíos, la implementación del algoritmo A* demostró ser eficiente para gestionar múltiples objetivos. El agente fue capaz de adaptarse rápidamente a los movimientos de los zombies y encontrar nuevas rutas hacia los objetivos.

Este enfoque permitió que el agente se moviera de manera efectiva, maximizando la eficiencia en el proceso de captura de zombies y recogida de tesoros.