Universidad Rey Juan Carlos

PROYECTO PRÁCTICO PARTE 1

DESARROLLO DE JUEGOS CON INTELIGENCIA ARTIFICIAL

Realizado por:
Ignacio Molina Casaus
David Ganfornina Alcón

ÍNDICE

1. Contexto

Para la realización de la práctica, se plantean dos escenarios, uno primero en el que el personaje debe de encontrar la meta en un mapa lleno de obstáculos y un segundo en el que el personaje deberá derrotar a un par de enemigos antes de seguir su búsqueda hacia la meta.

Escenario 1.

Escenario 2.

2. Descripción del algoritmo

A la hora de resolver el problema del primer escenario, se ha usado el Algoritmo A*, que como hemos visto a lo largo del curso, busca minimizar el coste estimado total de un camino en el árbol de búsqueda y combina el coste para llegar a cada nodo y el coste aproximado para llegar al nodo meta desde dicho nodo. Por su naturaleza, se trata de una búsqueda Offline, en la cual percibe el entorno, busca el plan más óptimo y, finalmente, lo realiza.
La optimalidad del plan ha sido estimada por la función heurística aprendida en clase “Distancia de Manhattan”:
h*: diferencia de filas entre n y meta + diferencia de columnas entre n y meta. Es decir, el número de movimientos en celdas que debes hacer como mínimo para llegar de una a otra.
La estrategia de este algoritmo por tanto es, elegir de entre las hojas del árbol de búsqueda, el nodo con valor f* mínimo.

Para resolver el escenario 2, al ser necesario usar un método Online ya que los enemigos cambian de posición continuamente haciendo que el entorno esté variando de forma constante, se ha usado el algoritmo de búsqueda por ascenso de colinas, por el que se genera un árbol de búsqueda de un solo un nivel y se elige la hoja más prometedora, repitiendo este ciclo de forma continua. Es decir, se realiza la acción más prometedora.
Tras derrotar a todos los enemigos, el entorno permanece fijo y no es necesaria una búsqueda Online. Por tanto, se pasa a resolver con el algoritmo A*.

3. Características de diseño e implementación

A partir del código base del proyecto de Unity que implementa un movimiento aleatorio para el personaje y el pseudocódigo del algoritmo A* proporcionados por los profesores, se siguieron los pasos y modificaciones que se exponen a continuación para la resolución del problema:

- Modificaciones previas:

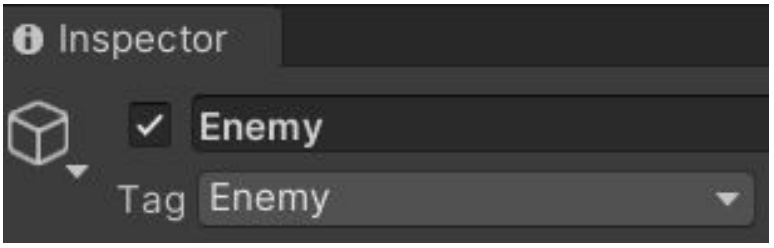
Previamente al desarrollo del algoritmo, se realizaron ciertas modificaciones al proyecto dado para facilitar y corregir ciertos aspectos. Una de ellas, fue la modificación del script `RandomMind`, en concreto, el método `GetNextMove`, ya que este generaba situaciones no deseadas como enemigos atravesando obstáculos. Para esto, simplemente se realizan distintas comprobaciones para determinar que el movimiento a realizar sea posible.

```
3 references
public override Locomotion.MoveDirection GetNextMove(BoardInfo boardInfo, CellInfo currentPos, CellInfo[] goals)
{
    var val = Random.Range(0, 4);
    if (val == 0 && currentPos.RowId < 6) // comprobamos si está en la altura máxima
    {
        if(boardInfo.CellInfos[currentPos.ColumnId, currentPos.RowId + 1].Walkable)
        {
            return Locomotion.MoveDirection.Up;
        }

    }
    if (val == 1 && currentPos.RowId > 0) // comprobamos si está en la altura mínima
    {
        if (boardInfo.CellInfos[currentPos.ColumnId, currentPos.RowId - 1].Walkable)
        {
            return Locomotion.MoveDirection.Down;
        }
    }
    if (val == 2 && currentPos.ColumnId > 0) // comprobamos si está en el ancho mínimo
    {
        if (boardInfo.CellInfos[currentPos.ColumnId-1, currentPos.RowId].Walkable)
        {
            return Locomotion.MoveDirection.Left;
        }
    }
    if (val == 3 && currentPos.ColumnId < 13) // comprobamos si está en el ancho máximo
    {
        if (boardInfo.CellInfos[currentPos.ColumnId + 1, currentPos.RowId].Walkable)
        {
            return Locomotion.MoveDirection.Right;
        }
    }

    // si no se ha podido determinar un movimiento, volvemos a calcularlo
    return GetNextMove(boardInfo, currentPos, goals);
}
```

Otra modificación a destacar, es la de añadir la etiqueta “Enemy” a los prefabs de los enemigos, facilitando su futura búsqueda en la realización de los algoritmos.



- Escenario 1

Aquí se comentará el desarrollo del algoritmo en el caso del primer escenario: llegar del punto A al punto B llevando a cabo el plan más prometedor y evitando todos los obstáculos (muros).

- Clase Nodo

En primer lugar, se creó una clase genérica `Nodo` la cual consta de los siguientes atributos:
Un `CellInfo Cell` que servirá para poder conocer la posición del nodo en el tablero.
Un objeto de la propia clase Nodo `parent` que indicará quién es el padre de cada nodo en el árbol de búsqueda.

Un float `fValue` que almacenará el valor de f* para cada nodo.
Un float `gValue` que almacenará el valor de g para cada nodo.
Un objeto genérico `ProducedBy` que almacenará el movimiento por el cual se llega a dicho nodo.

```
public CellInfo Cell; // celda propia del nodo
public Nodo<T> Parent; // nodo padre del nodo
public float fValue; // f*(n)
public float gValue; // g(n)
public T ProducedBy; // movimiento producido por el nodo
```

Además del constructor, que recibe la celda en la que se encuentra, su padre y la meta a la que debe llegar, para así calcular el f*, consta de dos métodos: `SetFValue` que actualiza el `fValue` de cara a la actualización del valor de g en función de los nodos recorridos, y `FindClosestGoal` que se utilizará para la resolución del escenario 2 y que por tanto se explicará con detalle en dicho apartado.

```
public void SetFValue(CellInfo goal){
    fValue = (Mathf.Abs(goal.ColumnId-Cell.ColumnId) + Mathf.Abs(goal.RowId-Cell.RowId)) + gValue;
}
```

- Creación de la Stack de movimientos y método Repath()

Antes que nada, se declara la Stack de Locomotion.MoveDirection `currentPlan` que va a contener los movimientos necesarios para llegar al objetivo.
Una vez creada esta, se puede definir el método `Repath`, en el cual limpiamos dicha stack, llamando al `Clear` de esta.

```
private Stack<Locomotion.MoveDirection> currentPlan = new Stack<Locomotion.MoveDirection>();
public override void Repath()
{
    currentPlan.Clear();// limpiar Stack
}
```

- GetNextMove

`GetNextMove` es un método que se encarga de rellenar la Stack `currentPlan` creada anteriormente con la secuencia de movimientos para llegar al nodo meta, devolviendo en cada llamada el movimiento a producir para llegar al nodo solicitado.

Este método recibe un objeto CellInfo `boardInfo`, que en este caso será la celda actual en la que se encuentra y de la que queremos conocer el movimiento y otro CellInfo que contiene la meta del camino.
En primer lugar, se comprueba que la Stack no esté vacía y devuelve el siguiente movimiento, llamando a `Pop`.

Cómo en la primera llamada la pila no va a contener nada se pasa al siguiente paso, el cual es invocar al método `Search`, encargado de calcular el camino a seguir, devolviendo el nodo meta, el cual se guarda en una variable `searchResult`.

Con esta variable searchResult, se irá recorriendo hacia atrás desde este nodo meta, usando el atributo de la clase nodo `Parent`, y metiendo en la pila el atributo `ProducedBy`, que si recordamos es el movimiento para llegar a cada nodo, hasta que `searchResult` no tenga padre, es decir, que sea el nodo inicial.

```
// recorre searchResult and copia el camino a currentPlan
while (searchResult.Parent != null)
{
    currentPlan.Push(searchResult.ProducedBy);
    searchResult = searchResult.Parent;
}
```

Una vez rellenada la pila, comprueba que la pila tenga elementos y devuelve el siguiente movimiento, si no, no devuelve el movimiento.

A continuación, se explicará el método `Search`, para así entender mejor el funcionamiento del while que rellena la pila.

- Search

Como fue mencionado anteriormente, `Search` es un método encargado de calcular el camino a seguir devolviendo el nodo con el destino del camino a realizar. Este recibe como parámetros la información del tablero, la celda en la que empieza y un array con las metas (siempre hay una).

Para comenzar, se define la lista abierta junto al nodo inicial, que será el primer elemento de esta lista. Tras esto, hemos definido un contador llamado `notFound` para que, en caso de no haber una solución, salir tras un número arbitrario de iteraciones, evitando así un while infinito que crashea Unity y así, trabajar más ágilmente.

```
1 reference
private Nodo<Locomotion.MoveDirection> Search(BoardInfo board, CellInfo start, CellInfo [] goals) // CellInfo [] goals
{
    // crea una lista vacía de nodos
    var open = new List<Nodo<Locomotion.MoveDirection>>();

    // node inicial
    Nodo<Locomotion.MoveDirection> n = new Nodo<Locomotion.MoveDirection>(start, null, goals[0]);

    // añade nodo inicial a la lista
    open.Add(n);

    //Variable para evitar que Unity crashee en caso de escenarios sin solución
    int notFound = 0;
```

Tras esto, analizaremos la lista cuando esta no esté vacía en un bucle while. En este bucle, se saca el primer nodo de la lista. Si este resulta ser el nodo meta, devolvemos este nodo como valor retornado por `Search`. En caso de que no, continuamos con el bucle.
Si hemos pasado el tope de iteraciones controladas por `notFound`, salimos del bucle con el nodo actual de ese momento. Esto nos ha agilizado bastante el proceso de desarrollo, evitando reiniciar Unity en numerosas ocasiones.

```
// mientras la lista no esté vacía
while (open.Any())
{
    // mira el primer nodo de la lista
    Nodo<Locomotion.MoveDirection> current = open.First();
    open.RemoveAt(0); // Lo sacamos de la lista

    // si el primer nodo es goal, returns current node
    if(current.Cell.CellId.Equals(goals[0].CellId))
    {
        return current;
    }

    // Número arbitrario de bucles antes de encontrar una
    // solución para evitar un while infinito
    if (notFound == 1000)
    {
        Debug.Log("No es posible llegar a goal.");
        return current;
    }
```

Si todo va bien, analizamos las celdas vecinas al nodo actual asignando el movimiento producido por estas según su posición, y si estas no se corresponden con el nodo padre del nodo estudiado, actualizamos el `gValue` sumando al de su nodo padre, el coste de sí mismo, y se añaden a la lista.
Finalmente, ordenamos la lista de cara a los valores de cada nodo y aumentamos el valor de nuestra variable de control `notFound`.

```
// expande vecinos (calcula coste de cada uno, etc)y los añade en la lista
CellInfo[] wN = current.Cell.WalkableNeighbours(board); // vecinos caminables de current

for (int i = 0; i<4; i++)
{
    if(wN[i]!=null){
        Nodo<Locomotion.MoveDirection> aux = new Nodo<Locomotion.MoveDirection>(wN[i], current, goals[0]);
        if (i == 0) aux.ProducedBy = Locomotion.MoveDirection.Up;
        if (i == 1) aux.ProducedBy = Locomotion.MoveDirection.Right;
        if (i == 2) aux.ProducedBy = Locomotion.MoveDirection.Down;
        if (i == 3) aux.ProducedBy = Locomotion.MoveDirection.Left;

        // Añadimos el vecino a la lista comprobando que no sea el padre de current
        if((current.Parent != null)){
            if(aux.Cell.CellId != current.Parent.Cell.CellId && !open.Contains(aux))
            {
                aux.gValue = aux.Parent.gValue + aux.Cell.WalkCost;
                aux.SetFValue(goals[0]);
                Debug.Log("Celda: " + aux.Cell.CellId + " fvalue: " + aux.fValue);
                open.Add(aux);
            }
        } else{
            aux.gValue = aux.Parent.gValue + aux.Cell.WalkCost;
            aux.SetFValue(goals[0]);
            Debug.Log("Celda: " + aux.Cell.CellId + " fvalue: " + aux.fValue);
            open.Add(aux);
        }
    }
}
```

Escenario 2

- ## Clase Nodo

Como se vió en el escenario 1, en la clase nodo se crea un método `FindClosestGoal` que se explicará a continuación.
El método recibe una lista de `CellInfo` “goals”, que almacenará los destinos por los que debe pasar el personaje, además de un booleano `noEnemies` que indica si hay enemigos por destruir o no, y acabará devolviendo el índice del objetivo más cercano.

El método comienza con la inicialización de dos variables, `distance` que almacenará la distancia mínima a un destino, y `closest`, que servirá para almacenar el índice del destino más cercano que se devolverá al final de la implementación.
A continuación, se recorre la lista de `goals`, calculando la distancia para cada uno y devolviendo el índice del destino cuya distancia sea menor teniendo en cuenta que los enemigos tienen prioridad sobre la meta final.

```
// De una lista de destinos, determina el más cercano y devuelve su índice
1 reference
public int FindClosestGoal(List<CellInfo> goals, bool noEnemies)
{
    int distance = 0; // distancia mínima
    int closest = 0; // índice del más cercano
    for(int i = 0; i<goals.Count; i++)
    {
        // cálculo de la distancia al nodo
        int value = (Mathf.Abs(goals[i].ColumnId - Cell.ColumnId) + Mathf.Abs(goals[i].RowId - Cell.RowId));

        // actualizar índice en caso de ser la distancia menor
        if ((distance == 0 || value < distance))
        {
            if((i == 0 && noEnemies || i!=0)){
                distance = value;
                closest = i;
            }
        }
    }

    return closest;
}
```

- FindEnemies

Es un método privado que recibe una lista de CellInfos, además de un booleano `noEnemies` que indica si existen enemigos y se encarga de rellenar dicha lista con los enemigos que se encuentren.
En primer lugar, se buscan objetos con el tag “Enemy”, y en el caso de encontrar, el booleano `noEnemies` pasa a ser false y se rellena un array de `GameObjects enemies` con los enemigos que encuentre.
A continuación se recorre el array y se añade a nuestra lista `destinations`, creada anteriormente, las posiciones de cada uno de ellos.

```
// Con entrada de una lista de destinos y una booleana noEnemies,
// insertamos los enemigos encontrados en la lista de destinos
1 reference
private void FindEnemies(List<CellInfo> destinations, ref bool noEmenies)
{
    if (GameObject.FindGameObjectWithTag("Enemy")) // buscamos objetos enemy
    {
        noEmenies = false; // al haber enemigos, noEnemies es falsa
        GameObject[] enemies = GameObject.FindGameObjectsWithTag("Enemy");// array para almacenar enemigos

        for(int i = 0; i<enemies.Length; i++){
            destinations.Add(enemies[i].GetComponent<Locomotion>().CurrentPosition());
        }
    }

}
```

- FindClosestDestination

Método que recibe el nodo actual, la lista de Cellinfos con los destinos y el booleano `noEnemies`, y se encarga de llamar al método `FindClosestGoal` del nodo recibido en concreto que se vio implementado en la clase `Nodo`.

```
// devuelve la celda de la meta más cercana
1 reference
private CellInfo FindClosestDestination(Nodo<Locomotion.MoveDirection> nodo, List<CellInfo> goals, bool noEnemies)
{
    return goals[nodo.FindClosestGoal(goals, noEnemies)];
}
```

- Search

Tras las modificaciones vistas y la inclusión de nuevos métodos para la identificación de los enemigos, es necesario cambiar el método `Search` que implementamos en el Escenario 1.
La primera diferencia que se encuentra es la declaración del varias veces mencionado booleano `noEnemies` a true y la creación de la lista de `CellInfos` que serán los destinos.

A continuación añadimos la meta a dicha lista, que como en el escenario 1, está guardada en `goals[0]`, llamamos al método `FindEnemies` que se encarga de meter a los enemigos en dicha lista, y se guarda en el `CellInfo closestGoal` el destino más cercano usando la llamada al método `FindClosestDestination` visto anteriormente.
Una vez se tiene esta celda `closestGoal` que almacena el destino al que queremos llegar en esa búsqueda, bastará con usarla en vez de la anterior `goals[0]`.

```
// booleana que devuelve verdadero cuando no hay enemigos
bool noEnemies = true;

// lista de celdas que serán destinos
List<CellInfo> destinations = new List<CellInfo>();

destinations.Add(goals[0]); // añadimos la meta
FindEnemies(destinations, ref noEnemies); // encontramos a los enemigos
CellInfo closestGoal = FindClosestDestination(n, destinations, noEnemies); // el destino sobre la que buscar el camino
// mientras la lista no esté vacia
```

Aquí se ramificará el código en función de si hay enemigos o no:

- Si hay enemigos, se usará el algoritmo Hill Climbing en el método `HillClimb`.
- Si no hay enemigos, se usará el algoritmo A* en el método `AStar`.

```
if(!noEnemies){
    return HillClimb(open, board, closestGoal, goals[0]); // algoritmo Hill Climb
} else{
    return AStar(open, board, closestGoal, notFound); // algoritmo A*
}
```

- AStar

El A* es el algoritmo que utilizamos en el Escenario 1, en el cual no hay enemigo alguno. Como en el Escenario 2, habrá un momento en el que se eliminen a todos los enemigos, es conveniente llevar a cabo este algoritmo en el método `AStar` debido a lo preciso que resulta. Por tanto, el código de `AStar`, consta del while existente en el `Search` del Escenario 1.

- HillClimb

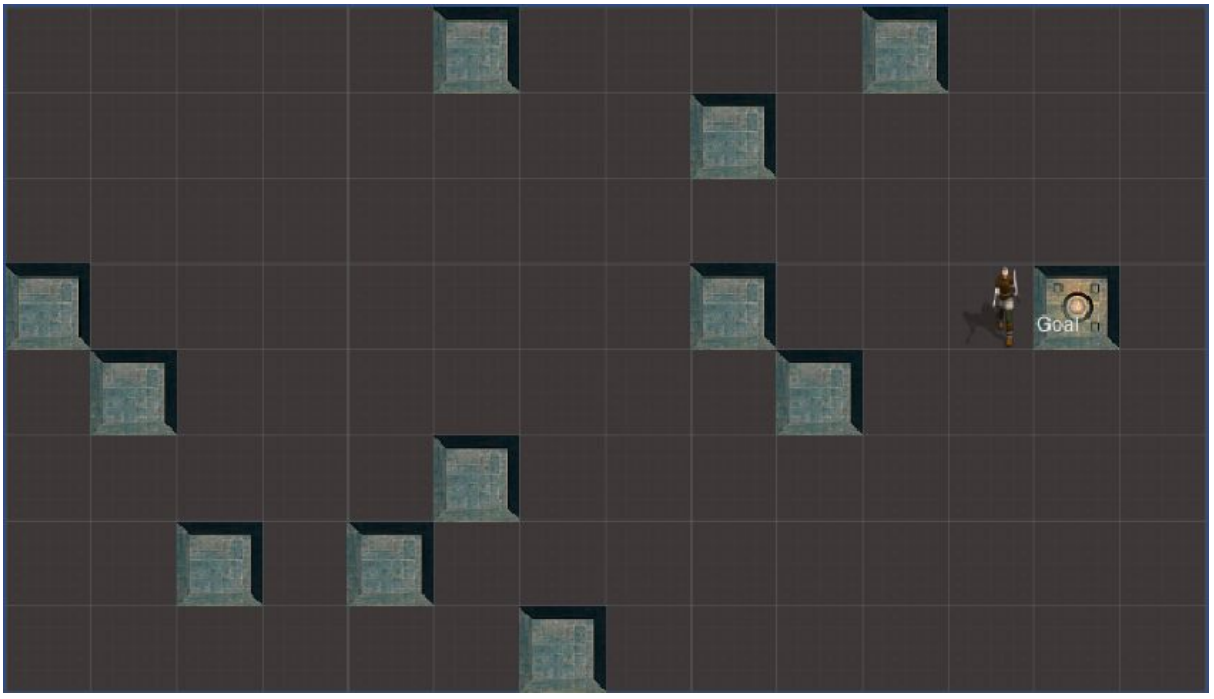
Como los enemigos están en constante movimiento, es conveniente percibir el entorno para posteriormente ejecutar una acción, siendo esto más óptimo que calcular un plan entero.
El código de `HillClimb` es muy similar al de `AStar` pero, en lugar de devolver el nodo meta tras recorrer todo el camino hasta este, simplemente devuelve el nodo vecino más cercano al enemigo más cercano. Resumiendo, es el mismo código pero sin el while.

Además, de cara a ciertos resultados obtenidos, fue realizada otra modificación al código del `HillClimb` respecto al `AStar`. Esto será comentado en la discusión sobre los resultados obtenidos.

4. Discusión sobre los resultados obtenidos

Tras la ejecución de múltiples semillas, comprobamos como el personaje llega a la meta y completa el recorrido, y por tanto, se confirma que el algoritmo A* se ha implementado correctamente. Como vemos en las siguientes capturas del Escenario 1, el personaje se dispone a entrar en la meta tras un recorrido exitoso.

Seed: 1357



Seed: 777912

En el escenario 2, podemos añadir tantos enemigos como se quiera, que siempre los elimina a todos finalizando en la meta.

Seed: 1234 Enemies: 23

A pesar de la correcta implementación, nos encontramos con varios casos a resaltar en los que el programa no se comporta de la forma esperada:

Meta, ¿estás ahí?

Como la generación del mapa con los obstáculos es aleatoria dependiendo de cada semilla, hay semillas, como es el caso de la 8 para este ejemplo, que producen que la meta se genere encima de un obstáculo, por tanto el personaje entraría en un bucle infinito para encontrarla y Unity se colgaría. Como se ha explicado en la clase `Search`, se crea una variable `notFound` para así evitarlo y que aun así se ejecute, para poder ver como se aprecia en la figura, que el mapa no se ha creado correctamente.

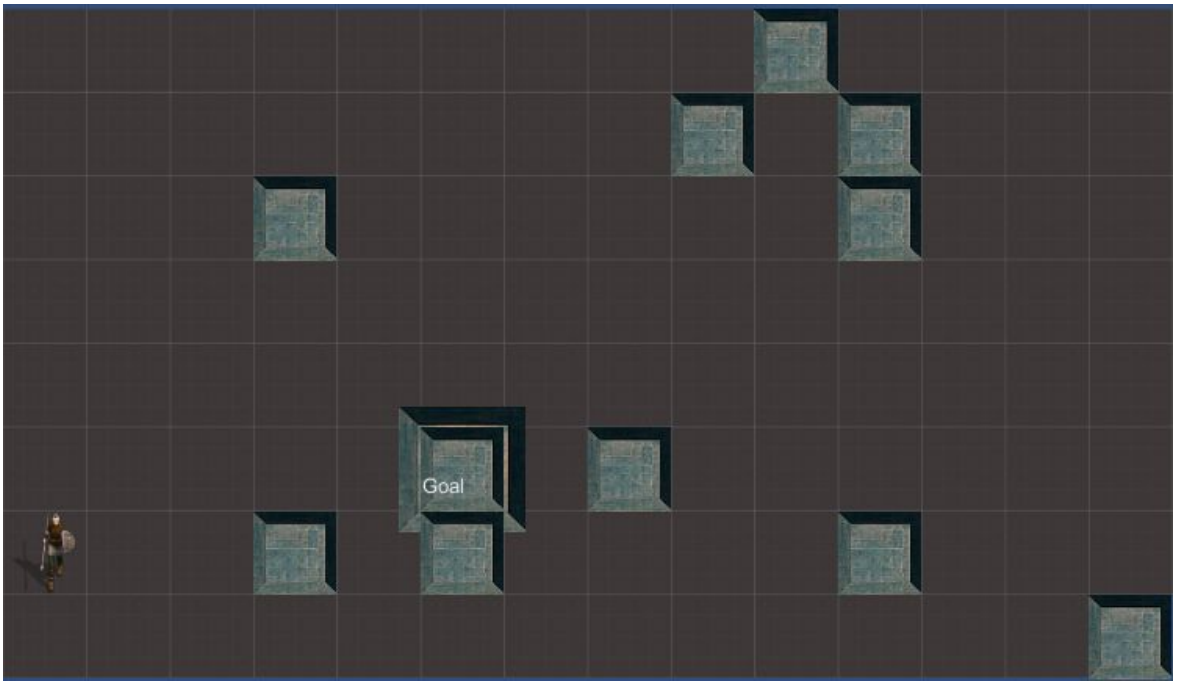


Figura 2. AStarMind Seed: 8

No tan rápido, amigo.

A la hora de ejecutar el Escenario 2, dependiendo de la semilla se podía dar el caso de que el personaje se dirigiese a un enemigo antes de llegar a la meta final, pero el trayecto elegido por el algoritmo pasase por esta misma, por tanto al no detectar esta como una celda NotWalkable, como podemos ver en la figura 3 ejecutando la semilla 7, el personaje en ese punto decidió girar a la derecha para encaminarse hacia su enemigo, entrando en la meta y acabando por tanto la partida antes de tiempo.

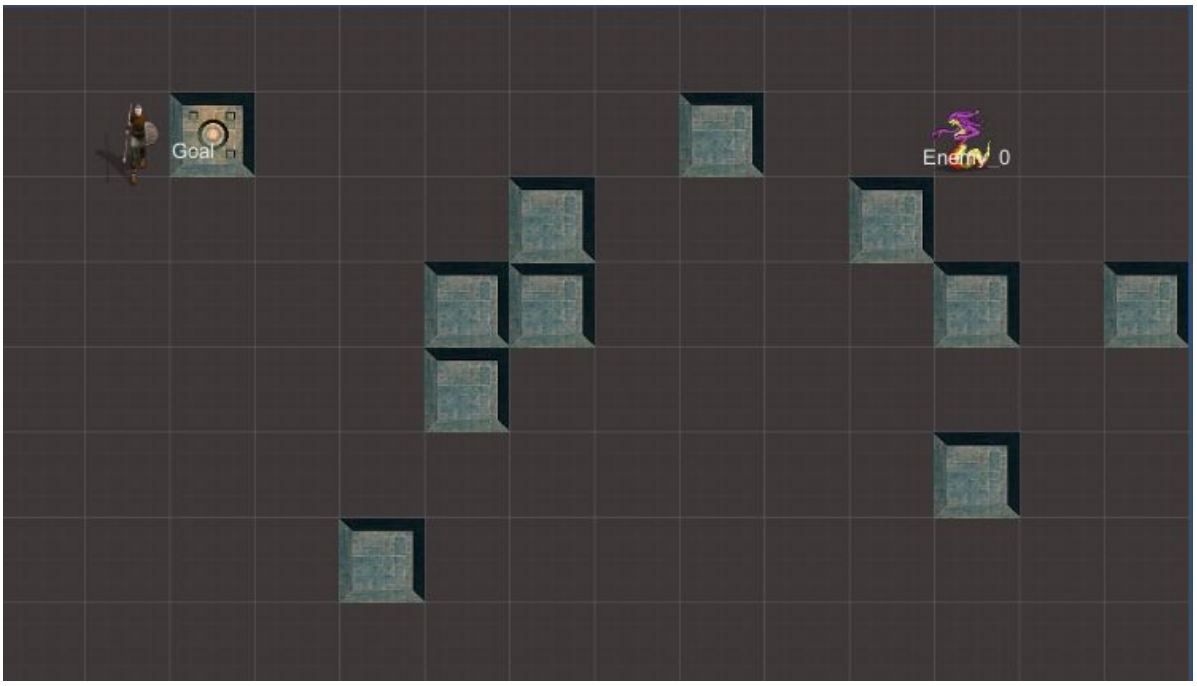


Figura 3: SolutionEnemy Seed: 7

Para solucionarlo, antes de meter la celda en la lista open en el algoritmo de Hill Climbing se comprueba que la celda no coincida con la meta. Por tanto, al ejecutar ahora esta misma semilla, el personaje evitará la meta e irá hacia el enemigo para posteriormente acabar.

```
if ((current.Parent != null) && aux.Cell.Walkable && !aux.Cell.CellId.Equals(finalGoal.CellId))
{
        aux.gValue = aux.Parent.gValue + aux.Cell.WalkCost;
        aux.SetFValue(closestGoal);
        open.Add(aux);
}
else{
    if(aux.Cell.Walkable && !aux.Cell.CellId.Equals(finalGoal.CellId)){
        aux.gValue = aux.Parent.gValue + aux.Cell.WalkCost;
        aux.SetFValue(closestGoal);
        open.Add(aux);
    }
}
```

Atascado en el muro.

A la hora de implementar el algoritmo de Hill Climbing, hay ocasiones en las que dependiendo de la semilla, como vemos en la figura 4 con la semilla 1222, el personaje queda atrapado entre varios muros repitiendo el mismo movimiento de una celda a otra, hasta que el enemigo cambia de posición y se consigue salir, acabando el camino.
Esto se debe a que una vez en esa situación puntual de estar rodeado entre muros y estando el enemigo en cierta posición, la celda con mejor valor es continuamente una de las dos en las que se produce la reiteración, debido a que no se puede mover hacia la derecha por el muro. Por la naturaleza del código, en caso de empate, el orden de preferencia de movimientos es arriba, derecha, abajo, y finalmente, izquierda, dando así prioridad a algunos movimientos en concreto.
Este problema acaba cuando el enemigo se mueve y ahora sí el personaje encuentra que la celda de arriba es mejor que volver a la anterior, acabando el problema.
Lo óptimo que resulta el algoritmo de Hill Climbing se compensa con distintos inconvenientes como el mostrado en este apartado, que podría solucionarse con otros algoritmos que supondrían un mayor coste computacional.

Figura 4. SolutionEnemy Seed: 1222

Conclusión

Para concluir, nos gustaría resaltar que la realización de la práctica nos ha ayudado a afianzar los conceptos del tema de la mejor forma posible: mediante la práctica y puesta en escenarios potencialmente útiles para nuestros proyectos. Además, la experiencia de programar nuestra primera Inteligencia Artificial ha sido enriquecedora y satisfactoria a la par, teniendo así la vista puesta en profundizar en este campo y en descubrir sus distintas ramas existentes.

Con la realización del trabajo, hemos conseguido organizar las tareas para así realizar un trabajo equitativo entre ambos, comunicándonos en todo momento y manteniéndonos informados de las ideas que se nos ocurrían y cambios que se realizaban ya que, como hemos visto en esta práctica, con una buena planificación, probablemente se obtendrán resultados óptimos.