Facultad
de
Ciencias

Desarrollo de un videojuego de mazmorras multiagente

(Development of a multi-agent dungeon crawler video game)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Antonio Solana Suárez

Directores: Camilo Palazuelos Calderón

Julio-2024

Índice

1 Introducción **1**
- 1.1 Objetivo . . . 2
- 1.2 Análisis de requisitos . . . 2
  - 1.2.1 Requisitos funcionales . . . 3
  - 1.2.2 Requisitos no funcionales . . . 3
- 1.3 Herramientas utilizadas . . . 3
- 1.4 Unity . . . 3
  - 1.4.1 Conceptos básicos . . . 4
  - 1.4.2 Lucidchart . . . 5
  - 1.4.3 Visual studio . . . 5
- 1.5 Estructura del trabajo . . . 5

2 Conocimientos previos **6**
- 2.1 Algoritmos de búsqueda de rutas . . . 6
  - 2.1.1 Búsqueda en profundidad . . . 6
  - 2.1.2 Búsqueda en anchura . . . 6
  - 2.1.3 A* . . . 7
- 2.2 Algoritmos basados en Minimax . . . 9
  - 2.2.1 Poda alfa-beta . . . 9
- 2.3 Arboles de decisión . . . 10
- 2.4 Arboles de comportamiento . . . 11
  - 2.4.1 Tareas compuestas . . . 12
- 2.5 Algoritmo de barajado Durstenfeld . . . 13

3 Desarrollo del juego **15**
- 3.1 Proyecto de Unity roguelike . . . 15
- 3.2 Comprensión general del funcionamiento del juego . . . 15
- 3.3 Generación del mapa . . . 16
  - 3.3.1 Generación lógica del mapa . . . 16
  - 3.3.2 Generación de mapa visual . . . 17
- 3.4 GameManager . . . 18
- 3.5 Entidades . . . 19
  - 3.5.1 Instanciacion de entidades . . . 20
- 3.6 BattleManager . . . 21
  - 3.6.1 Introducción . . . 21
  - 3.6.2 Gestión de turnos . . . 21
  - 3.6.3 Control automático de los enemigos y del jugador . . . 21

4 Modo desarrollador **22**
- 4.1 Herramientas de desarrollador . . . 22
  - 4.1.1 Añadir enemigos . . . 23
  - 4.1.2 Eliminar enemigo . . . 23
  - 4.1.3 Dibujar Baldosas . . . 23
  - 4.1.4 Mover Entidades . . . 24

5 Control de los enemigos **26**
5.1 Voraz . . . 26
5.1.1 Voraz A* . . . 26
5.2 Poda alfa-beta . . . 27
5.3 Arboles de decisión . . . 28
5.3.1 Árbol de decisión enemigos a larga distancia . . . 29
5.3.2 Árbol de decisión enemigos a corto alcance . . . 31
5.4 Arboles de comportamiento . . . 31

6 Pruebas **32**
6.1 GameManagerTest . . . 32
6.2 Autoplay . . . 33
6.3 Resultados . . . 33
6.4 Análisis de daño recibido por el jugador . . . 33
6.4.1 Resultados de la ejecución con pocos enemigos . . . 33
6.4.2 Resultados de la ejecución con muchos enemigos . . . 34
6.5 Análisis número de turnos empleados . . . 34
6.5.1 Análisis con pocos enemigos . . . 34
6.5.2 Análisis con muchos enemigos . . . 34
6.5.3 Análisis resultados . . . 38

7 Conclusiones y futuros trabajos **39**
7.1 Conclusión . . . 39
7.1.1 Futuros trabajos . . . 39

Resumen

La industria de los videojuegos es uno de los sectores más demandados dentro del mundo del ocio. Dentro de esta industria se encuentran diversos tipos de juegos, como juegos de deportes, de conducción etc.

Una parte importante de un videojuego es el nivel de desafío que este ofrece, ya que tiene un impacto significativo en la experiencia del usuario.

El nivel de desafío depende en gran medida depende de cómo de inteligentes son percibidos los enemigos por el usuario. Por lo que unos enemigos desafiantes serán imprescindibles para una buena experiencia del usuario. Ya que si los enemigos son demasiado habilidosos o muy torpes resultará en que el jugador sienta frustración o aburrimiento respectivamente.

El objetivo de este trabajo es el desarrollar un juego en el que un jugador se enfrente a múltiples enemigos, que sean capaz de ofrecer un nivel de desafío adecuado. Para esto, una vez desarrollado el juego, se programarán y probarán diversos algoritmos de toma de decisiones para los enemigos, analizando su comportamiento e identificando las fortalezas y desventajas que estos ofrecen.

Palabras clave: Videojuegos, Toma de decisiones, Algoritmos, Múltiples Enemigos, Enemigos Desafiantes

Abstract

The video game industry is one of the most relevant sectors in the world of entertainment. Within this industry there are several types of games, such as sports games, driving games, etc.

The level of challenge that a videogame offers it is very important, as it has a significant impact on the user experience

The level of challenge depends on how intelligent the enemies are perceived by the user. So challenging enemies will be a must for a good user experience. If the enemies are too skilled or too clumsy it will result in the player feeling frustration or boredom respectively.

The objective of this work is to develop a game in which a player faces multiple enemies, which are able to offer an adequate level of challenge. For this, once the game is developed, several decision-making algorithms for the enemies will be programmed and tested, analyzing their behavior and identifying the strengths and disadvantages they offer.

Keywords: Video game, Decision-making, Algorithms, Challenging Enemies, Multiple Enemies.

Capítulo 1

Introducción

Los videojuegos se han convertido en uno de los sectores del ocio más importantes de España. Llegando a facturar en 2022, 2012 millones de euros [1], cifra que se encuentra muy por encima de los 367.5 millones facturadas por la industria del cine [2] y que la música grabada tanto en formato físico como digital con 462 millones [3].

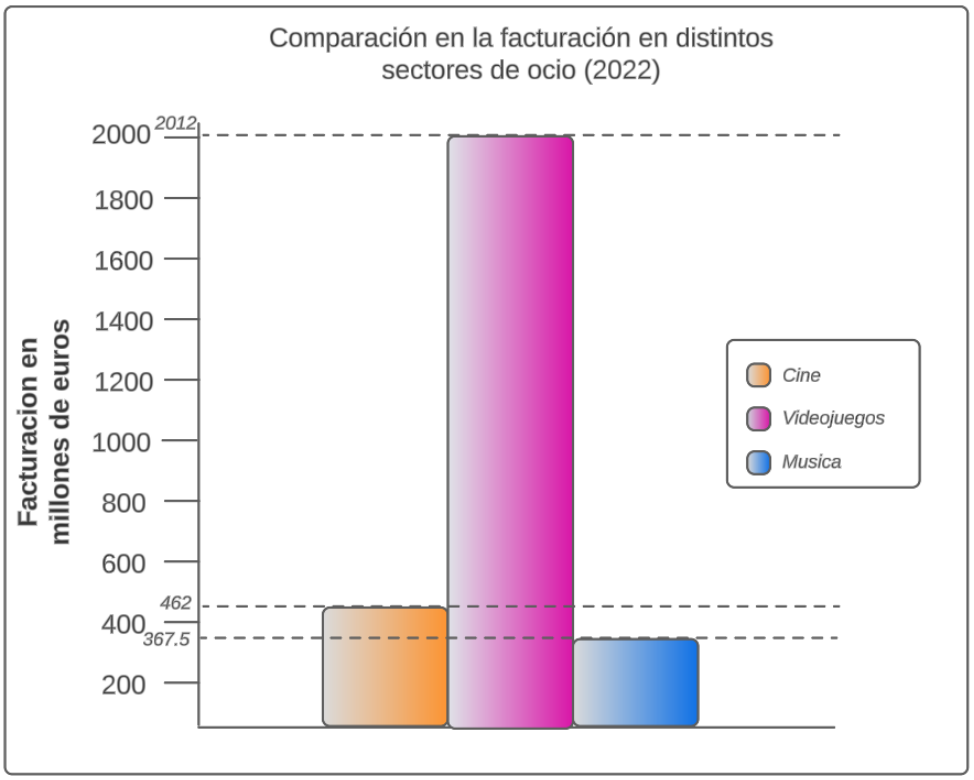


Figura 1.1: Grafica de comparación de la facturación en distintos sectores de ocio, datos extraídos de [1]

Esta diferencia como se puede observar en la gráfica 1.1, puede parecer muy descabellada, pero dentro de este valor se encuentran una gran variedad de videojuegos, como los videojuegos para *smartphone* que se enfocan en un público más casual como el juego `Candy Crush Saga`, en la que los usuarios desembolsan pequeñas cantidades de dinero en objetos del juego, pero como su base de usuarios muy grande acaban siendo la parte más relevante de las ganancias de estos.

Además de estos, existen muchos otros tipos de videojuegos, como deportes, simulación, aventura... El juego que se va a desarrollar en este trabajo es de tipo *Role-playing-game*(ROL), en concreto de uno de sus subgéneros llamado *Roguelike*

En un videojuego de ROL encarnas a un personaje imaginario que vive una aventura en un mundo distinto, que puede ser un mundo realista, de fantasía etc.

Figura 1.2: Imagen de Pokémon Mundo Misterioso Exploradores del Cielo

Del ROL deriva el género *Roguelike*, Este consiste en una aventura a través de laberintos generados de forma aleatoria, en los que el personaje se irá fortaleciendo a medida que avance a través de los objetos y experiencia que obtendrá al derrotar enemigos en la mazmorra.

Consecuentemente, cuanto más avance el protagonista en el laberinto, más desafiantes se volverán estos.

Algunos de los juegos más conocidos del género *Roguelike* son *The binding of Isaac* o *Dead cells*. De todos los de este género, el que me ha inspirado a realizar este trabajo es la saga *Pokémon Mundo Misterioso* 1.2.

1.1 Objetivo

El objetivo del proyecto se centra en desarrollar el apartado desafiante de un juego *Roguelike* en el que el personaje tiene que derrotar a todos los enemigos de la sala para ir avanzando de pantalla. La intención no es acabar con un juego comercial, si no centrarse más en el desarrollo de los algoritmos necesarios en un juego de este estilo.

Por este motivo, no se centrará en cosas relacionadas con la usabilidad y la experiencia de usuario como lo son el audio o la realización de un tutorial, ni se generará un ejecutable final.

Para poder cumplir el objetivo principal será necesario que el juego cuente con los siguientes características:

- Una serie de enemigos con distintas características que tomen decisiones en función de sus atributos, por ejemplo un enemigo más resistente irá al frente mientras que uno menos resistente y con más alcance se encontrará en la retaguardia.
- Generación de mapas distintos formado por habitaciones y pasillos en el que se moverán los enemigos y el jugador, este es un apartado importante ya que habrá tipos de estrategia que sean mejores en unos tipos de mapas que otros.
- Una puntuación que determine como de bien lo ha hecho el jugador frente a los enemigos y como han actuado los enemigos, ya que si la inteligencia de los enemigos es elevada lo lógico es que se vea reflejado en la puntuación del jugador.

1.2 Análisis de requisitos

En esta sección se recogerán formalmente los aspectos fundamentales que se espera que posea el proyecto en su finalización.

1.2.1 Requisitos funcionales

En este apartado se recogen los requisitos que definen cómo debe comportarse el sistema frente a los eventos y las características que estos deben poseer.

Nombre	Descripción
RF01	El jugador podrá mover a su personaje empleando las teclas W,A,S,D del teclado.
RF02	El jugador podrá saber su vida actual y la de los enemigos en cualquier momento.
RF03	El jugador podrá observar el mapa completo durante su turno.
RF04	El jugador tendrá un mecanismo para acercarse a los enemigos más rápido de lo que pueden huir.
RF05	El juego deberá poder generar mapas de diversos tamaños.
RF06	Los mapas generados estarán formados por habitaciones y pasillos.
RF07	Los enemigos podrán configurarse con al menos 3 tipos de inteligencia distintos.
RF08	Deberá haber distintos tipos de enemigos, existiendo al menos uno con la capacidad de atacar al jugador desde una posición desde la que no pueda realizar un contraataque sin moverse.
RF09	Deberá existir una funcionalidad para automatizar el control del jugador.
RF10	Deberá existir un modo desarrollador que facilite la depuración del juego.
RF11	El modo desarrollador permitirá en eliminar y añadir nuevos enemigos.
RF12	El modo desarrollador permitirá desplazar a los enemigos por el mapa.
RF13	El modo desarrollador podrá modificar el mapa, permitiendo introducir nuevas estancias y bloquear las ya existentes.

1.2.2 Requisitos no funcionales

En este apartado se recogen los requisitos no funcionales que son los que establecen restricciones globales sobre las características que ofrece el sistema.

Nombre	Tipo	Descripción
RNF01	Portabilidad	La aplicación solo se podrá ejecutar en un ordenador de sobremesa
RNF02	Compatibilidad	La aplicación deberá funcionar en Windows 11
RNF03	Rendimiento	Los enemigos deben tomar una decisión en menos de 2 segundos
RNF04	Rendimiento	El juego deberá funcionar a más de 30 fotogramas por segundo
RNF05	Rendimiento	El juego debe ser capaz de controlar hasta 50 enemigos, sin degradar la experiencia
RNF06	Usabilidad	El juego debe ofrecer un reto significativo al jugador

1.3 Herramientas utilizadas

Desde que surgió la informática se han ido desarrollando y perfeccionando herramientas software que facilitan en gran medida el desarrollo del software actual. En esta sección, se hará una descripción de las herramientas que se han empleado tanto en el desarrollo del juego.

1.4 Unity

Unity es un motor de juegos, que permite el desarrollo de aplicaciones de simulación o videojuegos, controlando el comportamiento de estos a partir de *scripts* escritos en el lenguaje de programación C#.

1.4.1 Conceptos básicos

En este apartado se verán los elementos fundamentales que ofrece la API de Unity para el desarrollo de videojuegos.

GameObject

El elemento principal de la API de Unity son los `GameObject` que representan los elementos que se encuentran en las escenas, como los personajes, elementos del fondo o la cámara. Para dotarlos de funcionalidad, se les asignan componentes, que son los que permiten distinguir a un personaje jugable frente a una luz del entorno.

Algunos de estos componentes son:

- **Transform**: Controla la posición y la orientación en la que se encuentra el *GameObject* en la escena.
- **Script**: Es el componente más flexible, ya que permite al desarrollador controlar el comportamiento del *GameObject*, mediante código en C#. Este control lo realiza mediante los eventos de control de flujo de juego.

Los *GameObject*, pueden tener `GameObject` **hijos** asignados formando una jerarquía. Acciones como instanciar o deshabilitar implicará que está acción se desarrolle en cadena con los hijos. Por ejemplo, si deshabilitas al `GameObject` padre, se deshabilitarán todos los `GameObject` que se encuentren por debajo en la jerarquía.

Además, dentro de Unity, existen los denominados *assets*, que son cualquier archivo que puede ser empleado en un proyecto de Unity, ya sea procedente de este como pueden ser los *prefabs* o que sean externos como un archivo de música .mp3.

Por último, uno de los atributos más importantes de un GameObject es `enabled`, que permite habilitar y deshabilitar el comportamiento de estos.

Prefabs

Aunque en el apartado anterior se hayan visto solo dos componentes, en Unity existen muchos más. Asimismo, es frecuente que sea necesario instanciar el mismo `GameObject` múltiples veces. Para facilitar esto existen los *Prefab*

Un *Prefab*, es un *asset* que contiene la información de un *GameObject* sirviendo de base para instanciar muchas copias del mismo [4].

Eventos de control del flujo del juego

En Unity existe una gran cantidad de eventos que se ejecutan en un orden fijo, estos permiten indicar dentro de un *script* cuando se quiere que se produzca el comportamiento deseado. En esta sección únicamente se hablará de los que se utilizarán en este trabajo, que en función de su orden de ejecución son:

1. `Awake`: Son los primeros eventos que se lanzan cuando se instancian al iniciar el juego o al instanciar un *prefab* con dicho *script*. Todos los eventos *Awake* de todos los *scripts* se lanzarán antes de que se produzcan los siguientes eventos.
2. `Start`: Se lanzan antes de que se renderice el primer fotograma de juego, se emplea para inicializar las variables y la lógica del *GameObject*.
3. `Update`: Se lanza una vez por fotograma y permite controlar el comportamiento del objeto durante el desarrollo del juego, debido a su frecuencia de ejecución conviene que la complejidad temporal y espacial del código en esta sección sea la menor posible.

Tilemap

La clase `tilemap` de la API de Unity [5], contiene funcionalidades para renderizar y gestionar un mapa de baldosas. Las baldosas que forman un mapa de baldosas, pueden ser de distintas formas geométricas como cuadrados o hexágonos. Esta forma las determina el `Grid`, que es un `GameObject` que se coloca como padre de los mapas de baldosas.

1.4.2 Lucidchart

Lucidchart [6] es una herramienta web de pago que permite el diseño de diagramas conceptuales. Su funcionamiento se basa en un tablero con cuadriculas, sobre el que permite introducir figuras y texto, además de relaciones entre ellos mediante flechas.

Aunque sea de pago ofrece un uso gratuito con limitaciones en el número de figuras que puedes usar en cada tablero, siendo este límite uno bastante elevado, por lo que con la licencia de uso gratuito es suficiente.

Se ha empleado esta herramienta tanto para visualizar mejor los algoritmos y realizar algunas de las distintas figuras que se encuentran en este trabajo.

1.4.3 Visual studio

Visual studio es un "Integrated Development Environment" (IDE) lanzado por Microsoft en 1997. Un IDE es una herramienta de software que facilita y agiliza el desarrollo de aplicaciones. Visual Studio cuenta con componentes de depuración, analizador y autocompletado de código.

Este editor tiene soporte con Unity, de forma que al asociarlo con Unity permite depurar el software que se esta desarrollando durante la ejecución del bucle del juego (*Gameloop*).

1.5 Estructura del trabajo

La estructura que se seguirá en este trabajo a partir de este punto se basarán en capítulos, que contendrán la siguiente información:

Capitulo 2: Conocimientos previos : En este capítulo se abordarán una serie de algoritmos tales como algoritmos de búsqueda de rutas y de toma de decisiones que serán imprescindibles para el buen funcionamiento del juego.

Capitulo 3: Desarrollo del juego : En este capítulo se abordarán aspectos relativos a la funcionalidad del juego, desde sus mecánicas principales, los tipos de enemigos que hay, la generación del mapa, etc.

Capitulo 4: Modo desarrollador : En este capítulo se expondrá una serie de herramientas que permiten al desarrollador del juego, modificar el estado actual del juego para poder depurar el juego con mayor eficiencia.

Capitulo 5: Control de los enemigos : En este capítulo se describirá que algoritmos se han implementado para la toma de decisiones de los enemigos.

Capitulo 6: Pruebas : En este capítulo, se realizarán pruebas sobre los algoritmos explicados en el capitulo 6, y se analizarán los resultados.

Capitulo 7: Conclusiones : En este capítulo se realiza una conclusión del trabajo y se presentan posibles opciones para ampliar el videojuego.

Capítulo 2

Conocimientos previos

En este apartado, se van a desarrollar los algoritmos empleados en el desarrollo del juego, como los algoritmos empleados para controlar a las entidades del juego y algunos aspectos fundamentales de Unity.

2.1 Algoritmos de búsqueda de rutas

El objetivo de los algoritmos de búsqueda de rutas es encontrar el camino de un punto a otro, evitando los obstáculos que se encuentren por el camino. En el caso del desarrollo de este juego, es importante saber que camino debe seguir una entidad para poder llegar de un punto a cualquier otro punto del laberinto.

2.1.1 Búsqueda en profundidad

El algoritmo de **búsqueda en profundidad**, comúnmente conocido como **DFS**(Depth-first search), consiste en ir explorando el árbol de posibilidades priorizando siempre los nodos más profundos del grafo, es decir los que se encuentran más lejos del nodo inicial.

Un ejemplo de este, se puede observar en la fig.2.1, en este se puede observar el grafo inicial identificado con (a), donde cada nodo se le nombra con una letra del abecedario. En caso de que dos nodos se encuentren en la misma profundidad, el desempate se realizará por orden alfabético. A continuación muestro los pasos de la ejecución parcial de este algoritmo, mostrando únicamente la exploración de la componente conexa de la izquierda:

1. Partiendo desde A, se pueden explorar los nodos B y E. Como ambos nodos se encuentran a la misma distancia del origen, se explorará B ya que está por delante de E en el abecedario.

2. Como B no tiene más vecinos, se explorará E que está conectado con I y J.

3. siguiendo el mismo razonamiento que en el primer paso, como I y J se encuentran a la misma profundidad primero se explorará I.

4. Como en I no hay nodos conectados que sean más profundos, se explorará J. Completando el árbol.

Observando el árbol generado en el apartado (b), se puede ver que para explorar J se ha acabado realizando el camino más profundo posible desde A, por esto se le llama búsqueda en profundidad.

2.1.2 Búsqueda en anchura

El algoritmo de **búsqueda en anchura**, conocido en ingles como **BFS**(Breath-first search), consiste en ir explorando el árbol de posibilidades, priorizando aquellos nodos que se encuentren a un nivel de profundidad menor.

Este algoritmo es capaz de encontrar el camino que menos pasos requiere para poder llegar al destino.

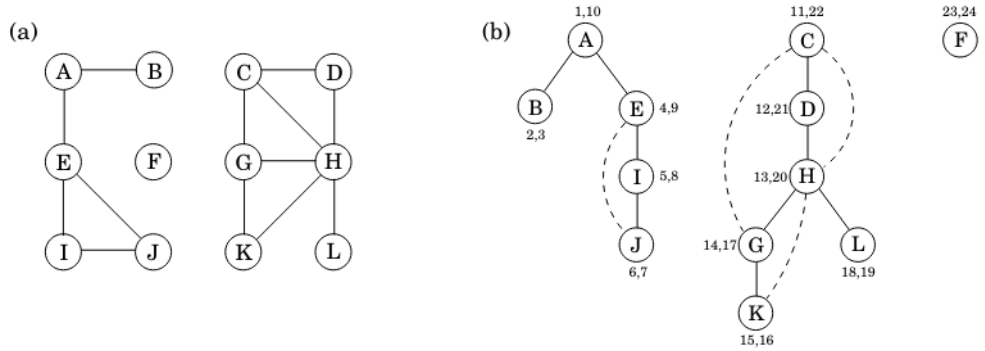


Figura 2.1: Ejemplo de búsqueda en profundidad extraído del libro *Algorithms* [7] (Las líneas discontinuas representan las aristas no visitadas)

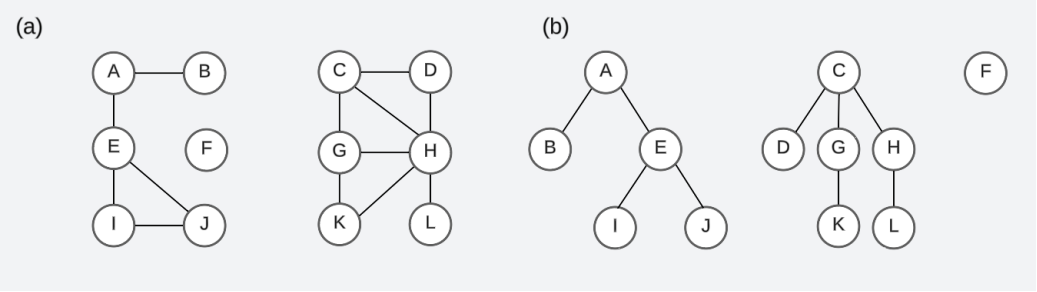


Figura 2.2: Aplicación del algoritmo búsqueda en anchura del ejemplo 2.1

Para mostrar la diferencia con el algoritmo de búsqueda en profundidad, se va a mostrar la ejecución de este algoritmo con el mismo ejemplo empleado en el apartado 2.1.1.

1. Partiendo desde A, se pueden explorar los nodos B y E. Como ambos nodos se encuentran a la misma distancia del origen, se explorará B ya que está por delante de E en el abecedario.
2. Como B no tiene más vecinos, se explorará E que está conectado con I y J.
3. Desde el nodo E se explorará I
4. Como desde el nodo E hay un camino más corto al nodo J que desde el I, se explorará J desde E.

En la fig.2.2, se observa en el apartado (b) el árbol generado, que en este caso es más plano que en el de la fig.2.1.

2.1.3 A*

A diferencia de los dos algoritmos anteriores, en el algoritmo A^* el orden de exploración se decide en base a la función f, que se calcula como la suma entre el coste empleado hasta llegar al punto actual g, más el valor de un heurístico h que sirve como estimador para indicar lo cerca que se cree que está del punto final [8].

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

En el ejemplo 2.3, partiendo de la situación en la que se ha explorado A y B, en la cola de descubiertos las siguiente tabla 2.1. En esta se observa que de los nodos que se encuentran por explorar el que tiene el menor coste es E, por lo tanto será el siguiente nodo en ser explorado.

El algoritmo A^* es capaz de encontrar el camino más corto si el heurístico es **admisible**. Un heurístico es admisible si en ninguna circunstancia sobreestima el coste para alcanzar

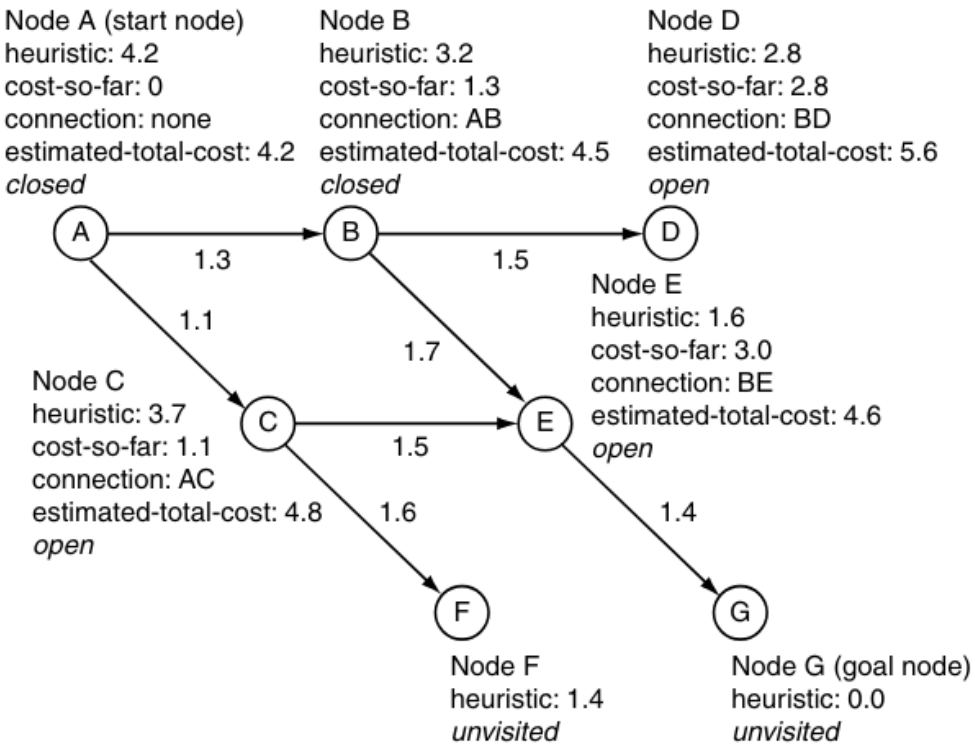


Figura 2.3: Ejemplo A^* extraído del libro [8]

Nodo	Coste hasta el punto	Heurístico	Coste	Orden
D	2.8	2.8	5.6	3
E	1.6	3	4.6	1
C	3.7	1.1	4.8	2

Table 2.1: Tabla con la cola de la situación actual del ejemplo 2.3

el objetivo, es decir que el heurístico siempre es menor o igual que el coste real para llegar al punto final.

2.2 Algoritmos basados en Minimax

Uno de los algoritmos empleados en juegos de tableros por turnos es el llamado `minimax`, ideado por John Von Neummann y publicado en la revista Alemana *Mathematische Annalen* [9]

Este se basa en la construcción de un árbol n-ario de profundidad m, donde n es el número de acciones que puede tomar cada jugador y m el número de jugadas. En cada nivel de forma alternativa actuarán dos agentes, *max* y *min*, el primero representa la mejor elección que puede realizar el jugador y el segundo representa la mejor elección que puede tomar el enemigo en esa situación.

Los nodos hoja representan las posibles situaciones tras m jugadas, conteniendo en su valor la puntuación del juego, que se calculará a través de una función `score`.

A partir de estas evaluaciones, en los niveles que corresponde a *max* se tratará de elegir las acciones con la mayor puntuación, mientras que en las del jugador *min* se elegirá las de menor puntuación.

El algoritmo se encuentra representado en la fig.1, donde como parámetros de entrada se reciben:

- **nodo**: Contiene el estado del juego desde el cual se desea evaluar la mejor jugada posible.
- **profundidad**: Es un entero que indica la profundidad máxima hasta la cual el algoritmo debe evaluar el árbol de juego.
- **esMax**: Es un booleano que determina si está actúando el agente *max* o el agente *min*.

Observando el ejemplo de la fig.2.5, a la izquierda del todo aparece que agente se encuentra actuando en ese nivel. La elección de los nodos *max* y *min* aparecen representadas con una flecha roja o una flecha azul respectivamente.

Un ejemplo del calculo del valor del nodo *max* de profundidad 3 que se encuentra a la izquierda, es que su valor es $max(8, -47) = 8$, en el caso del nodo *min* de profundiad 2 que se encuentra más a la izquierda es $min(8, 13) = 8$.

Viendo el funcionamiento del algoritmo, se puede apreciar que va a generar una gran carga computacional. ya que el árbol generado tendrá $\sum_{i=0}^{m-1} n^i$, es decir crece de forma exponencial con la profundidad del árbol.

2.2.1 Poda alfa-beta

Dada la complejidad del algoritmo *minimax*, no es de extrañar que se intentarán recurrir a métodos para mitigar el coste computacional de dicho algoritmo. Una de las optimizaciones que se hizo fue, la denominada **poda alfa-beta** [10]. Esta consiste en reducir el número de nodos explorados, conservando dos variables α y β.

El valor alfa es la mejor opción para *max* y beta es la mejor opción para *min*. Pasando estos valores, a los distintos nodos permite conocer a los nodos inferiores cuales han sido las mejores elecciones para *max* y *min* hasta el momento.

El algoritmo se encuentra representado en la fig.2, a los parámetros de entrada explicados en 2.2, se le añaden:

- **alfa**: Valor flotante con la mejor opción que puede tomar `max` hasta el momento.
- **beta**: Valor flotante con la mejor opción que puede tomar `min` hasta el momento.

Por ejemplo, en el caso de profundidad 3 en el segundo nodo empezando por la izquierda, el agente *max* conoce que la mejor elección de *min* es 8. Esto permite que en caso de que *max* encuentre una elección cuyo valor es mayor que 8, no sea necesario seguir explorando por el resto de hijos, ya que *min* no mejorará su elección. En este caso *max* encuentra 13, que es mayor que 8 y por tanto la búsqueda se poda.

Algorithm 1: minimax

```
input : nodo, profundidad, esMax
if nodo es terminal then
    return valor del nodo
end
if esMax es verdadero then
    valor← −inf
    foreach hijo de nodo do
        valor ← max(valor, minimax(hijo, profundidad - 1, Falso))
    end
end
else
    valor← inf
    foreach hijo de nodo do
        valor ← min(valor, minimax(hijo, profundidad - 1, Verdadero))
    end
end
return valor
```

Figura 2.4: Algoritmo ideado por John Von Neuman en [9]

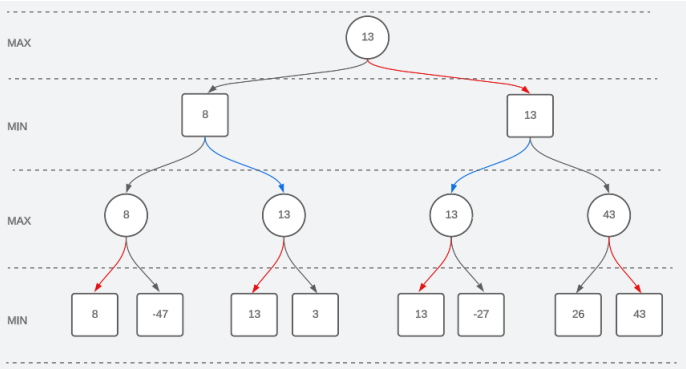


Figura 2.5: Ejemplo de minimax 2-ario para m=4

2.3 Arboles de decisión

Los árboles de decisión son un tipo de algoritmos basados en un árbol, en los que las acciones a tomar se encuentran en las hojas. Para tomar una acción, se partirá desde el nodo raíz y se irá respondiendo a una serie de preguntas, que los llevará hasta una hoja del árbol.

Estas preguntas pueden ser respondidas con un si o no como se puede observar en la 2.8. Mientras que otras preguntas pueden ser respondidas con múltiples opciones, como se puede ver en la fig.2.9, donde el estado de alerta está identificado con un color, y en base a este debe de tomarse una acción u otra.

Esto junto a la estructura de árbol hace que sean sencillos de entender, además de rápido de ejecutar, ya que únicamente será necesario responder a la altura del árbol-1 en el peor de los casos y que la complejidad de la mayoría de preguntas no es elevada.

Otra de las ventajas de los árboles de decisión es que debido a su construcción y estructura, es que es sencillo añadir nuevas decisiones, ya que basta con añadir un nodo con una pregunta más. Además, se pueden extraer subárboles del árbol principal y cambiarlas por otros modificando significativamente la toma de acciones del árbol. [11]

Algorithm 2: minimax con poda alfabeta

```
input : nodo, profundidad, alfa, beta, esMax
if nodo es terminal then
    return valor del nodo
end
if isMax es verdadero then
    foreach hijo de nodo do
        alfa ← max(valor, minimax(hijo, profundidad - 1, alfa, beta, Falso)) if
         beta > alfa then
            return alfa
        end
    end
end
else
    valor← inf
    foreach hijo de nodo do
        beta ← min(valor, minimax(hijo, profundidad - 1, Verdadero)) if
         alfa > beta then
            return beta
        end
    end
end
return valor
```

Figura 2.6: Algoritmo inventado por T.P.Hart y D.J Edwards en [10]

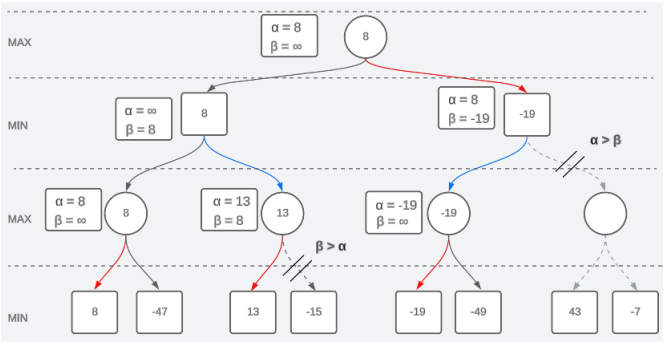


Figura 2.7: Ejemplo de poda alfa-beta para n=4

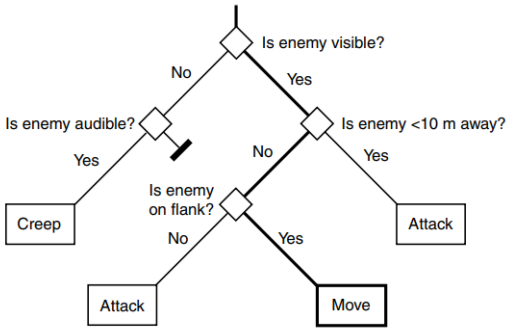


Figura 2.8: Arbol de decision extraido de [11]

2.4 Arboles de comportamiento

Los árboles de comportamiento permiten a los personajes, reaccionar en base a unas decisiones que han tomado los enemigos a esto se le llama **planificación reactiva**. [12]

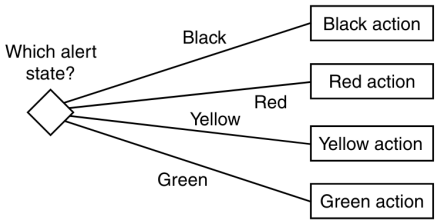


Figura 2.9: Ejemplo de pregunta con respuesta múltiple extraído de [11]

La unidad esencial de un árbol de comportamiento es la tarea, estas se componen en sub-arboles para formar comportamientos complejos. Las tareas pueden ser de tres tipos:

- Condiciones: Comprueban una propiedad del juego.
- Acciones: Alteran el estado del juego.
- compuestas: Su comportamiento se basa en la funcionalidad de los hijos que estos poseen.

Las condiciones y acciones están en las hojas del árbol y la mayoría de las ramas están unidas por tareas compuestas.

Las tareas compuestas diferencian al árbol de decisión 2.3, del árbol de comportamiento, ya que permiten recorrer el árbol de una forma distinta. Esto se explicará en el apartado 2.4.1.

Todas las tareas comparten es una interfaz común `task`, permitiendo combinarlas sin necesidad de conocer que hay en el resto del árbol de comportamiento y permitiendo eliminar, introducir y mover ramas que pueden modelar un comportamiento complejo.

2.4.1 Tareas compuestas

Las tareas compuestas, son una herramienta potente de los árboles de comportamiento, que permiten recorrer el árbol de formas distintas. Esto se logra, haciendo que su comportamiento sea dependiente del resultado de ejecución de sus nodos hijos.

Tareas compuestas deterministas

Las tareas compuestas deterministas, son aquellos en el que el orden de ejecución de los hijos es fijo y no cambia. Pueden ser de los siguientes tipos:

Seleccionador : Devolverá un código de ejecución correcta cuando uno de sus hijos se ejecute de forma satisfactoria. Es decir, dejará de recorrer sus hijos cuando uno se ejecute correctamente.

Secuencia : Devolverá un código de ejecución correcta cuando todos sus hijos se ejecuten de forma satisfactoria. Es decir, ejecutará todos sus hijos hasta que uno de ellos no se ejecute correctamente.

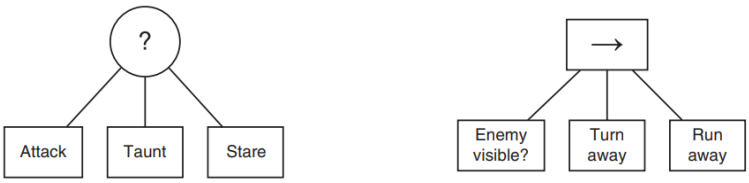
La representación gráfica de estos nodos se puede observar en la fig.2.10.

Nodos composites no deterministas

Basándonos únicamente en nodos **seleccionador** y **secuencia**, las IA creadas con estos nodos poseen un comportamiento bastante **predecible**, ya que los personajes siempre repiten y comprueban las acciones y condiciones en el mismo orden.

El cambio del orden de ejecución puede implementarse de varias formas, una de ellas es barajar los hijos antes de recorrerlos, por ejemplo en el código 2.11. Otra forma de hacerlo es haciendo que sus hijos se ejecuten de forma aleatoria permitiendo incluso repeticiones como se observa en 2.12.

Estas formas de recorrer el árbol da lugar a una IA más impredecible y por tanto más interesante para el jugador. Su representación se puede observar en la fig.2.13.



(a) Representación gráfica de un nodo `Seleccionador`

(b) Representación de un nodo `sequencia`

Figura 2.10: Tareas compuestas deterministas extraídos de [12]

```
class NonDeterministicSelector (Task):

    children

    def run():
        shuffled = random.shuffle(children)
        for child in shuffled:
            if child.run(): break
        return result
```

Figura 2.11: Ejecuta todos los hijos una vez en orden aleatorio, extraído de [12]

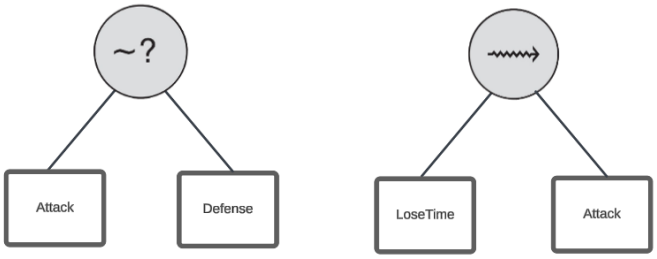
2.5 Algoritmo de barajado Durstenfeld

Durante el desarrollo se ha considerado conveniente tener un algoritmo de barajado. Basado en la recomendación del libro, se ha optado por utilizar el algoritmo de *Durstenfeld* [13].

El funcionamiento del algoritmo se puede ver en la fig. 2.15. Este consiste en seleccionar un índice aleatorio de entre las posiciones no bloqueadas que son las blancas. Los valores del índice y la última baldosa no bloqueada serán intercambiados. Tras esto, esta última se bloqueará. Este proceso se repetirá hasta que solo quede una baldosa no bloqueada.

```
class RandomSelector (Task):
    children
    def run():
        while True:
            child = random.choice(children)
            result = child.run()
            if result:
                return True
```

Figura 2.12: Ejecuta sus hijo de forma reiterada y aleatoria, hasta que uno de ellos se ejecute correctamente, extraído de [12]



(a) Representación gráfica de un nodo `Seleccionador no determinista`

(b) Representación de una tarea secuencia no determinista

Figura 2.13: Simbología de tareas compuestas no deterministas extraídos de [12]

Algorithm 3: Barajado de Durstenfeld

input : lista
for $i \leftarrow$ *lista.count - 1* **to** 1 **decrement do**
  $j \leftarrow obtieneRandom(0, i)$
  $aux \leftarrow lista[i]$
  $lista[i] \leftarrow lista[j]$
  $lista[j] \leftarrow aux$
end
return valor

Figura 2.14: Código extraído de la fuente [13] (adaptado para que el primer índice sea el 0)

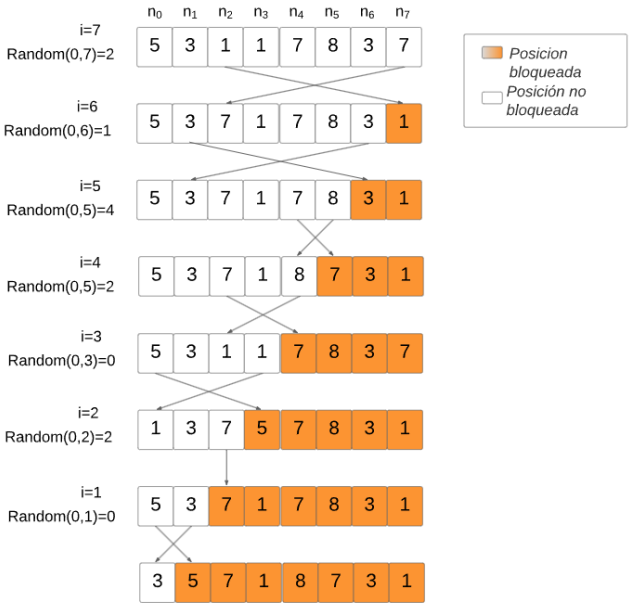


Figura 2.15: Ejemplo algoritmo de barajado Durstenfeld, basado en [14]

Capítulo 3

Desarrollo del juego

En este apartado se tratarán los detalles acerca de la implementación del juego, empezando por el proyecto sobre el que se han extraído algunas ideas. Posteriormente, se pasará a explicar la generación del mapa y finalmente la instanciación de las entidades y la gestión de los turnos de estas.

3.1 Proyecto de Unity roguelike

Para el desarrollo del juego se parte del proyecto **RogueLike 2D** de la plataforma *Unity Learn* [15], del que se ha extraído los *sprites* del juego, que son *assets* que almacenan los diseños de los elementos del juego, además de algunas ideas del código sobre como realizar la separación por turnos.

Figura 3.1: Roguelike 2D unity project

3.2 Comprensión general del funcionamiento del juego

El juego es inicializado por el *script* `GameManager`, que manda generar la información de la mazmorra completa, es decir al mapa y la posición de las entidades al `GameStateGenerator`. La información la devolverá como un objeto de la clase `GameState` que representa el **estado de juego**.

Una vez generado el estado del juego, le mandará la información sobre el mapa de la mazmorra al script `MapUIController` que se encargará de renderizarlo. Cuando el mapa se haya renderizado, sobre este se instanciarán todos los *prefabs* de los enemigos, esto generará tanto su modelo 2d, como el control de estos sobre la interfaz visual del videojuego.

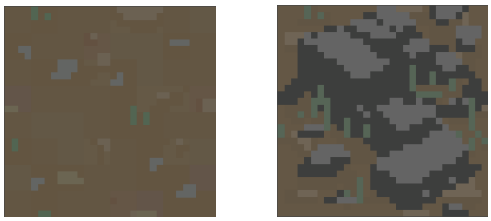
Finalmente, una vez cargado el mapa, el `GameManager` dará la información de los controladores al `battleManager`, que se encargará de la gestión de las batallas por turnos y del control de la inteligencia de los enemigos.

3.3 Generación del mapa

La generación de mapa del juego en un juego de tipo *roguelike* es importante, dado que el funcionamiento de estos es completar mazmorras mientras obtienes objetos que hacen más poderoso al personaje.

A pesar de que en este juego no se haya implementado la posibilidad de que el jugador se vuelva más poderoso, los enemigos y el jugador se moverán por el mapa, el cual será importante para tomar decisiones estratégicas. Por ejemplo, en un pasillo es más difícil rodear al jugador que en una habitación.

El mapa está compuesta por dos tipos de baldosas principales, las baldosas mazmorra que forman las habitaciones y los pasillos donde se pueden encontrar las entidades y los muros que no pueden ser atravesados por estas. Los *sprites* empleado para estas baldosas se pueden ver en la fig.3.2.



(a) Baldosa mazmorra (b) Muro

Figura 3.2: Tiles del juego

3.3.1 Generación lógica del mapa

Antes de poder generar el apartado visual del mapa, es necesario generar la información lógica del mapa. De esta generación se encarga la clase `MapGenerator`, capaz de generar un objeto de la clase `map` que contendrá las habitaciones y los pasillos de las mazmorras, además de un limite, que impedirá agregar baldosas fuera de este límite.

Las habitaciones están formadas por baldosas. A un subconjunto de estas baldosas se les denomina **baldosas puerta**. Estas sirven como puntos de conexión, siendo utilizadas como extremos en la generación de los pasillos entre habitaciones.

Las habitaciones se construyen sobre una estructura en *grid* cuadricular, con unas dimensiones indicadas por las variables `squareHeightMap` y `squareWidthMap` que indican el número de cuadrados de alto y de ancho del *grid* respectivamente, que dará lugar al máximo número de habitaciones que se puedan crear dentro del mapa.

A los cuadrados del grid se les llamará *slot* y poseerán un tamaño indicado en la variable `squareSize`. Cada slot se encuentra identificado por un número de 0 a (squareHeightMap$\cdot$ squareWidthMap) $-$ 1. Un ejemplo del uso de estos valores se observa en la fig.3.3,

Para la construcción del mapa, se seleccionan de forma aleatoria en que *slots* del mapa se va a construir una habitación. Estas se construyen sobre el origen de coordenadas y después son trasladadas al *slot* correspondiente.

Una vez generadas las habitaciones, se iniciará la construcción de pasillos siguiendo los siguientes pasos:

La generación de la información de pasillos se produce en dos pasos:

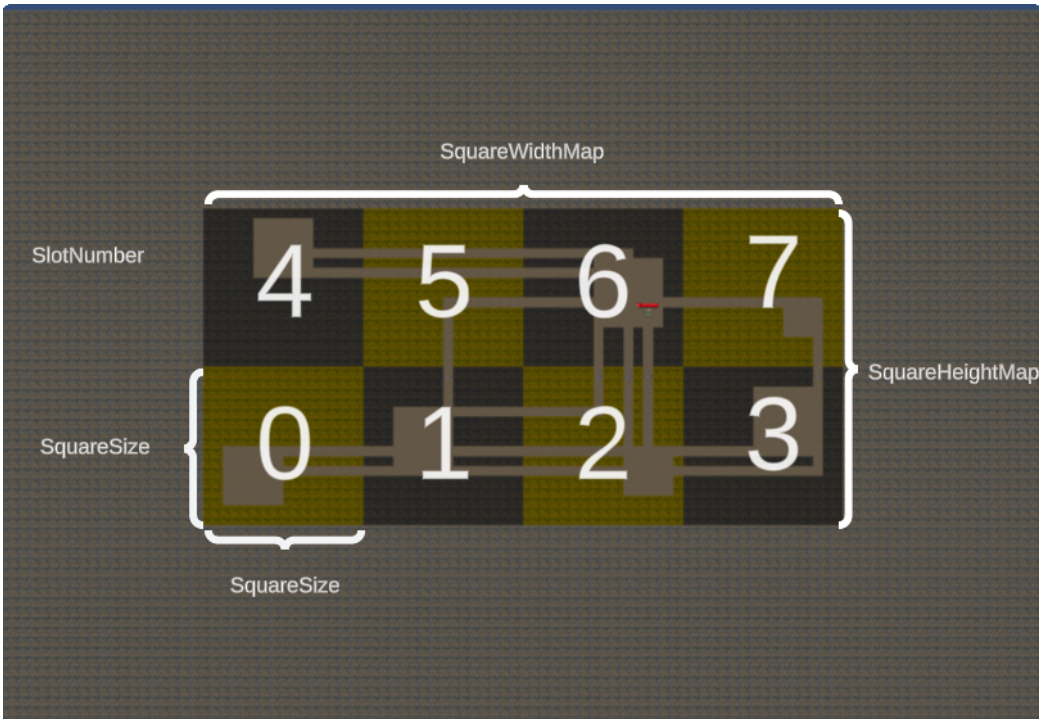


Figura 3.3: Mapa generado `SquareSize`=16, *SquareHeightMap* = 2, `SquareWidthMap` = 4

1. Se barajará la lista de habitaciones, uniendo las habitaciones que tengan el índice i con el índice i+1 en el array.

2. Posteriormente, se generarán pasillos en función del número de habitaciones que se unirán con la habitación del índice 0.

Un ejemplo de este proceso se puede observar en la fig. 3.4, en este caso se observa que se ha formado el pasillo que une 0 con 0, en ese caso se ignora el pasillo y solo se construyen 3.

Una vez generada la información, se generarán los pasillos en forma de L, pudiendo suceder que un pasillo generado pase por encima de una habitación, haciendo que el pasillo generado una también a la habitación con la que se ha solapado.

3.3.2 Generación de mapa visual

Una vez generado el objeto `Map`, el script `MapUIController` se encarga de transformar esta información a la API de Unity para poder renderizar los mapas. Esta renderización se ha logrado en base a dos objetos de clase `Tilemap`, `tilemap` y `tilemapWall`.

En el primero de ellos se recogerán todas las baldosas que forman parte de las habitaciones y los pasillos, en el segundo los muros que no podrán ser atravesados por las entidades.

El orden de construcción del mapa es el siguiente:

1. Se llena el mapa de muros rellenando toda la superficie que indicados por los límites del mapa, más un poco más para evitar que la cámara del jugador pueda ver más allá de los muros, donde no se encuentra renderizada ninguna baldosa.(Fig.3.5)

2. Se dibujan las baldosas que corresponden a los pasillos y habitaciones de la mazmorra.

Figura 3.4: Ejemplo de la generación de pasillos para 8 habitaciones

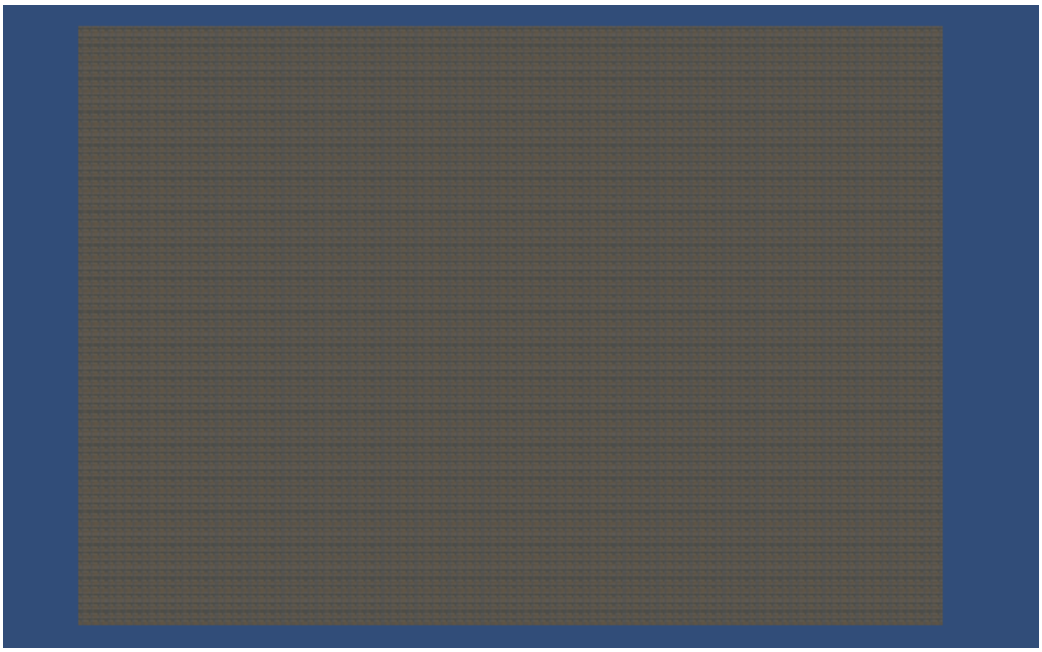


Figura 3.5: Primer paso de la creación del mapa

3.4 GameManager

Además de la funciones mencionadas en el apartado 3.2, el *GameManager* lleva la información acerca de la puntuación del jugador y de los niveles que se van avanzando, haciendo que se activen y desactiven los elementos de la interfaz en función de la situación, como por ejemplo mostrar un mensaje de que el jugador ha muerto o se ha pasado el nivel.

Para hacer esto se basa en una serie de las siguientes funciones:

- `StopGame`: Ordena al battleManager que pare la batalla
- `loseGame`: Muestra el mensaje de muerte, tras el jugador perder todos los puntos de vida.
- `increaseScore`: Añade un punto a la puntuación y lo actualiza en la interfaz.
- `floorCompleted`: Muestra el mensaje de piso completado y te permite decidir si quieres continuar explorando o no.

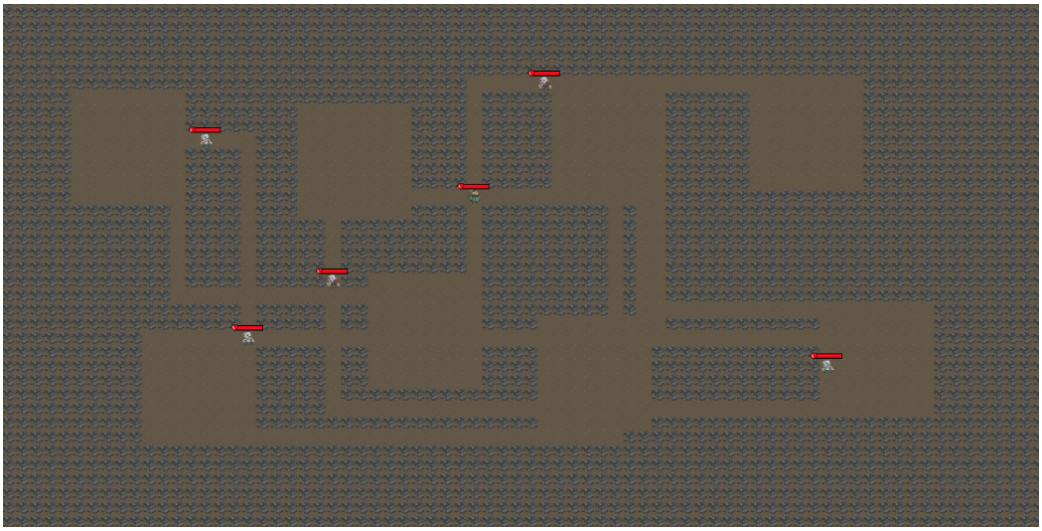


Figura 3.6: Mapa generado por el algoritmo propuesto, fijando el tamaño de las habitaciones

- `resetGame`: Genera un nuevo mapa, sin reiniciar la puntuación.

Por último el *GameManager* también se encarga de la gestión de la funcionalidad de las herramientas de depuración, siendo llamada por las herramientas de depuración para modificar el estado del juego. El funcionamiento de estas herramientas, se explicará en el capitulo 4.

3.5 Entidades

Las entidades son todos aquellos personajes que sean capaz de moverse y cuente con una serie de estadísticas que determinan cuanta vida tiene o cuanto daño hacen al atacar.

Estos datos y los comportamientos básicos de todas las entidades se encuentran en el *script* `movableEntity`.

Las estadísticas que poseen las entidades son las siguientes:

- `id`: Identificador único de cada entidad.
- Puntos de salud: Puntos de vida restantes de la entidad, una entidad está viva cuando tiene al menos un punto de salud.
- Daño de ataque: Puntos de daño que recibe otra entidad cuando es atacada por esta.
- Defensa: Puntos que reducen el daño recibido por otras entidades.
- Rango de ataque: Distancia en baldosas desde la que puede atacar una entidad a un enemigo.
- Posición: La posición de la baldosa en la que se encuentra la entidad en el mapa.
- Tipo: El tipo de entidad que es, puede ser una de las que se encuentra en la fig.3.8.

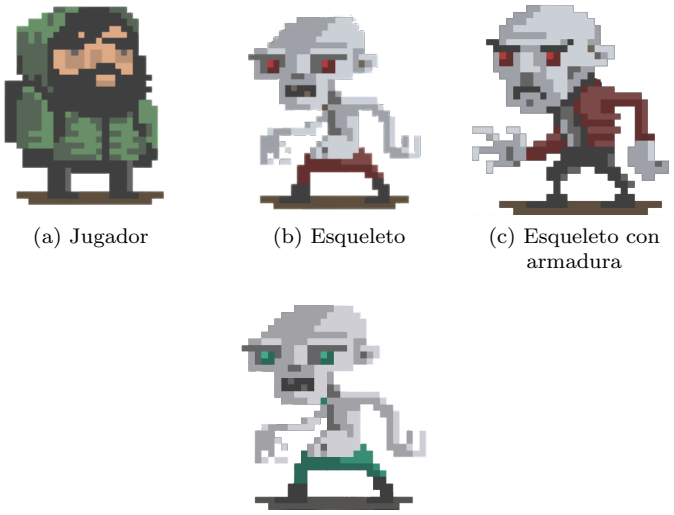
Todas las entidades de un mismo tipo, poseen un Controlador del mismo tipo y las mismas estadísticas iniciales, que se encuentran definidos en su propio **prefab**. Estas estadísticas permanecerán inmutables a excepción de los puntos de salud, que se verán **reducidos** cada vez que se recibe un **ataque**, y su **posición** que variará cuando se mueva la entidad por el mapa.

Las entidades pueden realizar dos acciones **atacar** y **moverse**.

Figura 3.7: Barra de vida del jugador con distinta cantidad de puntos de vida

Los **ataques** se podrán realizar a entidades cuya distancia se encuentre a una distancia menor al rango de ataque del agresor. Tras un golpe, si los puntos de salud de una víctima de un ataque son 0 o menor que este valor, la entidad será eliminada del juego.

El **movimiento** se puede hacer en dos direcciones vertical y horizontal, es decir una entidad se puede mover hacia arriba, abajo, izquierda o derecha.



(a) Jugador (b) Esqueleto (c) Esqueleto con armadura

(d) Esqueleto a distancia

Figura 3.8: Tipos de entidades del juego

3.5.1 Instanciacion de entidades

Es necesario instanciar tanto los enemigos como al jugador que se encuentran en la mazmorra, este puede darse en dos situaciones:

1. Colocar los enemigos de forma **aleatoria** al iniciar los niveles.
2. Instanciar a los enemigos en una posición concreta con la función *generar enemigos* del depurador.

Para el primero de los casos será suficiente con seleccionar de forma aleatoria, un número de baldosas en la que instanciar los enemigos. Para realizar esto se ha modificado el algoritmo de Durstenfeld 2.5, deteniéndolo cuando las posiciones bloqueadas coinciden con el número de baldosas que se quiere extraer, estas posiciones bloqueadas serán las posiciones en las que se colocarán los enemigos.

Una vez seleccionada las baldosas, se seleccionarán los tipos de enemigos 3.8 de forma aleatoria, se les asignará las posiciones obtenidas en el paso anterior y se instanciarán en base a la `prefab` del tipo de enemigo.

En el caso de instanciar los enemigos en una posición concreta, a diferencia del primer caso será necesario comprobar que no hay una entidad en la posición, ya que no puede haber dos enemigos en la misma posición.

3.6 BattleManager

3.6.1 Introducción

El encargado de gestionar las batallas entre el jugador y los enemigos es el *script* `BattleManager`. Las batallas se basan en turnos, que son gestionadas a través de una variable booleana que indica si en el turno actual le corresponde actuar al jugador. También se encargará de crear los objetos necesarios, para controlar a los enemigos de forma automática y al jugador si se encuentra en modo *autoplay*.

3.6.2 Gestión de turnos

El primer turno, siempre corresponderá al jugador, una vez ejecute su acción, se cambiará a falso el valor de la variable booleana y se iniciará el segundo turno en el que se moverán los enemigos.

Para evitar que los enemigos escapen y el jugador no les pueda alcanzar, se ha introducido un **turno de dash**. Este turno se producirá cada vez que el jugador actúa 3 veces, permitiéndole realizar una segunda acción de forma consecutiva.

Los enemigos, actuarán en orden en función de su `id`, actuando primero los que tienen un id menor, su turno acabará cuando hayan actuado todos los enemigos.

3.6.3 Control automático de los enemigos y del jugador

Al `battleManager` le corresponde el control de los enemigos, decidiendo que acción toma cada uno. Este control se realiza con un objeto de la clase `Brain` o de una de sus subclases, que se creará al inicio de cada nivel.

Cada una de las subclases de `Brain`, implementa un algoritmo distinto para el control de enemigos, existiendo los siguientes, que implementarán los algoritmos que se explicarán en la sección 5 *Control de los enemigos*:

- EagerEnemiesBrain: Algoritmo Voraz
- EagerastarEnemiesBrain: Algoritmo Voraz A^*
- AlphaBetaPruneBrain: Algoritmo poda alfa-beta
- BehaviorTreeBrain: Algoritmo árbol de comportamiento
- DecisionTreeBrain: Algoritmo árbol de decisión

A parte de esto será necesario que en cada turno se actualice la información que posee `brain`, para que pueda realizar acciones con la información más reciente.

battleManager se encargará de mandar al objeto de tipo `brain` la acción de tomar una decisión, cuando le corresponda y hacerle esperar hasta que le vuelva a tocar su turno.

Asimismo, en caso de que el juego esté en ejecución en modo automático, este empleará un objeto de la subclase `PlayerBrain`, que permite que el personaje del jugador tome decisiones de forma automática. Esta funcionalidad se explicará en la sección 6.2.

Capítulo 4

Modo desarrollador

A la hora de implementar uno de los algoritmos de la IA de los enemigos, se observó cierta dificultad a la hora de depurar los comportamientos erróneos. Esto se debe, a que en un inicio no se contaba con una forma de colocar a las entidades en las situaciones deseadas. Por este motivo, se desarrollaron las **Herramientas de desarrollo** (Fig.4.1).

Para poder usar las herramientas de desarrollador, es necesario que el juego esté detenido. Para ello se podrá pulsar el botón de *stop*.

En este estado, la cámara principal se **desligará** del personaje del jugador y pasará a estar en control del desarrollador.

Para **desplazar** la cámara hacia arriba, izquierda, abajo y derecha, se emplearán las teclas **W,A,S,D** respectivamente. Además de poder **acercar** la cámara usando la tecla Q y **alejarla** usando la tecla E, dentro de un límite establecido.

En caso de querer reanudar el juego bastará con pulsar el botón de *play*, que parpadeará indicando que el juego no se encuentra detenido.

4.1 Herramientas de desarrollador

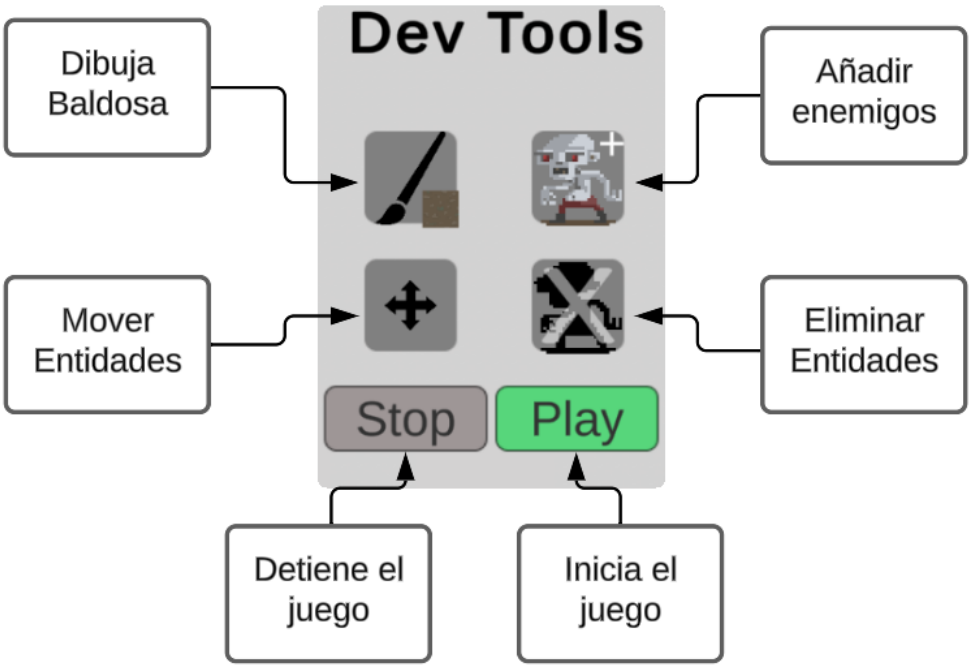


Figura 4.1: Interfaz Herramientas de desarrollo

Las **herramientas de desarrollador** cuentan con 4 funciones principales:

- **añadir enemigos**: Permite añadir enemigos de distinto tipo al mapa.
- **dibujar baldosas**: Permite modificar el mapa añadiendo y borrando baldosas de mazmorra.
- **mover entidades**: Permite desplazar la posición de una entidad de un punto del mapa a otro.
- **eliminar enemigos**: Elimina a una enemigo del mapa.

Asimismo, será necesario monitorizar la situación del ratón, para evitar que al hacer clic a los elementos de la interfaz de depuración, esta sea atravesada y el efecto de la herramienta seleccionada se aplique sobre el mapa.

4.1.1 Añadir enemigos

Esta funcionalidad es imprescindible junto a la de mover enemigos para analizar si la IA de los enemigos funciona correctamente.

La generación de estos enemigos, al igual que sucedía al generar la mazmorra, se basa en instanciar las `prefabs` correspondientes a sus tipos.

Después de haber seleccionado la opción *Añadir Enemigos*, basta con dar un clic en una casilla vacía que no contenga a ninguna entidad. El enemigo se añadirá exactamente cuando se **levante** el botón del ratón.

Esta opción también permite cambiar el enemigo que se está instanciando. Para esto, basta con hacer un clic de nuevo sobre la opción de Añadir Enemigo. Antes de dar clic, si pones el cursor encima de esta opción se mostrará una previsualización sobre el tipo de enemigo al que se cambiará 4.2.

(a) Cursor fuera Añadir Enemigo

(b) Previsualización del siguiente enemigo

Figura 4.2: Control de que entidad se está

4.1.2 Eliminar enemigo

En caso de por error haber generado un enemigo o que haya un enemigo que sea de interés su eliminación, se ha considerado útil tener la capacidad de eliminarla.

La opción de **eliminar enemigos** permite eliminar al enemigo de la baldosa seleccionada. Para esto se comunica con `GameManager`, indicándole que borre el enemigo del juego.

Para que el desarrollador puede detectar fácilmente, que elementos son entidades, en caso de que el cursor pase por encima de estos se volverán más oscuros, indicando que se pueden borrar.

4.1.3 Dibujar Baldosas

La creación de terreno de laberinto y muros, es útil para ver como se desenvuelven las entidades por la mazmorra.

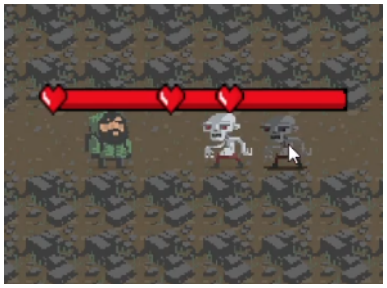


Figura 4.3: Deteccion de entidad mediante puntero

Tras seleccionar la opción *Dibujar Baldosas*, se podrá pintar cualquier baldosa del laberinto, manteniendo el botón del ratón pulsado y permitiendo dibujar sin tener que levantarlo.

El **pincel** podrá pintar distintos **tipos** de baldosa, la baldosa que se encuentra dibujando se encuentra en la esquina inferior izquierda del contenedor *Dibujar Baldosas*. Para cambiarla, se podrá mover el ratón hacia dicha esquina, que provocará un agrandamiento de la baldosa para poder visualizarla mejor 4.4. Para cambiar entre las baldosas disponibles, bastará con hacer clic en la imagen de la baldosa.

En función de la baldosa, en el mapa quedarán reflejados que baldosas se pueden dibujar. En caso de ser una baldosa muro, se teñirán de rojo, las baldosas de la mazmorra que se puedan eliminar 4.5. Mientras que si fuesen baldosas de mazmorra, se teñirán con verde 4.6 los muros que se puedan convertir en estas.

(a) Cursor fuera de Dibuja Baldosa

(b) Cursor en Dibuja Baldosa

(c) Cambio de baldosa

Figura 4.4: Funcionamiento de dibuja baldosa

4.1.4 Mover Entidades

Mover permite mover cualquier entidad a una baldosa libre del laberinto seleccionada.

Una vez seleccionado *Mover Entidades*, para mover a una entidad bastará con mantener pulsado el botón del ratón y moverlo hasta la posición final a la que la quieres mover. En caso de que la posición final, no sea válida debido a que se trata de un muro o una baldosa ocupada, la entidad volverá a su posición anterior.

De la misma forma que en el caso de *Eliminar Enemigo*, si el *sprite* se oscurece quiere decir que es una entidad y por tanto que se puede mover.

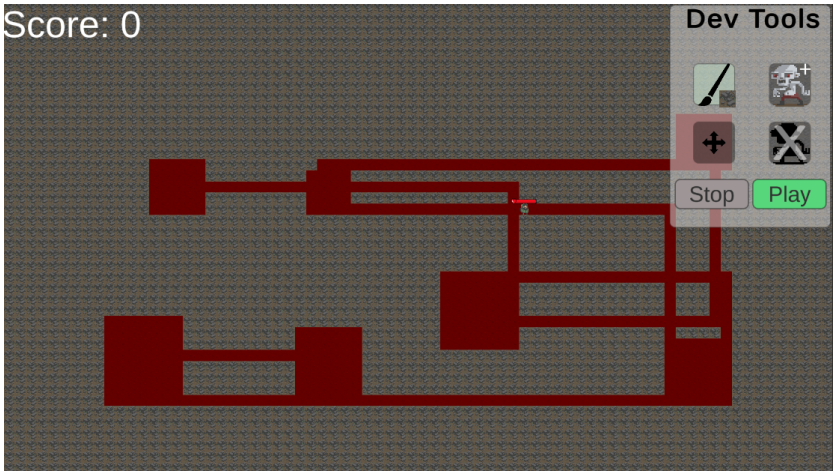


Figura 4.5: Se tiñen de rojo las baldosas de mazmorra que se pueden eliminar

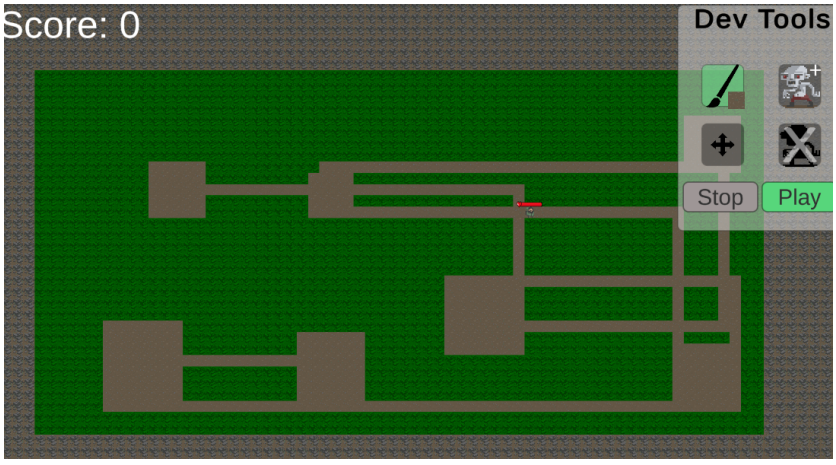


Figura 4.6: Se tiñen de verde donde se pueden dibujar baldosas de mazmorra

Capítulo 5

Control de los enemigos

El objetivo del juego es matar al máximo número de enemigos, por lo que es de vital es de vital importancia que los enemigos sean competentes a la hora de hacer daño al jugador y evitar recibirlo.

Asimismo, es importante que la velocidad en la que los enemigos toman una decisión, ya que en caso de ser lenta perjudicaría la experiencia de juego.

En este apartado se implementarán y probarán a prueba diversos algoritmos y se compararán con el objetivo de saber cuál es el mejor en este tipo de juego.

5.1 Voraz

Esta estrategia se basa en que todos los enemigos actúan de forma agresiva acercándose al jugador, independientemente de las circunstancias.

El funcionamiento se ha representado en la fig.5.1. El enemigo observará a que posiciones se puede mover, p_1, p_2, p_3, p_4, eligiendo aquella que posea la menor distancia euclídea con el jugador.

Sin embargo esto presenta un problema, puede pasar que un enemigo se quede sin moverse porque otro le ha interrumpido el camino. Por ejemplo en la (Fig.5.2), tras moverse el esqueleto 1, hace que el esqueleto 2 pierda la capacidad de moverse hacia la izquierda ya que ese espacio se encontraría ocupado.

Para evitar esto, se ha trabajado sobre una copia del estado de juego, sobre el que se han ido aplicando las acciones de cada enemigo de forma secuencial y se almacenan para luego aplicarlas sobre el juego.

5.1.1 Voraz A*

Al comprobar el comportamiento de los enemigos con el algoritmo voraz, se ha observado que en ocasiones los enemigos se quedan atrapados en los pasillos y en habitaciones impidiendo que salgan.

Para evitar esto lo que se ha hecho es sustituir el algoritmo voraz, por el algoritmo A^*. Como coste se usará donde el número de baldosas recorridas hasta el momento, más la **distancia Manhattan** de la posición actual al enemigo.

Este cambio se ha recogido en la clase `EagerastarEnemiesBrain` Esta posee el siguiente funcionamiento:

1. Los enemigos calcularán el camino al jugador empleando A^*, ignorando a los enemigos del camino, por lo que siempre encontrarán un camino. Este se almacenará en forma de cola, que a su vez se guardarán en un diccionario llamado `memory`, usando al enemigo de clave.

2. Se extrae el siguiente paso de la cola y se comprueba si hay un enemigo interrumpiendo el camino, en caso de ser así no podrá realizar el paso y esperará al siguiente turno.

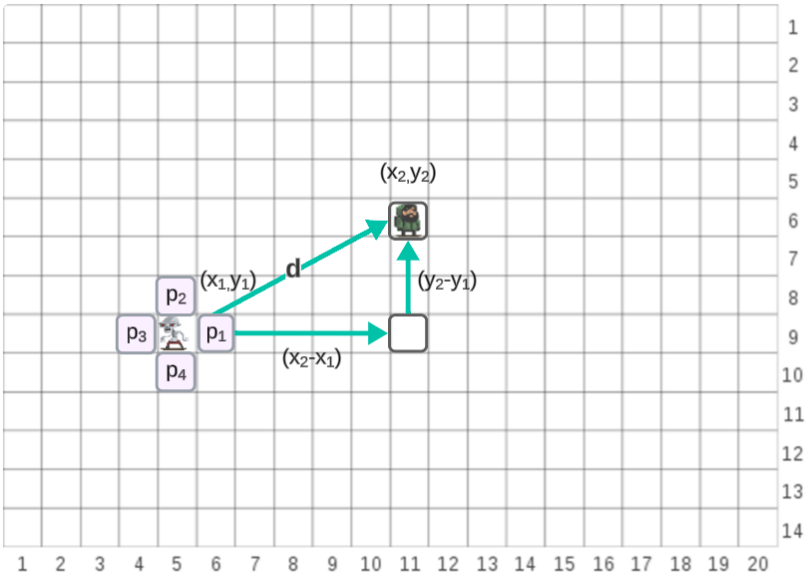


Figura 5.1: Distancia de la posición p1 al jugador

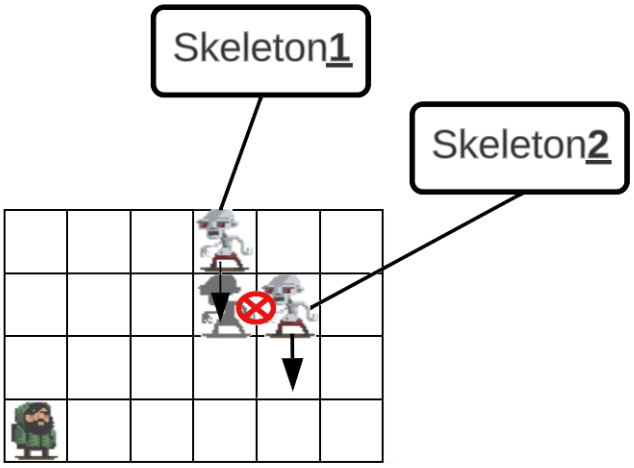


Figura 5.2: Enemigo impidiendo alcanzar la otra baldosa

3. Cuando el jugador se encuentre lo suficientemente cerca del enemigo, es decir cuando queden un número de baldosas fijo para llegar al enemigo o se encuentre a una distancia euclídea fija, se volverá a emplear el algoritmo voraz.

4. Cuando se emplea el algoritmo voraz, la cola generada con el A^* será borrada.

5.2 Poda alfa-beta

El algoritmo poda alfa-beta que se va a implementar presenta una diferencia con respecto al visto en el *background* 2.2.1, en este caso se enfrenta un agente *max* que es el jugador contra varios agentes *min* que es el número de enemigos vivos en ese momento 5.3.

La puntuación de los nodos hoja se calcula con la siguiente fórmula:

$$\mathrm{p} = D + S - N \quad \text{donde:}$$

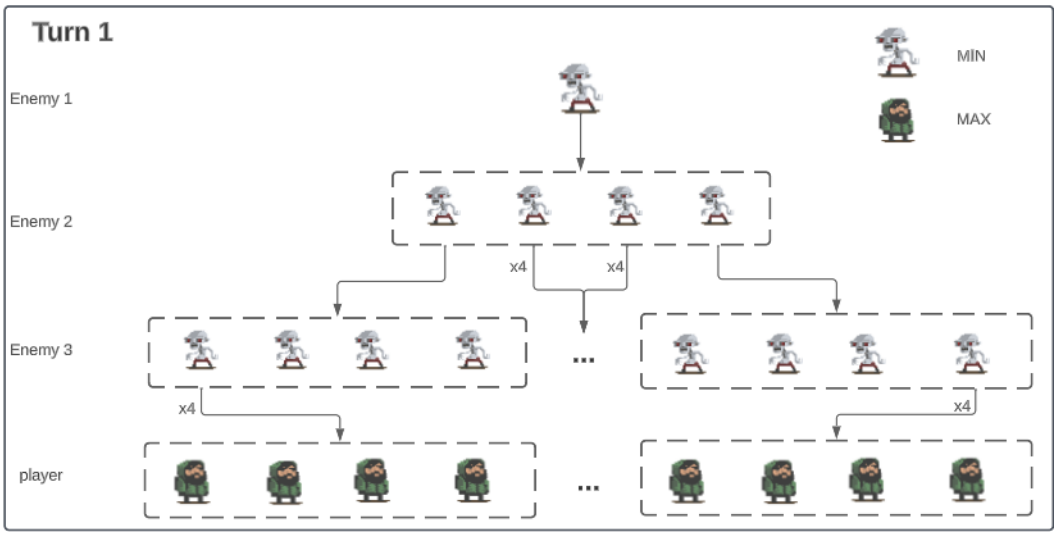


Figura 5.3: Un turno de poda alfa-beta con 3 enemigos

p : Puntuación

D : Daño realizado al jugador

S : Suma de las distancias euclídeas de los enemigos al jugador

N : Número de enemigos asesinados

La poda en este juego presenta problemas ya que a la que un jugador o un enemigo no se encuentren en la misma posición, son dos situaciones distintas. Eso sumado a que en el peor de los casos es que un esqueleto a distancia pueda realizar los 4 movimientos y atacar, dando lugar a a una complejidad computacional de $O(5^{tn})$, donde t son los turnos y n el número de enemigos.

A este problema se le conoce en ingles como *The impossible tree size* [12], existen formas de agilizar este proceso como emplear una estrategia divide y vencerás o introducir un heurístico en la búsqueda.

- Divide y vencerás no es una buena opción ya que por mucho que dividas el espacio de búsqueda este sigue creciendo de forma exponencial.
- Para que el algoritmo poda alfa-beta vaya rápido, los heurísticos que se tienen que aplicar son muy agresivos, por lo que no tiene mucho sentido seguir por este camino, ya que el objetivo de esta es buscar la mejor opción.

Debido a estas razones, la poda alfa-beta imposibilita que haya haya un número de enemigos grande, asimismo al no poder tomarse tampoco decisiones con varios turnos por delante este algoritmo no presenta ningún tipo de inteligencia.

En este algoritmo **no** se ha tenido en cuenta el **dash** 3.6.2, ya que fue descartado antes de ser introducido. Como era infactible dado que no cumplía el requisito no funcional RNF05, se ha considerado un sinsentido modificarlo para que lo tenga cuenta.

5.3 Arboles de decisión

Los árboles de decisión son una forma sencilla de implementar el comportamiento de los enemigos de este juego, debido a su simplicidad permitiendo realizar árboles *adhoc* fácilmente.

Para la implementación de la estructura del árbol de decisión se ha seguido la implementación del libro [11], en la que todos los nodos del árbol heredan de `DecisionTreeNode`.

Se han implementado los siguientes nodos:

```
private DecisionTreeNode trueNode;
private DecisionTreeNode falseNode;
private System.Func<bool> testValue;

6 referencias
public Decision(DecisionTreeNode trueNode, DecisionTreeNode falseNode, System.Func<bool> testValue)
{
    this.trueNode = trueNode;
    this.falseNode = falseNode;
    this.testValue = testValue;

}
```

Figura 5.4: Generalización de la clase decision

- `Decision`: Es una decisión booleana, es decir la respuesta a su pregunta, es un si o no.
- `MultiDecision`: Enruta entre más de una rama hija en base a una pregunta cuyo resultado es un entero.
- `TreeNodeAction`: Contiene la acción a realizar por el enemigo.

Para que el código sea más sencillo se ha introducido una modificación con respecto a la versión del libro. Esta consiste en utilizar `System.Func` de C# para trabajar con programación funcional, por lo que ya no es necesario crear una clase para cada pregunta si no por cada tipo de nodo.

Por ejemplo, con la clase *decision* en el que se ha empleado `System.Func<bool>` para representar las decisiones booleanas. 5.4.

Los árboles de decisión son construidos e interpretados por la clase `DecisionTreeBrain`. Esta permite explorar los árboles de decisión, que consistirá en ir viajando por las ramas respondiendo a preguntas hasta llegar a un `TreeNodeAction`.

Se han implementado dos árboles de comportamiento, uno para los esqueletos a distancia, que son enemigos que pueden atacar desde una posición lejana y otro para el resto de enemigos que son a corta distancia.

5.3.1 Árbol de decisión enemigos a larga distancia

Un enemigo a larga distancia es aquel cuyo rango de ataque es mayor que 1. Pudiendo realizar ataques a más de una baldosa de distancia. Esto les permite atacar al jugador sin que este sea un peligro para ellos, ya que se encuentran lejos y no pueden recibir daño de este. Sin embargo, estos enemigos suelen encontrarse en desventaja cuando se enfrentan a corta distancia, debido a su fragilidad.

Por lo que intuitivamente, un comportamiento inteligente de un enemigo con rango es esconderse detrás de otros enemigos y buscar apoyo

A este tipo de enemigos se les ha asignado el árbol 5.5, este árbol hace que los enemigos tomen una decisión distinta en función de la distancia con el jugador. La rama elegida depende de la distancia del enemigo con respecto al jugador, estas ramas son:

Distancia = 1: Trata de alejarse del jugador y esconderse detrás del enemigo para poder atacar sin ser alcanzado.

Distancia = attackRange: En caso de que pueda atacar, sin ser atacado por el enemigo, aprovechará para golpearlo.

attackRange < Distancia <= 5 : Espera hasta que otro enemigo a melee se encuentre peleando con el jugador, para poder acercarse con más seguridad.

Distancia > 5: Si hay enemigos cerca, se agrupa con ellos y si no busca al jugador.

Este árbol trata de hacer que los enemigos que ataquen a distancia se escondan detrás de los enemigos cercanos, para poder atacar sin recibir daño.

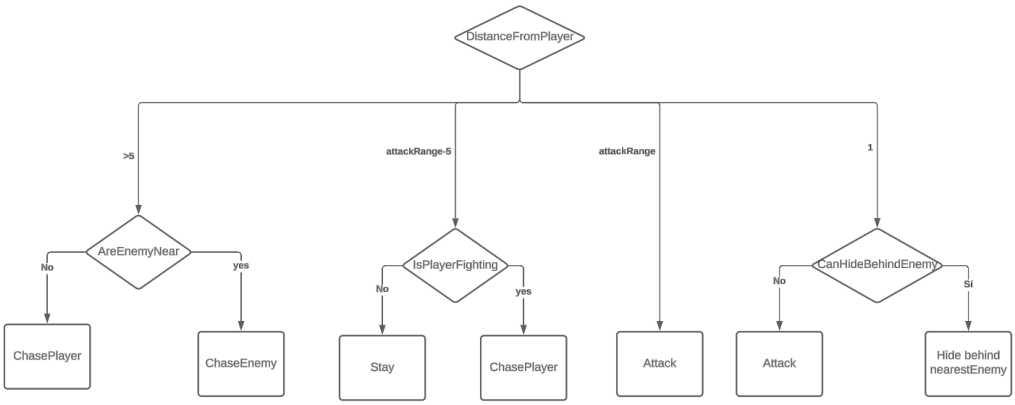


Figura 5.5: Árbol de decisión para enemigos de largo alcance

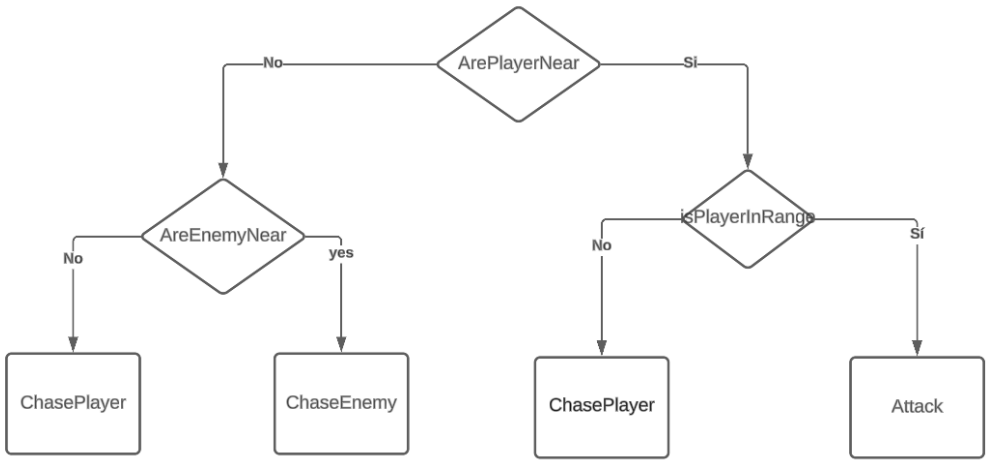


Figura 5.6: Árbol de decisión para árboles de corto alcance

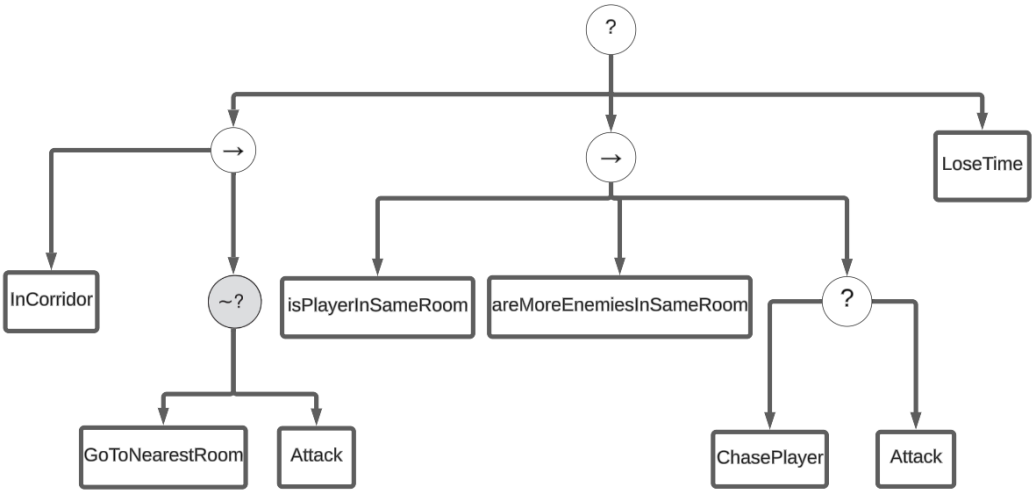


Figura 5.7: Árbol de comportamiento implementado en `BEHAVIOR_TREE_BRAIN`

5.3.2 Árbol de decisión enemigos a corto alcance

Los enemigos de corto alcance son aquellos cuyo rango de ataque es 1, es decir solo pueden atacar a los enemigos que se encuentran a una baldosa de distancia.

A estos enemigos se les ha asignado el árbol 5.6, su funcionamiento es sencillo, teniendo un comportamiento voraz cuando el jugador está cerca, mientras que cuando el jugador está lejos en caso de que haya enemigos cerca se agruparán, para poder enfrentarse al jugador con superioridad numérica.

5.4 Arboles de comportamiento

En este caso a diferencia de la implementación de los árboles de decisión, se empleará un único árbol de comportamiento para todos los tipos de entidades.

El árbol de comportamiento desarrollado 5.7, tendrá como objetivo hacer que los enemigos se agrupen en las habitaciones, para poder tener más espacio y rodear al jugador.

La primera tarea secuencia empezando por la izquierda, se encarga de que los enemigos se agrupen en las habitaciones. El que se encuentra en el medio, se encarga de gestionar el comportamiento de los enemigos cuando se encuentran con el jugador y más enemigos en la misma habitación.

En caso de que ninguna de las dos tareas secuencia anteriores se ejecuten correctamente, dando lugar al caso en que los enemigos se encuentren en una habitación y el jugador no se encuentre en esta, estos enemigos perderán el tiempo.

Capítulo 6

Pruebas

La puntuación de cada partida es en base al número de enemigos que ha matado el jugador antes de morir, dado que no se ha implementado el caso en el que muere el jugador en el test, se le ha asignado una gran cantidad de vida para evitar que muera. Por lo que el daño recibido por el jugador, será el indicador del mejor algoritmo.

Por cada partida se van a almacenar el número de enemigos, su tipo y sus estadísticas, así como el número de habitaciones para comparar los resultados de niveles que se han generado con los mismos parámetros.

También será importante tener en cuenta el número de turnos empleados, para determinar la agresividad de los enemigos, ya que si el enemigo es agresivo entrará rápidamente en contacto con el enemigo y batallarán antes. Esto lleva menos turnos que algoritmos en los que los enemigos emplean turnos agrupándose y huyendo del jugador.

Dado que no cuento con personas que prueben los algoritmos, se ha implementado una funcionalidad en el que el personaje del jugador juega de forma automática, **autoplay** 6.2.

Los algoritmos se probarán corriendo en los mismos escenarios, que se habrán generado al inicio de la ejecución del *script* de test `GameManagerTest`.

Se han probado todos los algoritmos a **excepción** del algoritmo *minimax* con poda alfa-beta, debido a que tardaba **demasiado** en tomar decisiones.

6.1 GameManagerTest

`GameManagerTest` es el *script* encargado de realizar las pruebas, se ha implementado heredando de `GameManager`, aprovechando que se pueden sobrecargar las funciones que controlan el flujo del juego 1.4.1.

El funcionamiento de las pruebas es el siguiente:

1. Se generarán múltiples niveles, variando de 2 a 8 el **número de habitaciones** y de 5 a 50 con pasos de 5 la cantidad de **enemigos**.

   Se generarán varias fases (20) con la misma configuración, para evitar que la posición inicial de las entidades influya en el resultado.

   Los algoritmos ejecutarán los mismos mapas generados de forma independiente.

   El jugador siempre empezará los niveles con la vida al máximo.

2. Será empleado el *autoplay* 6.2, para controlar al jugador, que irá eliminando a los enemigos.

3. Cada vez que finaliza un nivel se almacenarán los puntos de salud restantes del jugador, el número de turnos ejecutados y el número de enemigos asesinados.

4. Cuando se han finalizado todos los niveles, con los resultados parciales obtenidos, se genera un resultado final.

6.2 Autoplay

El **autoplay** es una funcionalidad desarrollada para el juego, que permite que el personaje del jugador sea controlado por una IA de forma automática. El objetivo del *autoplay*, es que sea capaz de llegar a los enemigos sin quedarse repitiendo en un bucle los mismos movimientos. Asimismo, también tendrá que reaccionar en caso de que un enemigo se encuentre cerca.

Se han probado los algoritmos de búsqueda en profundidad, búsqueda en anchura y A^*, llegando a las siguientes conclusiones.

Búsqueda en anchura : Permite llegar al enemigo más cercano en el mínimo de pasos posibles. Cuando el enemigo se encuentra lejos este que explorará muchas baldosas para encontrar la ruta óptima. Sin embargo, cuando el objetivo está cerca, resultan más eficiente que A^*, debido a que una cola que es más eficiente que una cola de prioridad al no necesitar que los nodos descubiertos estén ordenados.

Búsqueda en profundidad : El jugador da demasiadas vueltas e incluso llega a esquivar el objetivo, dando una gran cantidad de pasos para llegar al objetivo.

A^* : El heurístico empleado será distancia Manhattan que en nuestro caso es el número de pasos mínimos que costaría llegar desde el punto actual al destino. Como ese es el mejor de los casos, pudiendo ser peor si hay muros de por medio, la distancia real será siempre igual o superior al heurístico y por tanto nunca se sobreestimará.

En base a esto, $A*$ encontrará el camino más corto.

Viendo las fortalezas y debilidades de los distintos algoritmos se ha decidido implementar una combinación de búsqueda en anchura y A^*. Empleando la búsqueda en anchura con una profundidad limitada para comprobar si hay enemigos cerca, en caso de que no se encuentre ninguno se utilizará A^* para ir al enemigo más cercano.

Por último, la velocidad de funcionamiento del *autoplay* depende de lo que tardan en moverse todas las entidades. La velocidad de animación de las entidades está controlada por el atributo `MOVEMENT_SPEED` de la clase `MovableEntity`, por lo que para modificar la velocidad de ejecución bastará con cambiar los valores de este atributo.

6.3 Resultados

En este apartado se realizará una comparación de los resultados generados empleando un `script` de `python` que se encuentra en el repositorio de `github`.

6.4 Análisis de daño recibido por el jugador

6.4.1 Resultados de la ejecución con pocos enemigos

En la fig. 6.1, los gráficos de lineas representan el daño para un número de habitaciones variable de 2 a 8 y un número fijo de enemigos 5 en el caso de la gráfica de la izquierda y 10 en el de la derecha.

En ambas gráficas se observa, que los resultados de todos los algoritmos son más o menos similares a excepción del árbol de comportamiento, perdiendo prácticamente en casi todas las pruebas, independientemente del número de habitaciones.

Asimismo, se observa que para pocos enemigos el número de habitaciones no ha influido demasiado en el daño causado al jugador, ya que pese a que para 10 enemigos y 2 habitaciones se registre un daño elevado, en el caso de los algoritmos voraces y del árbol de decisión, el daño de 10 enemigos con 5 habitaciones, supera al daño con los mismos enemigos configurados con 3 y 4 habitaciones.

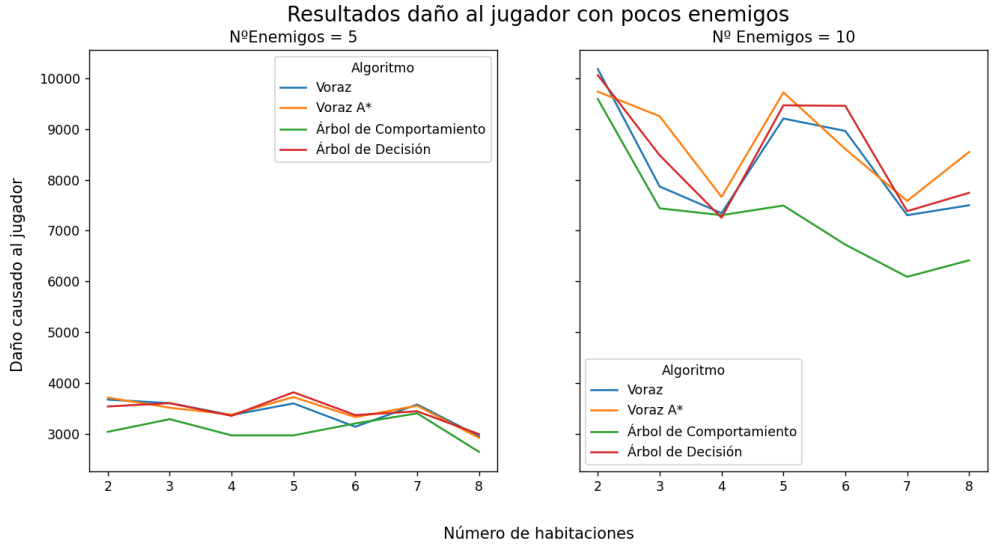


Figura 6.1: Gráficas con el daño causado al jugador por los diferentes enemigos cuando hay pocos enemigos

6.4.2 Resultados de la ejecución con muchos enemigos

En este caso es de esperar que ha medida que suben los enemigos, se concentrarán más enemigos por sala y por tanto las estrategias conservadoras, en la que los enemigos trataban de agruparse perderán fuerza.

Los datos de las ejecuciones con muchos enemigos, se han plasmado en la fig.6.3, donde en el eje y se representa la vida restante del jugador, por lo que el algoritmo que posea la línea que se encuentre más abajo será el mejor. En el eje x se encuentra la configuración seleccionada.

En esta figura se pueden observar dos cosas:

1. Cuanto menor número de habitaciones y por tanto más densidad de enemigos por habitación, provocan que el jugador reciba más daño ya que se encuentra rodeado constantemente de enemigos. Por lo que las diferencias entre los algoritmos son menores que cuando el número de habitaciones es mayor.

   Estos se debe también, a que en los algoritmos árbol de comportamiento y árbol de decisión se tratan de agrupar los enemigos esperando al jugador. Mientras que en el caso de los algoritmos voraces, debido a su agresividad, los enemigos de otras habitaciones pueden llegar al jugador antes de que este elimine a sus enemigos cercanos, esto favorece que los enemigos rodeen al jugador, provocando que el número de habitaciones sea menos relevante con respecto al daño recibido por el jugador.

2. Asimismo, se observa un claro ganador que es el algoritmo voraz A^*, teniendo un rendimiento superior en prácticamente en todas las pruebas.

6.5 Análisis número de turnos empleados

6.5.1 Análisis con pocos enemigos

Al observar los resultados 6.2, se aprecia que el árbol de comportamiento es el algoritmo que más turnos ha empleado con llegando incluso a prácticamente duplicar en turnos al resto de algoritmos, siendo más notable cuando el número de habitaciones es mayor.

6.5.2 Análisis con muchos enemigos

En este caso, los resultados 6.4, la diferencia que había anteriormente entre el árbol de comportamiento y los algoritmos voraces se ha mantenido, sin embargo esta diferencia se

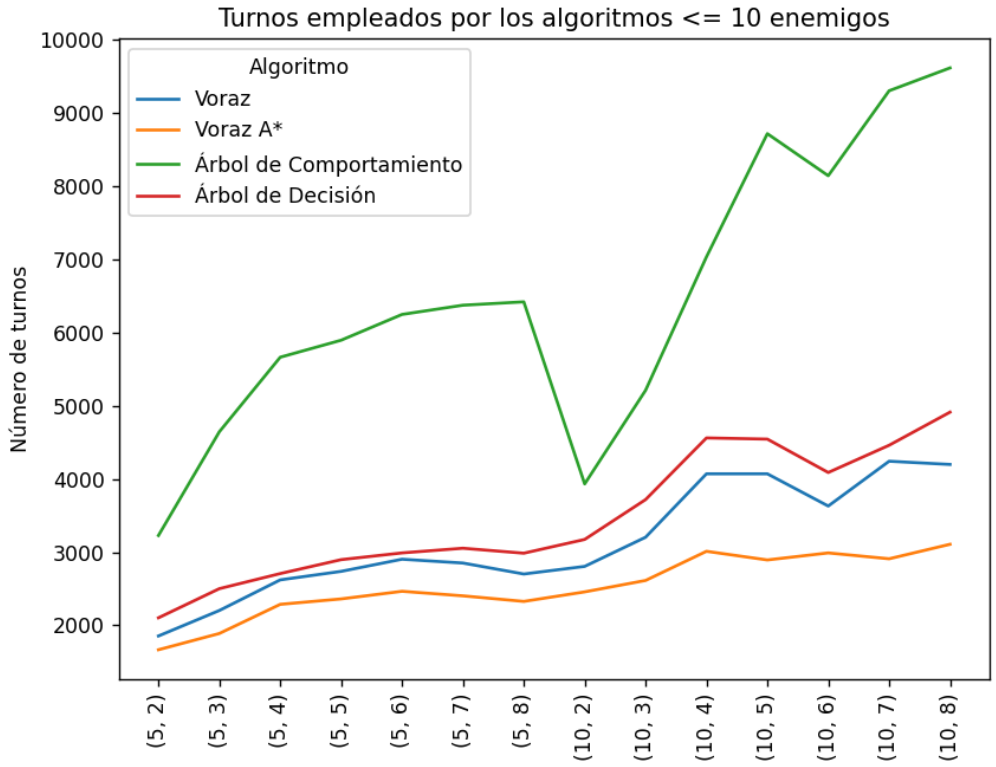


Figura 6.2: Turnos empleados por los enemigos con las distintas configuraciones de habitaciones y enemigos

ha reducido con el árbol de decisión, llegando a estar incluso por encima en turnos del árbol de comportamiento para los experimentos con 2 habitaciones.

Esto es un indicador de que el árbol de comportamiento es más agresivo que el árbol de decisión cuando están todos en la misma habitación, pero cuando se dispersan los enemigos este pierde agresividad de forma considerable dando lugar a una línea con forma de sierra.

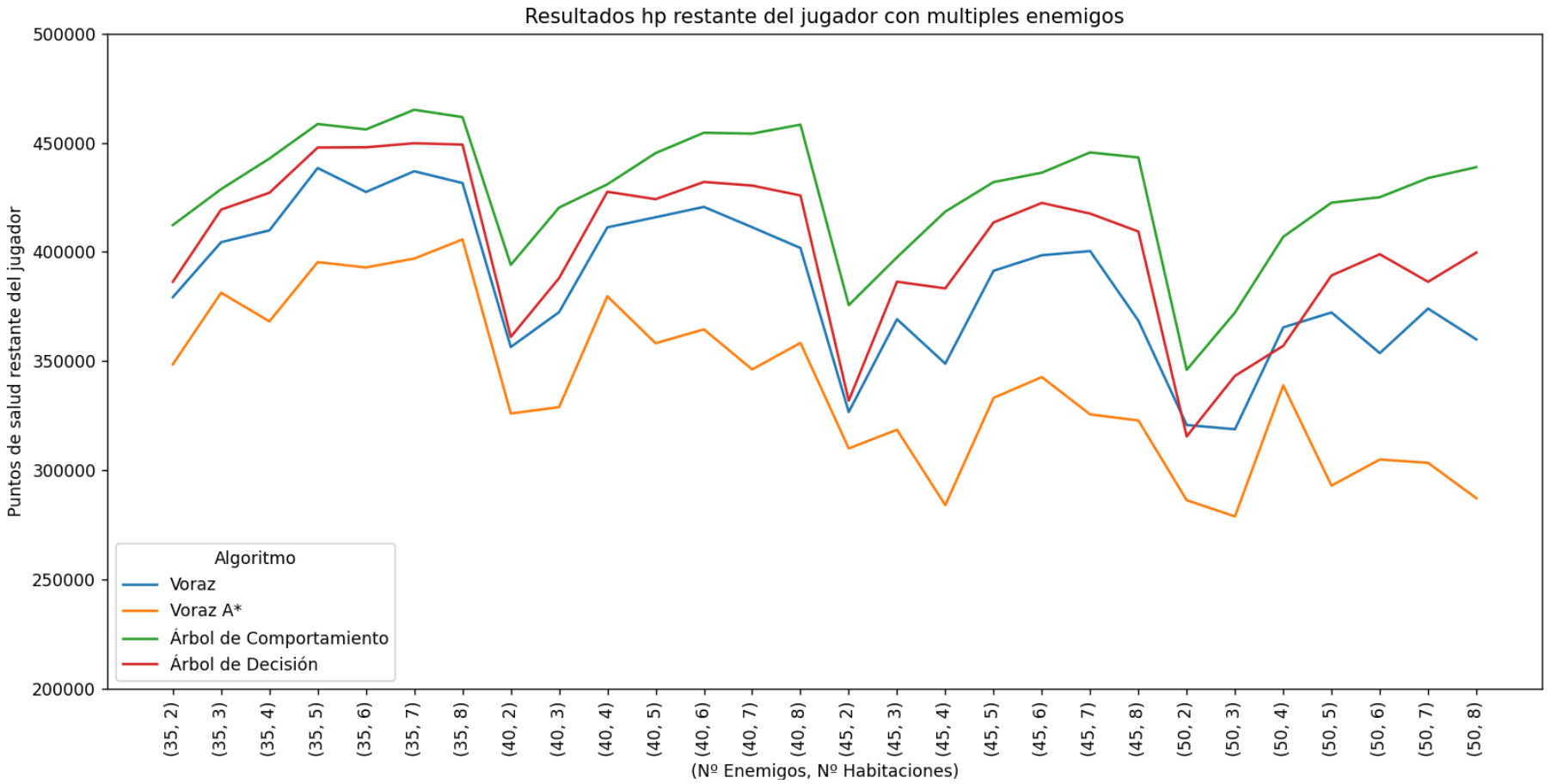


Figura 6.3: Resultados de los experimentos con muchos enemigos

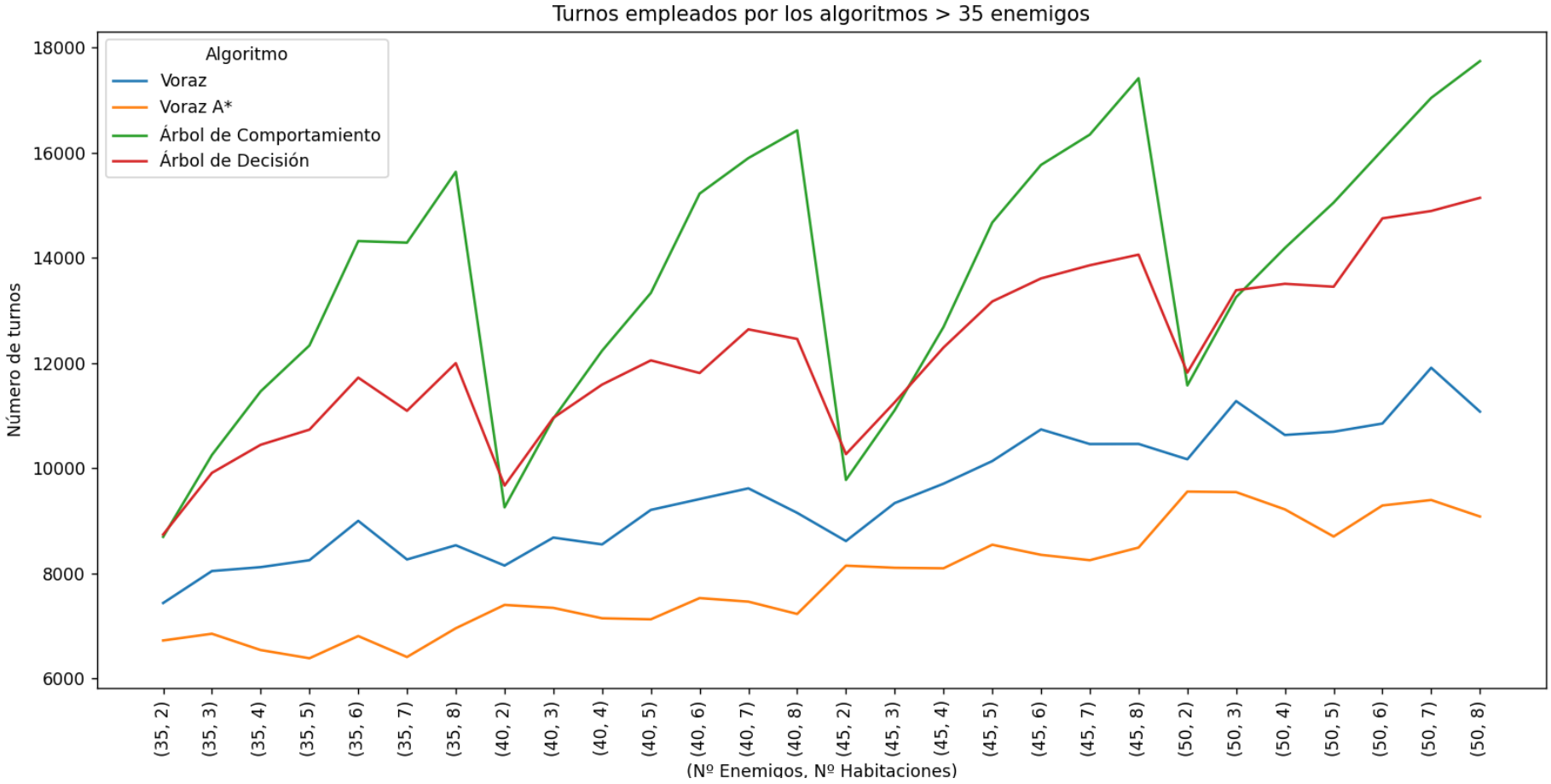


Figura 6.4: Resultado turnos con muchos enemigos

6.5.3 Análisis resultados

De entre los algoritmos voraces y los árboles de comportamiento y decisión generados, ha resultado vencedor el algoritmo voraz con A^*.

Sin embargo, a la hora de jugar estos no parecen tan inteligentes como podrían ser tanto el árbol de decisión como el árbol de comportamiento. Estos resultados se deben, a que no hay otros elementos como podrían ser objetos curativos o que los enemigos puedan puntos de su salud cada cierto tiempo.

La única ventaja que tienen los enemigos al agruparse es rodear a los enemigos y golpearlos entre varios, lo cual dado al tamaño del mapa de las pruebas, el algoritmo que mejor lo aprovecha es el voraz con A^*, ya que mientras el jugador se pega con los enemigos de una habitación, los demás llegan y vuelven a rodearlo constantemente.

Tampoco se ha presentado mucho la debilidad de los algoritmos voraces, que es que el jugador pelease en un pasillo de una baldosa de grosor, lo que limita el número de enemigos que pueden rodearle a 2. Mientras que en el árbol de comportamiento por ejemplo, los enemigos tratarían de llegar a una habitación, ya que es una situación más ventajosa para ellos.

Capítulo 7

Conclusiones y futuros trabajos

7.1 Conclusión

Partiendo de los objetivos que tenía al inicio del trabajo, se ha logrado desarrollar exitosamente el videojuego, sobre el que se han probado diversos algoritmos.

He llegado a la conclusión, de que desarrollar la inteligencia de los enemigos no es tarea sencilla, especialmente a la hora de depurar no solo los errores en tiempo de ejecución, si no también a nivel de fallos de funcionalidad.

Asimismo, En esta tarea he presenciado comportamientos inesperados, lo que me ha hecho pensar en lo laborioso y el tiempo que requiere al desarrollar la inteligencia en videojuegos más complejos. Debido a esto en comprendido, por que en los videojuegos actuales aparecen este tipo de problemas.

Finalmente, frente a lo que pensaba mientras me informaba sobre los algoritmos, me ha sorprendido que los voraces que son los más simples han resultado más efectivos en las circunstancias de este juego, aunque tras haber jugado al juego, considero que a nivel de jugabilidad el árbol de decisión o el de comportamiento dan una mejor experiencia.

7.1.1 Futuros trabajos

Para un futuro trabajo, se podría añadir más funcionalidades como por ejemplo objetos curativos por la mazmorra, que hagan más interesante la toma de decisiones de los enemigos.

Asimismo, también se podría probar la efectividad de otros algoritmos como máquinas de estados y redes neuronales para controlar a los enemigos.

Bibliography

[1] Estudios y análisis — aevi.org.es. http://www.aevi.org.es/documentacion/estudios-y-analisis/. [Accessed 19-05-2024].

[2] Estadística de cinematografía: Producción, exhibición, distribución y fomento. resultados. https://www.culturaydeporte.gob.es/servicios-al-ciudadano/estadisticas/cultura/mc/culturabase/cine-contenidos-audiovisuales/resultados-cine-contenidos-audiovisuales.html. [Accessed 19-05-2024].

[3] SGAE. Anuario SGAE 2023: La Cultura en datos — sgae.es. https://www.sgae.es/noticia/anuario-sgae-2023-la-cultura-en-datos/. [Accessed 14-05-2024].

[4] Unity Technologies. Unity - Manual: Prefabs — docs.unity3d.com. https://docs.unity3d.com/es/530/Manual/Prefabs.html. [Accessed 20-05-2024].

[5] Unity Technologies. Unity - Manual: Tilemaps — docs.unity3d.com. https://docs.unity3d.com/Manual/Tilemap.html. [Accessed 20-05-2024].

[6] Software de diagramación en línea y solución visual — Lucidchart — lucidchart.com. https://www.lucidchart.com/pages/es. [Accessed 13-05-2024].

[7] S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani. *Algorithms*. McGraw-Hill Higher Education, 2006.

[8] Ian Millington and John Funge. *Artificial Intelligence for Games, Second Edition*, pages "215–220". Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

[9] J.V Neumann. Mathematische annalen. https://gdz.sub.uni-goettingen.de/id/PPN235181684_0100?tify=%7B%22pages%22%3A%5B298%2C299%5D%2C%22pan%22%3A%7B%22x%22%3A0.654%2C%22y%22%3A0.331%7D%2C%22view%22%3A%22info%22%2C%22zoom%22%3A0.312%7D. [Accessed 09-08-2023].

[10] T.P.Hart D.J Edwards. The alfa-beta heuristic. https://dspace.mit.edu/bitstream/handle/1721.1/6098/AIM-030.pdf?sequence=2&isAllowed=y. [Accessed 19-05-2024].

[11] Ian Millington and John Funge. *Artificial Intelligence for Games, Second Edition*, page "200". Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

[12] Ian Millington and John Funge. *Artificial Intelligence for Games, Second Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

[13] Richard Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, jul 1964.

[14] File:Durstenfeld shuffle.svg - Wikimedia Commons — commons.wikimedia.org. https://commons.wikimedia.org/wiki/File:Durstenfeld_shuffle.svg. [Accessed 27-06-2024].

[15] 2D Roguelike - Unity Learn — learn.unity.com. https://learn.unity.com/project/2d-roguelike-tutorial. [Accessed 19-05-2024].