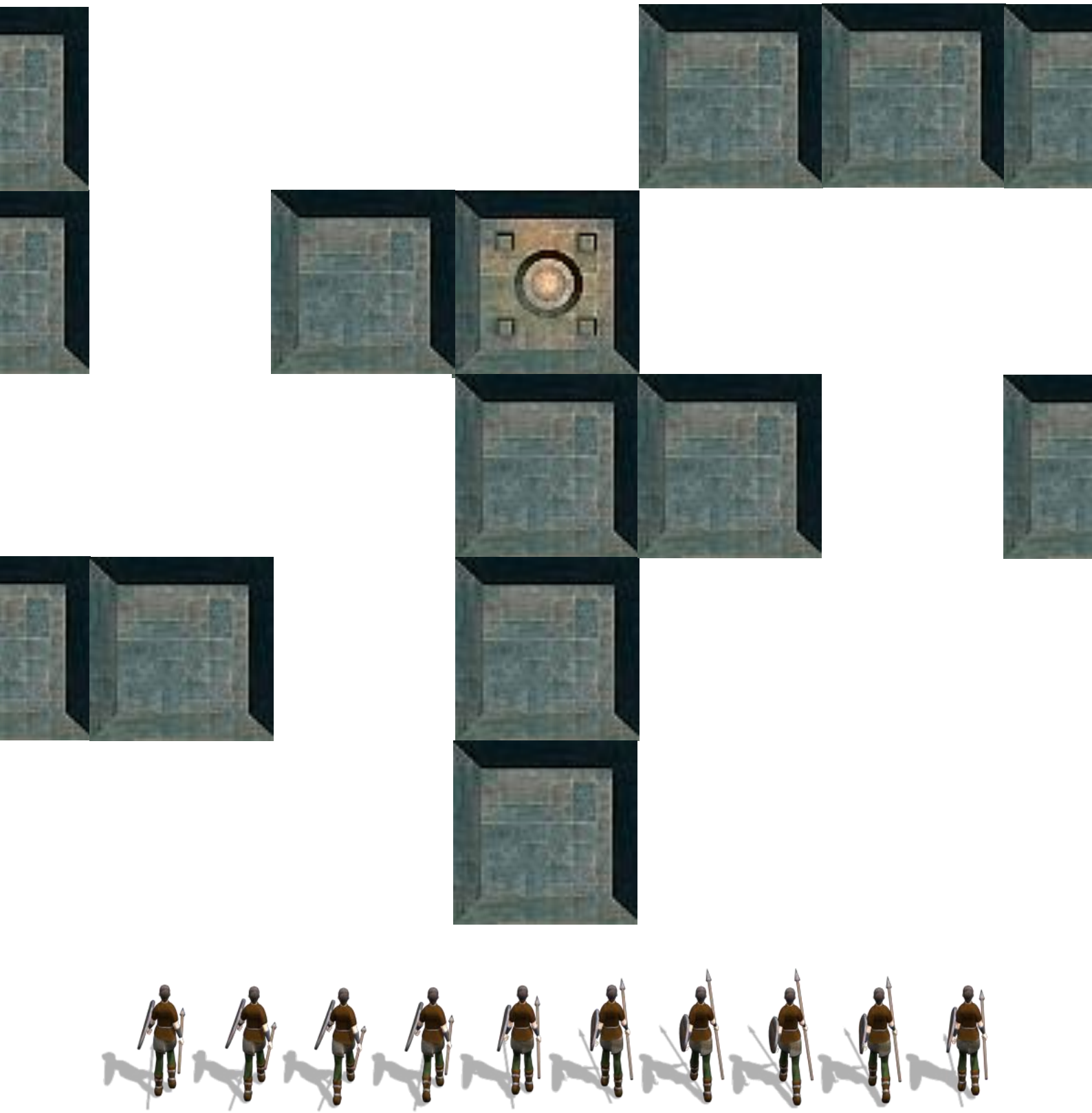
Grupo 4
Sebastián Loges de Faria
Carlos Santayana Vicente

Desarrollo de Juegos con Inteligencia Artificial
Madrid (Quintana)
3º Grado en Diseño y Desarrollo de Videojuegos

Universidad
Rey Juan Carlos

Memoria de la Práctica 1



Índice

1. Descripción del algoritmo para solucionar el problema

1.1. Ejercicio 1. Encontrar salida del laberinto con un algoritmo de búsqueda Offline

El primer problema planteado a resolver es el siguiente.

Disponemos de un **entorno simple**, en concreto un laberinto, en el que nuestro agente debe encontrar la salida:

- Se conoce el estado inicial del agente y el estado meta.
- Se conocen las celdas caminables por el agente.
- Se conoce el coste de transición.

Con esta información, nuestro agente puede utilizar un algoritmo de búsqueda, en nuestro caso A*, con el que crear un árbol de espacios de estados que nos lleve a encontrar la secuencia de operaciones o cambios de estados para pasar del estado inicial al estado meta.

La decisión de **elección de A* (método de búsqueda con heurísticas débiles)** respecto a otros algoritmos se debe a que:

- A diferencia de los métodos de búsqueda con heurísticas fuertes, garantizamos **completitud**.
- Al usar una función h* óptima e informada como lo es la distancia Manhattan, aseguramos **optimalidad** y **reducimos la complejidad** algorítmica de los métodos de búsqueda no informada.

El método de búsqueda A* combina:

- Información del coste para llegar al nodo **n** desde el nodo inicial → **g(n)**.
- Información aproximada del coste restante desde el nodo **n** a la meta → **h*(n)**.

Ambos valores se suman y almacenan en la **f*(n)** del nodo.

Sobre el método general de búsqueda, **A* ordena la lista abierta en función del valor de f*(n)**, de esta manera se expandirán antes los nodos más prometedores, pues al alejarnos de la meta, **f*(n)** crece mucho, y al acercarnos, crece poco.

1.2. Ejercicio 2. Atrapar a los enemigos con un algoritmo de búsqueda Online

El segundo problema planteado a resolver es el siguiente.

Disponemos de un entorno similar al visto anteriormente, con la adición de una serie de enemigos que nuestro agente debe eliminar antes de llegar a la salida del laberinto:

- Se conocen los mismos aspectos indicados en el ejercicio 1.
- Se conoce la posición de los enemigos (serán las metas iniciales).

Para poder afrontar este problema, el agente utilizará un **algoritmo de búsqueda Online** conocido como **Búsqueda por Horizontes**.

Los algoritmos de búsqueda Online se caracterizan por, a diferencia de los Offline, permitir **realizar búsquedas en entornos dinámicos**, es decir, entornos que cambian aunque el agente no actúe sobre ellos, como sucede con el caso de nuestros enemigos. Esto se debe a que se crean árboles de espacios de estados que le ayudan a decidir qué acción realizar en cada paso que tome, en lugar de buscar un plan completo y luego realizarlo.

Existen dos tipos de métodos de búsqueda Online: Búsqueda por Horizontes y Método de Ascenso de Colinas. Siendo el segundo un caso particular del primero con horizonte 1.

Se empleará Búsqueda por Horizontes, puesto que, al permitir visualizar a una mayor profundidad de 1, se reduce el error heurístico que podría descartar un nodo que lleve a la meta por el camino óptimo.

En este algoritmo, se construye un árbol de búsqueda hasta cierto nivel de profundidad **k** mediante un método base, en nuestro caso **Búsqueda en Profundidad**. En función del valor de la **f*** (el más bajo de todos) de los nodos terminales, se realiza una transición hacia uno de los hijos del nodo raíz que llevaría hacia ese nodo terminal.

2. Características de diseño e implementación

2.1. Clase Node común a ambos métodos de búsqueda

Para comenzar, se creó la clase **"Node"** con el fin de tener una estructura de datos donde almacenar la información necesaria de los nodos/estados del juego. Con dicha información, podremos realizar distintos tipos de búsquedas en árboles de espacio de estados. La implementación se muestra a continuación:

```
////////////////////////////// Clase Nodo ////////////////////////////////////
public class Node : IComparable<Node>
{
/////////////////////////// Atributos de la clase ////////////////////////////////

    public Node parent;          //Almacena al padre del nodo
    public int x;                //Almacena la coordenada x de la celda
correspondiente al nodo
    public int y;                //Almacena la coordenada y de la celda
correspondiente al nodo
    public int g;                //Almacena el coste total para llegar al nodo
desde el nodo origen, conocido como g
    public int hStar;            //Almacena el valor de la heurística del nodo,
conocido como h*
    public int fStar;            //Almacena el valor de f*, que es la suma de h* y
g

/////////////////////////// Constructor de la Clase ////////////////////////////////

    public Node(Node parentP,int xP, int yP, int hP)
    {
        parent = parentP;        //Asignamos a la variable parent el nodo que se
le pasa al constructor
        x = xP;                  //Asignamos a la variable x la coordenada de la
celda pasada en el constructor
        y = yP;                  //Asignamos a la variable y la coordenada de la
celda pasada en el constructor

        if (parent != null)      //Comprueba si el padre del nodo instanciado es
distinto de nulo
        {
            g = parent.g + 1;    //Si es distinto de nulo, asignamos a g el valor
g de su padre más el coste de ir al nodo en cuestión(en este caso siempre será 1)
        }
        else
        {
            g = 0;               //Si es nulo, quiere decir que el nodo que
queremos crear es la raíz, por lo tanto, asignamos a g el valor cero
        }

        hStar = hP;              //Asignamos a la variable hStar la heurística
pasada al constructor
        fStar = g + hStar;       //Asignamos a la variable fStar la suma del valor
de su variable g y hStar
    }
```

```
////////// Metodo Compare To, proveniente de la interfaz IComparable ///////////

    public int CompareTo(Node other)    //Este método permitirá ordenar los nodos
en función del valor de sus variables fStar
    {
        if (fStar > other.fStar)        //Si el valor de fStar del nodo es mayor
que el del otro nodo
        {
            return 1;
        }
        else if (fStar < other.fStar)   //Si el valor de fStar del nodo es menor
que el del otro nodo
        {
            return -1;
        }
        else                            //Si el valor de fStar del nodo es igual
que el del otro nodo
        {
            return 0;

        }
    }
}
```

2.2. Clase AStarMind que implementa el algoritmo A*

Una vez implementada la clase vista anteriormente, se creó el script **"AStarMind"** para desarrollar el algoritmo A* y dar solución al ejercicio 1.

Se comenzó declarando las variables necesarias, como puede verse en el siguiente fragmento:

```
////////////////// Clase AStarMind, que emplea el algoritmo A* //////////////////
public class AStarMind : AbstractPathMind
{
    ////////////////////////// Atributos de la clase //////////////////////////

    private List<Node> openList = new List<Node>(); //Lista abierta, donde se
meterán los nodos no evaluados
    private List<Node> plan = new List<Node>();    //Lista plan, donde se guardarán
los nodos que conforman el plan que debe hacer el agente para llegar a la meta
    private List<Node> closedList = new List<Node>();            //Lista cerrada,
donde se meterán los nodos que ya existen en el árbol y así evitar repetidos
    private int numNodesExpanded = 0;           //Variable que almacena el número
de nodos expandidos. Se inicializa a 0
```

Un script externo realiza una llamada a un método conocido como **"GetNextMove"** que recibe como parámetros la información del tablero, la posición actual del agente y un array de metas. Su funcionalidad es la encargada de iniciar el algoritmo A* (en el caso de que el agente no tenga un plan a realizar) y de ejecutar el plan en el caso contrario.

La primera parte de este método se muestra a continuación:

```
////////////////////////// Metodo GetNextMove //////////////////////////
    public override Locomotion.MoveDirection GetNextMove(BoardInfo boardInfo,
CellInfo currentPos, CellInfo[] goals)  //Devuelve el movimiento que debe hacer
el agente
    {
```

```
        if (plan.Count == 0)          //Si la lista plan está vacía
        {
            AStarMethod(boardInfo, currentPos, goals); //Hace una llamada al
método AStarMethod

            return Locomotion.MoveDirection.None; //Le dice al agente que no haga
ningún movimiento
        }
        else //Si la lista no está vacía
        {...
```

El resto del código del método **"GetNextMove"** se verá más adelante.

El método **"AStarMethod"** recibe los mismos parámetros que el método anterior y en este se implementa el algoritmo de búsqueda en sí.

Lo primero que hacemos es crear un booleano auxiliar llamado **"goalReached"** que nos ayudará a detener el bucle del algoritmo. El siguiente paso es calcular la heurística del nodo donde se encuentra el agente y posteriormente crear dicho nodo para meterlo en las listas abierta y cerrada. Se decidió utilizar la suma de distancias Manhattan como heurística, al ser optimista e informada.

En el siguiente fragmento de código puede apreciarse este procedimiento:

```
//////////////////////////// Método AStarMethod ////////////////////////

    public void AStarMethod(BoardInfo boardInfo, CellInfo currentPos, CellInfo[]
goals) //Realiza el algoritmo A*
    {
        bool goalReached = false; //Booleano que indica si se ha llegado a un
nodo meta

        //Primero, creamos el nodo origen, calculamos su heurística
        int heuristic = Math.Abs((goals[0].ColumnId - currentPos.ColumnId)) +
Math.Abs((goals[0].RowId - currentPos.RowId));
// heurística utilizada: Suma de Distancias Manhattan

        openList.Add(new Node(null, currentPos.ColumnId, currentPos.RowId,
heuristic));             // Agregamos el nodo origen creado a la lista abierta
        closedList.Add(openList.ElementAt(0)); //Se añade el nodo a la lista
cerrada
```

Una vez realizado este primer paso, se entra en el bucle del algoritmo, del cual solo se puede salir si se ha encontrado la meta o si la lista abierta se vacía (indicando que no hay solución). Se asigna el primer nodo de la lista abierta a una variable llamada **"currentNode"** y se comprueba si es un nodo meta. En caso de serlo, se agrega a la lista **"plan"**, se pone en true el booleano **"goalReached"** y se muestra por la consola un mensaje. Si no es un nodo meta, se expande dicho nodo con una llamada al método **"expand"**, aumentamos en 1 el número de nodos expandidos y ordenamos la lista abierta con el método **"Sort"** que usa el comparador implementado en la clase **"Node"**. En cada vuelta del bucle se comprueba que el número de nodos expandidos no sea mayor que 1000, ya que si esto ocurre, significa que no se está encontrando una solución.

El fragmento de código correspondiente a la explicación previa es el siguiente:

```
        //Bucle A*
        while(openList.Count != 0 && !goalReached){  //Mientras la lista tenga
nodos dentro y goalReached valga false
            Node currentNode = openList.ElementAt(0); //Almacenamos el primer
elemento de la lista en una variable llamada currentNode

            openList.RemoveAt(0);  //Eliminamos el primer elemento de la lista
```

```
            if (goal(currentNode, goals)) { //Comprobamos si currentNode es meta
con el método goal, en caso afirmativo

                plan.Add(currentNode);     //Agregamos el nodo a la lista plan
                goalReached = true;        //Cambiamos el valor de goalReached de
false a true
                Debug.Log("Meta alcanzada"); //Mostramos un mensaje por la
consola
            }
            else          //En caso de que currentNode no sea un nodo meta
            {
                numNodesExpanded++;                    //Vamos a expandir el
nodo e incrementamos en 1 el número de nodos expandidos
                expand(currentNode, boardInfo, goals); //Usamos el método expand
para expandir el nodo

                openList.Sort();   // Ordenamos la lista abierta con el método
Sort, que usara el método CompareTo de la clase Nodo
            }

//Control de seguridad. Si se expanden más de 1000 nodos salimos del bucle
            if (numNodesExpanded > 1000))
            {
                break;                         //Salimos del bucle
            }
        }
```

El método **"goal"** recibe el nodo actual y la lista de metas. Comprueba si el nodo actual es meta comparando sus coordenadas x e y, tal y como se ve en este fragmento del código:

```
    //////////////////////////// Método goal ////////////////////////////

    public bool goal(Node node, CellInfo[] goal) //Comprueba si el nodo que se
está evaluando es meta
    {
        if (node.x == goal[0].ColumnId && node.y == goal[0].RowId)
//Si las coordenadas del nodo coinciden con las coordenadas de las metas
        {
            return true;
//Devuelve true
        }
        else
//En el caso contrario
        {
            return false;
//Devuelve false
        }
    }
```

El método **"expand"** recibe los mismos parámetros que el método **"AStarMethod"**. Se crea una variable auxiliar llamada **"actualPosition"** con las coordenadas del nodo que se está expandiendo. En un array se almacenan los hijos del nodo, que se obtienen de la llamada al método **"WalkeableNeighbours"**, el cual devuelve 4 objetos de la clase **"CellInfo"**. En el caso de que alguno de estos valga null, quiere decir que ese vecino no es caminable por el agente.

Iteramos sobre el array y si el elemento evaluado no es nulo, calculamos su heurística correspondiente y posteriormente el nodo. Comprobamos si el nodo recién creado está en la lista cerrada (donde se ponen los

nodos que han estado en la lista abierta). En caso de que no esté en ella, un booleano auxiliar se mantendrá en falso y añadirá el nodo a ambas listas, sino, dicho booleano se pondrá en true y no se incluirá el nodo en ninguna lista.

Todo este procedimiento se encuentra a continuación:

```
//////////////////////////// Metodo expand ////////////////////////////

    public void expand(Node currentNode, BoardInfo board, CellInfo[] goals)
//Expande el nodo recibido para obtener los hijos del mismo
    {
        CellInfo actualPosition = new CellInfo(currentNode.x, currentNode.y);
//Usamos una variable actualPosition que tendrá las coordenadas del nodo actual

        //Guardamos los hijos del nodo actual en un array
        CellInfo[] childs = actualPosition.WalkableNeighbours(board);
//En un array de CellInfo guardamos lo que devuelve la función
WalkeableNeighbours, devolviendo nulo si el vecino no es caminable y CellInfo en
caso de que lo sea

        //Creamos los nodos correspondientes a los hijos
        for (int i = 0; i < childs.Length; i++) {

            if (childs[i] != null) {
//Si el elemento evaluado es distinto de nulo

                //Calculo de la heurística del hijo
                int heuristic = Math.Abs((goals[0].ColumnId -
childs[i].ColumnId)) + Math.Abs((goals[0].RowId - childs[i].RowId));

                bool repeatedNode = false;
//Variable auxiliar para determinar si un nodo está repetido

                Node nodeToInsert = new Node(currentNode, childs[i].ColumnId,
childs[i].RowId, heuristic); //Creación del nodo hijo

                foreach (Node node in closedList)
//Se comprueba si el nodo creado está en la closedList
                {
                    if(nodeToInsert.x == node.x && nodeToInsert.y == node.y &&
nodeToInsert.fStar >= node.fStar)    //En el caso de que haya un nodo con las
mismas coordenadas y de peor o igual f* en la lista cerrada
                    {
                        repeatedNode = true;
//Indicamos que se trata de un nodo repetido(Por ejemplo, podría ser el padre)
                    }
                }

                if (!repeatedNode)
//En el caso de que el nodo no esté repetido
                {

                    openList.Add(nodeToInsert);
//Se inserta el nodo en la lista abierta
                    closedList.Add(nodeToInsert);
//Se inserta el nodo en la lista cerrada
                }
            }
        }
    }
```

Cuando salimos del bucle del algoritmo (bien porque hemos hallado una meta o el número de nodos de la lista abierta es muy grande) comprobamos el estado de la lista "**plan**". Si está vacía, mostramos por pantalla que no hay solución. En caso contrario, con un bucle vamos recorriendo el camino desde la meta hasta la raíz con las referencias a los padres, agregando dichos nodos a la lista. Al acabar el bucle, la lista plan está ordenada desde la meta a la raíz, por lo que empleando el método **"Reverse"** reordenamos la lista para que esté en el orden correcto.

Esto le da fin al método **"AStarMethod"** y puede observarse su implementación en este fragmento:

```
if (plan.Count != 0)    //Verificamos, una vez terminado el bucle, si hay algún
elemento en la lista plan, en caso afirmativo
        {
            int i = 0;    //Declaramos una variable auxiliar i que tendrá como
valor inicial cero

            while (plan.ElementAt(i).parent != null)          //Mientras el valor
de la variable parent del nodo actual sea distinto de null
            {
                plan.Add(plan.ElementAt(i).parent);   //Agregamos a la lista plan
al padre referenciado en la variable parent, construyendo el plan desde la meta
hasta el origen
                i++;  //Aumentamos el valor de i
            }

            plan.Reverse();                                    //Al final del
bucle, el plan esta ordenado desde la meta al origen, por lo que usamos el método
Reverse para ordenarlo
        }
        else                                                   //Si la lista plan
está vacía al terminar el bucle
        {
            Debug.Log("No hay solucion");                      //Mostramos por la
consola que no hay solucion
        }
    }
```

Al terminar de ejecutar el método **"AStarMethod"**, en la siguiente llamada al método **"GetNextMove"** la lista "**plan**" no estará vacía, de manera que se decidirán los movimientos a realizar por el agente para seguir dicho plan.

La implementación de la elección del movimiento se muestra en el siguiente código:

```
else                                                  //Si la lista no está vacía
        {
            Node move = plan.ElementAt(0);
//Move toma el valor del nodo que se encuentra en la posición 0 de la lista plan
            plan.RemoveAt(0);
//Eliminamos dicho nodo de la lista plan

            //En este punto, se comprueban las diferencias entre las coordenadas
de la celda del nodo y la de personaje, para determinar el movimiento del agente

            if (currentPos.ColumnId == move.x && currentPos.RowId > move.y)
//Si las coordenadas x son iguales pero la y del jugador es mayor que la del nodo
            {
                return Locomotion.MoveDirection.Down;
//Se le dice al agente que se mueva para abajo
            }
```

```
            if (currentPos.ColumnId == move.x && currentPos.RowId < move.y)
//Si las coordenadas x son iguales pero la y del jugador es menor que la del nodo
            {
                return Locomotion.MoveDirection.Up;
//Se le dice al agente que se mueva para arriba
            }

            if (currentPos.ColumnId < move.x && currentPos.RowId == move.y)
//Si las coordenadas y son iguales pero la x del jugador es menor que la del nodo
            {
                return Locomotion.MoveDirection.Right;
//Se le dice al agente que se mueva a la derecha
            }
            //Si no se cumple ninguna de las condiciones anteriores
            return Locomotion.MoveDirection.Left;
//Se le dice al agente que se mueva a la izquierda
        }
    }
```

2.3. Clase HorizonSearchMind que implementa el algoritmo de Búsqueda por Horizontes

Para solucionar el ejercicio 2, se creó un nuevo script llamado **"HorizonSearchMind"** donde se implementará el algoritmo de Búsqueda por Horizontes.

Se declaró una pila para la lista abierta (ya que usaremos Búsqueda en Profundidad como método base), una lista para almacenar los nodos hoja, una lista para la lista cerrada, una variable que almacenará la acción que ejecutará el agente, una lista de enemigos (donde almacenaremos sus posiciones) y tres variables de tipo int, que almacenan el número de enemigos inicial, el número de enemigos vivos y la profundidad donde estarán los nodos hoja.

Esto puede verse en el siguiente código:

```
////////// Clase HorizonSearchMind, que emplea búsqueda por horizontes utilizando
recorrido en profundidad como base ////////////
public class HorizonSearchMind : AbstractPathMind
{
    /////////// Atributos de la clase ////////////

    private Stack<Node> openList = new Stack<Node>();     //Lista abierta para
cada búsqueda
    private List<Node> closedList = new List<Node>();     //Lista cerrada, donde
se meterán los nodosen el árbol y así evitar repetidos
    private List<Node> treeLeafs = new List<Node>();      //Lista de los nodos
terminales en cada búsqueda
    private Node plan;                                    //Nodo usado para
almacenar la mejor acción a realizar en cada búsqueda
    private List<CellInfo> enemies = new List<CellInfo>();//Lista de celdas de
enemigos (vivos o no)
    private int numEnemies;                               //Entero que almacena
el número de enemigos inicial durante toda la ejecución
    private int enemiesAlive;                             //Entero que almacena
el número de enemigos vivos

    public int depth;                                     //Miembro público que
almacena el horizonte de la búsqueda (modificable desde el editor)
```

Al comenzar el juego, en el método **"Awake"** inicializamos el número de enemigos inicial y el número de enemigos vivos. Posteriormente en el método **"Start"** hacemos una llamada al método **"findEnemies"** que itera sobre la jerarquía de la escena, buscando a los enemigos y agregándolos a la lista de enemigos.

Estas funcionalidades se implementan de la siguiente manera:

```
private void Awake()
    {
        numEnemies = GameObject.Find("Loader").GetComponent<Loader>().numEnemies;
//Almacenamos el número de enemigos especificado en el loader
        enemiesAlive = numEnemies;
//Al principio todos los enemigos están vivos
    }

    private void Start()
    {
        findEnemies();
//Se buscan las celdas de los enemigos en un inicio
    }

    ////////////////////////// Metodo findEnemies //////////////////////////
    public void findEnemies() //Busca los enemigos en la escena al inicio y los
agrega a la lista
    {
        for (int i = 0; i < numEnemies; i++)
//Recorre la lista en función del número de enemigos especificado en el loader
        {
            GameObject enemy = GameObject.Find("Enemy_" + i);
//Se declara una variable que almacena a cada enemigo

            CellInfo enemyCell =
enemy.GetComponent<EnemyBehaviour>().CurrentPosition();  //Se obtiene su celda

            enemies.Add(enemyCell);
//Se añaden las celdas de los enemigos a la lista
        }
    }
```

Similar al ejercicio 1, un script externo hace una llamada al método **"GetNextMove"**, el cual hace una llamada al método **"updateEnemies"** y limpia la lista y la pila de la clase. Luego hace la llamada al método **"HorizonSearchMethod"**.

Estas características se ven en el siguiente código:

```
////////////////////////////// Metodo GetNextMove //////////////////////////////

    public override Locomotion.MoveDirection GetNextMove(BoardInfo boardInfo,
CellInfo currentPos, CellInfo[] goals)  //Devuelve el movimiento que debe hacer
el agente
    {
        updateEnemies();
//Antes de realizar una búsqueda, actualizamos la lista de enemigos
        treeLeafs.Clear();
//y se limpian las listas de nodos hojas y la lista abierta y la lista cerrada
        openList.Clear();
        closedList.Clear();
        horizonSearchMethod(boardInfo, currentPos, goals);
//Hace una llamada al metodo HorizonSearchMethod
...
```

El resto del código del método **"GetNextMove"** se explicará más adelante

El método **"updateEnemies"** se encarga de comprobar si los enemigos de la jerarquía han muerto, observando si en la misma hay un game object con su nombre. Si este no se encuentra, se busca su posición en la lista de enemigos y se pone en nulo, para indicar que está muerto. Si el enemigo sigue vivo, actualizamos su posición en la lista de enemigos:

```
    ////////////////////// Método updateEnemies ////////////////////////////
    public void updateEnemies()
    {
        for (int i = 0; i < numEnemies; i++)
//Recorremos la lista de enemigos
        {
            if (enemies.ElementAt(i) != null)
//Si el enemigo no ha muerto en la lista (es distinto de nulo)
            {
                GameObject enemy = GameObject.Find("Enemy_" + i);
//Lo buscamos en la jerarquía

                if (enemy != null)
//Si sigue vivo en la jerarquía
                {
                    CellInfo enemyCell =
enemy.GetComponent<EnemyBehaviour>().CurrentPosition(); //Obtenemos su celda

                    enemies[i] = enemyCell;
//Actualizamos su celda
                }
                else
//Si está muerto en la jerarquía
                {
                    enemies[i] = null;
//Se pone a nulo en la lista
                    enemiesAlive--;
//Se reduce el número de enemigos vivos
                }
            }
        }
    }
```

El método **"HorizonSearchMethod"** recibe los mismos parámetros que **"GetNextMove"** y su implementación comienza verificando si hay enemigos vivos, de esta forma decidimos si ir a por un enemigo o a la meta. En caso afirmativo, se crea una variable llamada **"enemyPos"** que se inicializará con la posición que devuelva el método **"SelectNeariestEnemy"**. Dicho método comprueba cual es el enemigo más cercano al agente y devuelve su posición:

```
////////////////////// Metodo SelectNearestEnemy ////////////////////////////
    public CellInfo SelectNeariestEnemy(CellInfo currentPos)    //Recibe la celda
actual
    {
        int minDistance = int.MaxValue;
//Se inicializa la distancia mínima a un valor grande
        CellInfo enemyCell = null;
//Esta variable almacenará la celda del enemigo más cercano
```

```
        for (int i = 0; i < numEnemies; i++)
//Para cada enemigo
        {
            if (enemies.ElementAt(i) != null)
//Si está vivo
            {
                CellInfo enemyPos = enemies.ElementAt(i);
//Almacenamos en una variable su celda

                int distanceToEnemy = Math.Abs((enemyPos.ColumnId -
currentPos.ColumnId)) + Math.Abs((enemyPos.RowId - currentPos.RowId));
//Calculamos la distancia a ese enemigo

                if (distanceToEnemy < minDistance)
//Si la distancia calculada es menor que la almacenada en minDistance
                {
                    minDistance = distanceToEnemy;
//Actualizamos minDistance
                    enemyCell = enemyPos;
//Guardamos la celda de ese enemigo en enemyCell
                }
            }
        }

        return enemyCell;
//Devuelve la celda del enemigo más cercano
    }
```

Posteriormente, calculamos la heurística del nodo del agente y creamos su nodo correspondiente, el cual se mete en la lista abierta. Tras realizar este paso, se entra en el bucle del algoritmo, que seguirá ejecutándose mientras la lista no esté vacía. Se comprueba la profundidad del nodo mediante el valor almacenado en su variable **g**, en caso de ser menor que la profundidad de máxima (horizonte), se procede a comprobar si el nodo es meta con el método **"goal"**. Si no es meta, se procede a expandir el nodo. En cambio, si la profundidad es igual a la máxima, se añade dicho nodo a la lista de hojas terminales. Lo mismo ocurre si la comprobación de **"goal"** devuelve true.

La primera parte del método **"HorizonSearchMethod"** se puede ver a continuación:

```
//////////////////////// Metodo HorizonSearchMethod ////////////////////////

    public void horizonSearchMethod(BoardInfo boardInfo, CellInfo currentPos,
CellInfo[] goals)
    {
        if (enemiesAlive > 0)
//Si hay enemigos vivos
        {
            CellInfo enemyPos = SelectNeariestEnemy(currentPos);
//Buscamos el más cercano

            int heuristic = Math.Abs((enemyPos.ColumnId - currentPos.ColumnId)) +
Math.Abs((enemyPos.RowId - currentPos.RowId));   //Heurística utilizada: Suma de
Distancias Manhattan al objetivo (enemigo más cercano)

            openList.Push(new Node(null, currentPos.ColumnId, currentPos.RowId,
heuristic));           //Agregamos el nodo origen creado a la lista abierta
            closedList.Add(openList.ElementAt(0));//Se añade el nodo a la lista
cerrada
```

```
            //Bucle Profundidad
            while (openList.Count != 0)
//Mientras la lista abierta tenga nodos
            {
                Node currentNodeDepth = openList.Pop();
//Sacamos el primer nodo (recorrido en profundidad)

                if (currentNodeDepth.g < depth)
//Si el nodo no está en la profundidad horizonte
                {
                    if (!goal(currentNodeDepth, enemyPos))
//Si no es meta
                    {
                        expand(currentNodeDepth, boardInfo, enemyPos);
//Se expande
                    }
                    else
//Si es meta
                    {
                        treeLeafs.Add(currentNodeDepth);
//No se expande y se añade a la lista de nodos terminales
                    }
                }
                else
//Si el nodo está en la profundidad horizonte
                {
                    treeLeafs.Add(currentNodeDepth);
//Se añade a la lista de nodos terminales
                }
            }
        }
```

La implementación de los métodos **"goal"** y **"expand"** es el mismo que en el ejercicio 1, a excepción de lo siguiente. Dado que el objetivo no siempre era el CellInfo de la meta, sino también, en primer lugar, los enemigos, se cambiaron ciertos aspectos de nomenclatura.

Cuando la lista abierta se vacía y salimos del bucle, comprobamos si la lista de hojas tiene elementos, de ser así, ordenamos los elementos de menor a mayor y asignamos a **"plan"** el primer elemento de la lista. En caso de que la lista de hojas esté vacía, asignamos a la variable **"plan"** el valor null para posteriormente mostrar por consola que no se encontró la acción.

En el caso de que **"plan"** no sea nulo, vamos a actualizar su valor con el padre del nodo que almacena, hasta el nivel de profundidad 1, que representa la acción a realizar.

Estas características se pueden ver en el siguiente fragmento de código:

```
        if (treeLeafs.Count != 0)
//Si la lista de nodos terminales no está vacía
        {
            treeLeafs.Sort();
//La ordenamos en función de su distancia al objetivo y el coste en llegar a ese
nodo (f*)
            plan = treeLeafs.ElementAt(0);
//El plan toma el mejor nodo hoja
        }
        else
//Si la lista está vacía, no hay plan
        {
            plan = null;
        }
```

```
        if (plan != null)
//Si hay plan
        {
            Debug.Log("Accion encontrada");

            if (plan.parent != null)
//Si el plan tiene padre
            {
                while (plan.parent.parent != null)
//Mientras el padre de su padre sea distinto de null
                {
                    plan = plan.parent;
//Hacemos que el plan sea su padre
                }
            }

            //De esta forma, conseguimos que el plan almacene el nodo siguiente a
la raíz con la mejor acción a realizar para llegar al objetivo
        }
        else
//Si no hay plan, no hacemos nada
        {
            Debug.Log("Accion no encontrada");
        }
```

Si no hay enemigos en la escena, los únicos cambios que se realizan respecto al código mostrado previamente en la página 13, es que la meta ahora es la salida del laberinto, y que no se realiza la llamada a **"SelectNeariestEnemy"** ya que estos han muerto.

Cuando termina de ejecutarse el método **"HorizonSearchMethod"**, se comprueba si el plan es distinto de nulo, de ser así, realizamos la acción correspondiente para que el agente se mueva. Si el plan es nulo, el agente no realiza ningún movimiento. Esto está implementado en el resto del código del método **"GetNextMove"** como puede apreciarse en el siguiente fragmento de código:

```
            if (plan != null)  //Si hay plan
            {
            Node move = plan;
//Almacenamos el move el nodo al que se tiene que mover el personaje
            plan = null;
//Limpiamos el plan para la siguiente búsqueda

            if (currentPos.ColumnId == move.x && currentPos.RowId > move.y)
//En función de la posición actual y la posición del nodo al que se debe mover se
elige el movimiento
            {
                return Locomotion.MoveDirection.Down;
            }

            if (currentPos.ColumnId == move.x && currentPos.RowId < move.y)
            {
                return Locomotion.MoveDirection.Up;
            }

            if (currentPos.ColumnId < move.x && currentPos.RowId == move.y)
            {
                return Locomotion.MoveDirection.Right;
            }
```

```
            if (currentPos.ColumnId > move.x && currentPos.RowId == move.y)
            {
                return Locomotion.MoveDirection.Left;
            }

            return Locomotion.MoveDirection.None;
        }
        else
//En caso de no haber plan, no nos movemos
        {
            return Locomotion.MoveDirection.None;
        }
    }
```

3. Discusión sobre los resultados

- En lo que respecta al **primer ejercicio**:

Se realizaron pruebas con diversas semillas, entre ellas la 4, 35, 1222, 2000, 11112023 y 15112023.

Siempre que el escenario se genere de forma correcta, es decir, que la salida del laberinto no esté bloqueada por alguna pared del laberinto, el agente encontrará de manera satisfactoria un plan óptimo para llegar a la meta. Esto sucede en las semillas 4, 35 y 1222.

Para las 3 semillas restantes, o bien la meta está bloqueada por una pared, o bien el agente está encerrado. Lo que hace que el agente se quede quieto en su lugar mientras muestra por consola un mensaje indicando que no hay solución.

- En lo que respecta al **segundo ejercicio**:

Se hicieron pruebas con las mismas semillas que se usaron en el primer ejercicio y la siguiente, 2147483647. Asimismo, la cantidad de enemigos dentro del laberinto fue variando a lo largo de las pruebas.

Nos encontramos con un problema que dificultaba la evaluación del algoritmo: los enemigos lograban salirse del escenario en algunas ocasiones, y esto ocasionaba un error que cerraba el juego.

En las pruebas donde estos problemas no estaban presentes, el agente conseguía llegar hasta los enemigos para poder eliminarlos, aunque en algunas ocasiones, el agente “frena” frente a estos. Cuando ya no quedan enemigos, el agente ejecuta el algoritmo hasta llegar a la meta.

Existe un caso no controlado, en el que el agente puede llegar a la meta sin que los enemigos hayan sido eliminados. Esto ocurre cuando la meta está en el camino que va desde el agente al enemigo más cercano a este.