Universidad de Huelva

ESCUELTA TÉCNICA SUPERIOR DE INGENIERÍA

DE LA UNIVERSIDAD DE HUELVA

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Generación procedimental de mapas de tiles 2D. Análisis e investigación del problema.

Autor:

Alejandro Seguí Díaz

Tutores:

Gonzalo A. Aranda Corral

Daniel Márquez Quintanilla

Huelva, 30 de junio de 2015.

Curso académico 2014/15.

Índice general

Índice de figuras

Introducción

En la actualidad, los videojuegos han conseguido posicionarse en un mercado líder indiscutible a nivel internacional. En el ámbito del entretenimiento, la industria cinematográfica está dando paso a los videojuegos, que avanzan a pasos de gigante. Muchas grandes personalidades del cine se dirigen a los creadores de videojuegos para continuar su carrera. Un ejemplo claro de una persona adelantada para su época en este sentido, es George Lucas ([1]), autor de la famosa saga Star Wars. George Lucas se introdujo en el mercado de los videojuegos en el 1986 con el título Labyrinth, dando paso posteriormente a otros títulos muy sonados y de culto como Monkey Island o Grimm Fandango.

Un videojuego, al igual que una película, puede contar una historia. Aún así, existen diferentes géneros de videojuegos, que van desde simulación de juegos de mesa, donde evidentemente no existe historia, o está intrínseca en el origen del mismo juego, hasta las denominadas aventuras gráficas, donde el enfoque está en la parte argumental. Es por ello que la industria del cine deja paso a los videojuegos de manera tan rápida.

Mapas y escenarios.

Muchos géneros de videojuegos plantean su argumento en un escenario donde se da rienda suelta a la capacidad perceptiva del jugador, dando lugar en mayor o menor medida a que complete la historia con su imaginación. En este escenario, el jugador desarrollará las acciones que se le ofrezcan según el tipo de juego. Así, podrá completar la historia, permitiendo además en muchos casos que las acciones influyan en el desarrollo argumental del videojuego. Tipos de juegos que requieren mapas son, por ejemplo, juegos de acción, estrategia o rol entre otros.

El tipo de escenario que podemos encontrarnos en un juego es de lo más variopinto. A la hora de plantear la elaboración de un juego, se tiene en cuenta la elección del tipo de escenario. Muchas cuestiones sobre las que se basan estas elecciones radican sobre la complejidad de optimización del mismo:

- Si el escenario es demasiado grande, no nos interesa renderizar el escenario completo, sino solamente la parte visible. Al proceso de optimizar el renderizado del mapa se le llama *culling*[2].
- Lo mismo pasa con las físicas. No nos interesa comprobar colisiones con elementos que es obvio que no van a colisionar. Ésto también ha de ser tenido en cuenta a la hora de elaborar el sistema de escenarios.
- El *nivel de detalle* (Level of detail por el inglés [3]) es otro aspecto a tener en cuenta. Si una geometría está demasiado lejana del jugador, no necesitamos renderizarla exactamente como es. Este aspecto es relevante a los juegos 3D principalmente.
- La división o no del escenario por zonas. Esto puede evitarnos la necesidad de presencia en memoria de un mapa completo, pudiéndolo dividir en partes para ahorrar en el preciado recurso que es la memoria. Es un aspecto a tener en cuenta sobre todo cuando se elaboran juegos para videoconsolas, las cuales suelen contar con una memoria muy limitada.

Así, se exponen dos criterios principales como forma de clasificar los tipos de escenario:

- Organización del escenario. Este criterio se refiere a la existencia de cortes en el desarrollo del juego para la carga de las distintas partes del mundo en el que se desarrolla el juego. Algunos tipos son:
  - Escenarios totalmente continuos continuos representando un mundo totalmente abierto sin cortes. Ejemplos son *GTAV* o *Minecraft*.
  - Dividido por zonas donde no existen cortes en la misma zona. Un ejemplo es *Borderlands*.
  - Dividido por zonas donde pueden existir cortes en la misma para interiores. Un ejemplo es *Skyrim*.

- Composición del escenario. Nos referimos a si la geometría empleada ayuda de forma implícita al manejo a nivel computacional del escenario. Algunos tipos son:
  - Escenarios en 2D discretizados en *tiles*. Básicamente, la geometría está representada por una matriz de dos dimensiones, donde cada casilla de la matriz está relacionada con una textura (o porción de la misma), siendo todas las texturas del mismo tamaño para la misma matriz. A cada casilla, la denominamos *tile*. Este tipo de escenarios son muy fácilmente optimizables. Ejemplos de este tipo es, por ejemplo, *Final Fantasy VI*, así como multitud de juegos de las consolas de 8 y 16 bits como la *NES* o *SNES*.
  - Una extensión a 3D de los escenarios de tiles son los escenarios compuestos por *voxels* [4]. Estos han sido recientemente popularizados por el famoso título *Minecraft*, donde el escenario está gestionado por un *motor de voxels* que ayuda a optimizar tanto con culling como con físicas.
  - Escenarios con geometría deliberada. Es decir, la geometría (ya sea 2D o 3D), no presenta ninguna propensión a ser optimizada. Este tipo de escenarios, suelen optimizarse con el uso de *quadtrees* [5] (para 2D) u *octrees* [6] (para 3D)
  - Escenarios hechos con los denominados *brushes* [7]. Esta técnica se utiliza para 3D, y consiste en el uso de geometrías convexas para componer un escenario. Ejemplos famosos de este tipo de escenarios es *Quake* [8].

Una vez expuestos los criterios para clasificar los tipos de mapas, remarcar que hay casos en los que los desarrolladores convienen formas nuevas de organizar el mapa debido a necesidades particulares, por lo que los ejemplos expuestos en cada criterio no completan de ninguna manera las múltiples categorías para cada uno.

Para el desarrollo de este proyecto y atendiendo a las especificaciones presentadas en el capítulo 2, abarcaremos *mapas de tiles* para el género de juego *roguelike* [10]. Este género se caracteriza precisamente por mapas generados de forma procedimental. Cada nivel es entendido como una planta, donde el jugador tiene que llegar desde donde aparece hasta un tile considerado como final para poder continuar hasta la siguiente planta. Así, el jugador va avanzando niveles hasta que llega al último y completa el juego al finalizar el mismo.

Motivación y Objetivos.

Como se ha mencionado previamente, la industria de los videojuegos avanza a pasos de gigante. Es por ello que las compañías invierten cada vez más en videojuegos, siendo a veces el coste en marketing superior al de desarrollo [11]. Existe un amplio espectro de puestos encontrados en este campo [12] [13], entre los cuales existen

- *Game programmer*. Personal dedicado a la programación de las mecánicas del juego, eventos y otros relacionados con el propio juego
- *Core programmer*. Personal dedicado a la elaboración del motor que será la base sobre la que funcionará el juego. Normalmente se subdivide en subroles como *graphics programmer* o *physics programmer*. En muchos casos, se parte de un motor ya elaborado sobre el que se realizan las modificaciones necesarias para el juego en cuestión.
- *Game designer*. Dedicados a la elaboración de las mecánicas del juego, así como el plot del mismo.
- *Level designer*. Personal dedicado exclusivamente a la elaboración de escenarios empleando programas externos o el mismo motor que se esté utilizando, si éste posee dichas capacidades.
- *AI programmer*. Dedicado a la programación de los distintos sistemas de inteligencia artificial existentes en el juego.
- *Muchos más...*

En este listado, solo hemos tenido en cuenta algunos de los puestos del aspecto de desarrollo, pero quedaría añadir muchos más como diseñadores gráficos y apartado de marketing.

Destacar que el *level designer*, pese a que no se encarga de la construcción de las herramientas para crear el escenario, debe conocer las bases del tipo de escenario que se empleará en el juego, de forma que adapte sus técnicas de diseño al tipo de escenario y así poder optimizarlos en la medida de lo posible.

Recientemente, y debido a las cada vez más crecientes facilidades para crear un videojuego, han surgido los llamados *equipos indies* [9], caracterizados principalmente por tener un bajo presupuesto y personal. Normalmente suelen empezar con un presupuesto nulo, pero en algunos casos

llegan a triunfar de manera inesperada incluso por los mismos desarrolladores. Ejemplos son *Hotline Miami*, *Minecraft* o *Risk of Rain*.

Los equipos con un presupuesto considerable (coloquialmente denominados AAA), no tienen problemas a la hora de contratar *diseñadores de niveles*, ya que disponen de una gran cantidad de dinero para depositar en los distintos roles necesarios para un juego. Aún así, en combinación con un generador de mapas, un *diseñador de niveles* puede desarrollar ideas que den un resultado muy bueno. Ejemplos de esto son *Diablo* o *Torchlight*, donde los mapas son generados automáticamente partiendo de patrones elaborados a mano por *diseñadores de niveles*.

Así, el la motivación de este proyecto reside en dos ideas principales.

- Ahorro de coste y tiempo para equipos indies, que no pueden permitirse el lujo de gastar en personal exclusivo para el diseño de niveles.
- Adición de variedad a los escenarios de juegos, permitiendo incluso a un equipo de desarrolladores AAA

Solución Propuesta.

La solución que se propone es un sistema capaz de generar escenarios de manera automática con una mínima interacción (o incluso nula si se desea) de los diseñadores. Esto además, conlleva dinamismo en los escenarios que se podrán jugar, de forma que de partida a partida, la distribución del escenario será completamente distinta.

La elaboración de un generador de escenarios, une las competencias de dos roles en el desarrollo de videojuegos:

- *Diseñador de niveles.* Se necesitan conocimientos sobre la composición de los escenarios del juego, además de las propiedades concretas en cuanto a los criterios mencionados anteriormente.
- *Programador de IA*. La creación del sistema que genere los escenarios es un trabajo que compete a este rol, ya que estamos tratando de resolver un problema donde las posibilidades de resolución son casi infinitas.

Finalmente, se elaborará el sistema reduciendo el problema a una búsqueda, donde el *espacio de estados* son todas las posibles combinaciones de habitaciones conectadas.

Estructura de la memoria.

Esta memoria se estructura en varios capítulos, con la siguiente distribución de los temas trabajados:

- Capítulo 1. Mostraremos el origen y estado actual del campo de generación procedimental de contenido.
- Capítulo 2. Definiremos el problema junto con sus especificaciones y requisitos. Se comentará el tipo de juego al que se ha enfocado y el tipo de mapa utilizado.
- Capítulo 3. Analizaremos en profundidad la representación del modelo de datos elegida para el desarrollo del generador propuesto.
- Capítulo 4. Desarrollo de la estrategia empleada para el generador. Como se podrá comprobar, se ha utilizado una estrategia flexible sobre la que posteriormente, construiremos el método de búsqueda.
- Capítulo 5. Detalle de la implementación basada en búsqueda. Analizaremos cada uno de los elementos empleados tanto para la generación del mapa, como para la adición de diversidad.
- Capítulo 6. Una vez explicada la estrategia, en este capítulo se desarrollará una correspondencia entre el método de búsqueda y la estrategia implementada basada en el mismo.
- Capítulo 7. Comparación y análisis del impacto de diversos componentes que se han empleado para promover tanto la diversidad como la eficiencia.
- Capítulo 8. Posible trabajo futuro pertinente a la elaboración de una aplicación comercial con el método expuesto en este proyecto
- Bibliografía.

Capítulo 1

Estado del arte.

Analizaremos el ámbito de la generación procedimental de contenido para videojuegos. Veremos las distintas subdisciplinas presentes, así como los problemas que resuelven. Se hará hincapié en la generación de escenarios, ya que es el campo que compete a este proyecto.

1.1. Generación procedimental de contenido.

Más conocida por su nombre en inglés (Procedural Content Generation), se refiere a la disciplina de generar contenido audiovisual mediante de algoritmos en lugar de hacerlo manualmente, comunmente usado en videojuegos.

Tuvo sus inicios en la subcultura informática llamada *Demoscene* [14]. Este movimiento tuvo sus inicios a finales de los años 70 y principios de los 80, y continúa a día de hoy. Consiste en crear contenido visual y sonoro de forma programada, ya sea en parte o en su totalidad, y tenía como uno de sus objetivos escudriñar al máximo las limitadas capacidades de los ordenadores de la época.

Los *sceners* de este campo, conseguían generar contenidos que eran impensables para la época, como por ejemplo escenas en 3D cuando a OpenGL aún le quedaban un par de décadas para aparecer. Por su habilidad, la mayoria de los demosceners terminaban trabajando para empresas de videojuegos de la época, cuando aún no había tantas facilidades

a la hora de crearlos.

Existen diversas subdisciplinas en las que se aplica el concepto de generación procedimental de contenido:

- Texturas [15]. Un ejemplo de ello es la generación de texturas que imitan el mármol o la madera. En algunos casos se emplean autómatas celulares.
- Geometría. Generación de follaje, o árboles empleando L-Systems. [15]
- Mecánicas. El juego Left4Dead y su secuela, emplean un sistema procedimental para gestionar los momentos de tensión y la dificultad de las situaciones según las acciones que han ido tomando los jugadores.
- *Escenarios*. Este aspecto es en el que nos enfocaremos en el proyecto, concretamente para mapas de tiles 2D.

Otra clasificación de los métodos procedimentales es según el momento de ejecución del procedimiento. Si el procedimiento es ejecutado antes del lanzamiento del juego, o después con la particularidad de que se realiza en servidores ajenos al jugador, lo denominaremos *offline*. Sin embargo, si el procesamiento se realiza en el sistema en que se está ejecutando el juego, lo llamaremos *online*.

1.2. Generación de mapas.

La generación de escenarios se considera un concepto muy amplio, ya que según el contexto del juego, puede variar bastante. Por ejemplo, en un juego ambientado en el espacio exterior, el escenario puede entenderse como una galaxia completa, como es el caso de *Elite: Dangerous*.

Otro tipo de modelo a usar en la generación de escenarios es el ruido Perlin. Éste puede ser muy útil en conjunto con el concepto de mapas de alturas para generar terrenos al aire libre montañosos [16]. También se ha empleado el *algoritmo de Voronoi* para la generación de terrenos [17].

Algún ejemplo más histórico de generación de escenarios pueden ser los múltiples algoritmos de generación de laberintos que se conocen [18].

Debido a que los métodos anteriores son demasiado genéricos, una opción muy popular es idear y construir un generador específico para el juego en cuestión que se esté desarrollando. Un ejemplo de ello es *TinyKeep*, cuyo autor describe a grandes rasgos en [19] las fases y el desarrollo del algoritmo que ha creado.

En este proyecto se desarrollará un generador propio, procurando mantener un nivel de genericidad y capacidad de personalización del sistema.

Capítulo 2

Especificaciones.

Pasaremos a definir el problema, así como las especificaciones y directrices tanto del tipo de juego como del sistema de generación y escenarios. Dichas directrices que delimitarán el sistema a crear, han sido establecidas por la empresa indie de videojuegos *TheGameKitchen*[**?**].

2.1. Género de juego.

En esta sección abarcaremos el tipo de juego y las directrices establecidas en cuanto al género de juego y otras especificaciones relacionadas con el mundo en el que se desarrollará el mismo.

2.1.1. Roguelike

El género de juego en el que nos basaremos será *roguelike* [10]. El nombre proviene del juego *Rogue* que sirvió de fuente de inspiración para que desarrolladores elaboraran juegos con mecánicas similares.

Las características principales son:

- *Generación de mazmorras*. Los escenarios suelen ser generados automáticamente, así, cada vez que el jugador inicia una partida, la experiencia es ligeramente distinta.

- *Importancia considerable a la exploración*. El hecho de que las mazmorras no sean siempre iguales, incita al jugador a tener que invertir tiempo en explorar para poder encontrar el objetivo.
- *Desarrollo del juego por plantas*. El objetivo del jugador suele ser llegar a una habitación considerada como final, donde puede elegir entre pasar a una siguiente planta, o investigar un poco más en la presente.
- *Dificultad progresiva*. Cada planta tendrá una dificultad ligeramente mayor a la de la anterior hasta llegar a la última.
- *Muerte permanente*. Una vez que el jugador muere no hay manera de cargar la partida. La unica opción es comenzar de nuevo.

Históricamente, el género *roguelike* se identificaba además por otro tipo de características, como la mecánica por turnos o el énfasis en jugabilidad y desinterés en los gráficos, pero con el tiempo, el género se ha ido abriendo paso a una definición más genérica.

Debido a ésto, hoy en día existen desde *shooters* considerados *roguelike*, como por ejemplo *Tower of Guns* o *Paranautical Activity*, hasta *plataformas*, como *Risk of Rain* o *Spelunky*.

2.1.2. Especificaciones del mundo

Las especificaciones impuestas referidas a las reglas de juego son las siguientes:

- Entorno en dos dimensiones empleando mapas de tiles
- El jugador comenzará en una habitación y tendrá como objetivo llegar a una habitación final
- Se empleará una vista cenital (figura 2.2).

Estas especificaciones son bastante genéricas, por lo que el sistema elaborado podrá servirnos para otro tipo de juegos. Más adelante veremos que se ha procurado dar un enfoque de alta flexibilidad al sistema para cubrir este aspecto.

Figura 2.1: Vista cenital en el juego Action Hollywood por TCH, 1995

2.2. Requisitos del escenario.

La representación de los escenarios elegida será un *mapa de tiles*. Como se comentó anteriormente, los mapas de tiles están representados por una matriz, donde cada casilla a la que denominaremos *tile*, corresponde a un gráfico de un tamaño normalizado para todos los tiles (un gráfico para las paredes, otro para el suelo, etc).

Suelen componerse por capas para añadir decorado, pero para nuestra finalidad, hemos obviado esta característica, ya que se hará hincapié principalmente en la distribución de las habitaciones. No obstante, sería muy sencillo añadir capas como mejora futura.

La optimización en cuanto a *culling*[2] en un mapa de tiles está muy investigada. Debido a que el propio mapa es una rejilla, se puede recorrer y renderizar exclusivamente la zona que va a verse teniendo en cuenta la cámara.

En la figura 2.2 se representa en rojo el viewport del jugador. Los tiles amarillos están en el borde del viewport, y los verdes están completamente dentro. Así, solo hace falta recorrer y dibujar los tiles verdes y amarillos, ahorrándonos el dibujado de los azules.

Otra característica importante de los mapas de tiles es la sencillez y flexibilidad que da a la hora de componer escenarios; podemos reutilizar tiles

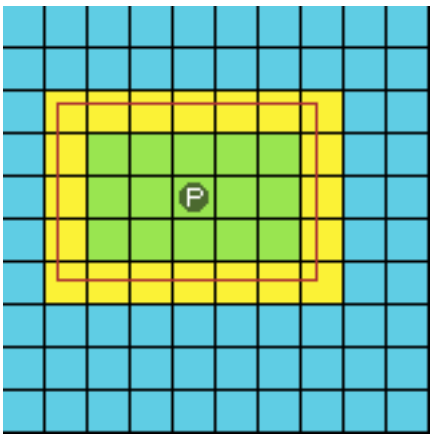

Figura 2.2: Clipping en un mapa de tiles

y añadir variaciones de los mismos para, en el mapa final, dar sensación de variedad sin tener que elaborar muchos gráficos.

Por todo ello, es un tipo de escenario muy popular desde las antiguas consolas como *NES* o *SNES* hasta hoy.

El escenario estará compuesto por habitaciones, cada una representada como una matriz. Será nuestra tarea plantear un sistema que elabore distribuciones de las habitaciones según los requisitos que se expondrán a continuación.

2.3. Directrices para la generación.

En esta sección explicaremos las directrices impuestas para la generación de los escenarios.

2.3.1. Lista inicial de habitaciones

El sistema que construiremos deberá cumplir una serie de características que analizaremos en detalle en esta sección. Como se ha mencionado previamente, se ha tomado la libertad de añadir características que no estaban en los requisitos previos de *TheGameKitchen*[**?**].

El sistema debe generar un mapa de tiles partiendo de una *lista inicial*

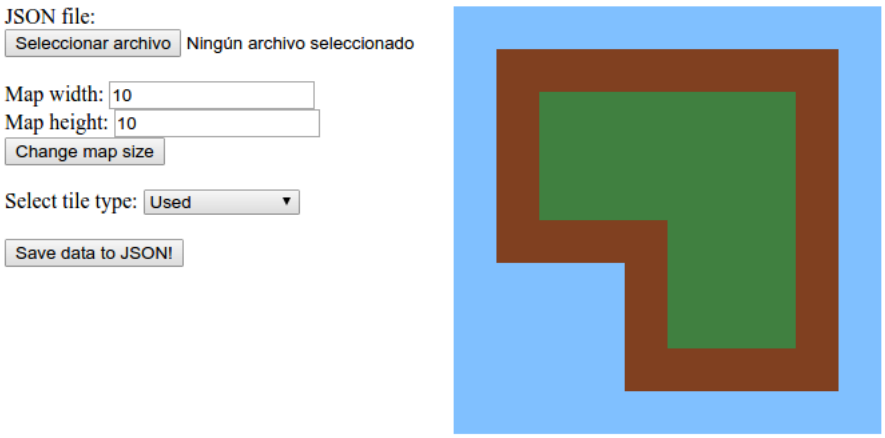

Figura 2.3: Editor de habitaciones

de habitaciones previamente construida. Para construir las habitaciones, se ha elaborado un editor via web usando el canvas que proporciona HTML y JavaScript (se puede ver en la figura 2.3).

Así, a grandes rasgos el objetivo será colocar las habitaciones presentes en esta lista inicial en un mapa de tiles vacío.

Las habitaciones *pueden repetirse*, es decir, si hemos creado dos modelos de habitación A y B, podemos tener 3 instancias del modelo A y 2 instancias del modelo B en la lista inicial. Hablaremos de este detalle en el próximo capítulo, que nos servirá para conseguir una mejora de optimización tanto de memoria como en procesamiento.

2.3.2. Guía para la distribución de habitaciones

Se ha de *maximizar el camino* desde la habitación inicial hasta la habitación final. De esta forma conseguimos en parte promover que el jugador tenga que investigar.

Obviamente, se ha de conseguir una variabilidad en los niveles añadiendo un componente aleatorio, de forma que la experiencia de cada jugador (o incluso del mismo en partidas distintas) no sea idéntica.

Es indiferente que par de habitaciones son la inicial y la final, podemos elegir cualquier par.

Partiremos de un mapa de tiles vacío, cuyo tamaño supondremos suficiente como para albergar todas las habitaciones para cualquier distribución de las mismas. Para ello, es posible crear un mapa con tamaño autoajustable según sea necesario, pero se ha optado por elegir un tamaño lo suficientemente grande para hacer las pruebas.

El tamaño del escenario no sobrepasará un área de 64x64 tiles. Este tamaño no es una restricción fuerte, sino una guía para saber a qué tamaños nos enfrentaremos. Debido a esto, el área final del escenario puede ser algo menor o mayor a este tamaño de 64x64 tiles.

Se ha de fomentar la existencia de caminos alternativos, no necesariamente caminos alternativos a la solución, sino más bien callejones sin salida, de forma que incite aún más a la exploración.

2.3.3. Requisitos técnicos

La generación será *online*, implicando que el sistema ha de actuar en un tiempo considerable para evitar largas esperas. Para esto, se ha construido un sistema de caché del que hablaremos más adelante.

Debe funcionar en sistemas móviles. Esto implica un especial cuidado en la optimización y en las librerías utilizadas. Para abarcar este punto correctamente, se ha utilizado Java como lenguaje de programación, ya que es el más popular para sistemas móviles. Además, el sistema de caché que se mencionó antes, nos servirá también en este aspecto.

Capítulo 3

Representación.

Entraremos en detalle de la representación de los distintos elementos que componen el generador. Repasaremos el concepto de *mapa de tiles* de nuevo. Analizaremos la representación lógica y física necesaria con el objetivo de establecer un modelo claro para las habitaciones y el mapa que usaremos posteriormente para la construcción del sistema de generación.

En nuestro contexto, disponemos de tres tipos de entidades:

- *Habitaciones* que compondrán el mapa.
- *Mapa* o escenario donde se desarrolla el juego, compuesto por habitaciones. Posee algunos elementos ausentes en las habitaciones.
- *Puertas* que servirán de conexión entre habitaciones.

Es importante tener claro el aspecto de la representación de las entidades que se utilizarán en el sistema, ya que el mismo se construirá en base a lo que se detalle en la representación.

3.1. Topología.

Como se ha repetido varias veces anteriormente, usaremos un *mapa de tiles* para la representación del escenario. Esto implica que los modelos tanto de las habitaciones como del propio mapa, han de mantener una matriz

para la representación física. En la figura 3.1 podemos ver la representación en memoria como una matriz de un mapa de tiles. En la figura 3.2 se observa la representación gráfica. Se puede comprobar la relación directa entre ambas representaciones.

$$\begin{matrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{matrix}$$

Figura 3.1: Representación en memoria de un mapa de tiles

En principio, los tipos de tile que se han considerado son los siguientes:

- *Tile pared*. Representa una porción de geometría sólida infranqueable. A la hora de colocar habitaciones, habrá que añadir un mecanismo para evitar el solapamiento entre tiles sólidos de entidades distintas.
- *Tile interior*. Representa una porción de geometría traspasable. A igual que el tile sólido, habrá que evitar solapamiento con tiles pertenecientes a otras entidades (habitaciones o el mapa).
- *Tile exterior*. Representa una porción de geometría traspasable *externa* a la habitación. Este tipo de tiles sí que puede solaparse con tiles pertenecientes a otras entidades.
- *Tile puerta*. Representa una conexión entre dos habitaciones. No podrá solaparse, pero si traspasarse para permitir el cruce entre habitaciones.

En la figura 3.3 se ve un resumen de las propiedades inherentes a cada tipo de tile.

Con todo ésto, ya tenemos clara la representación física de los distintos elementos implicados en el sistema de generación. Ahora, entraremos en detalle con elementos específicos para el mapa y la habitación, que principalmente nos serán necesarios para algunas optimizaciones y llevar a cabo la lógica del generador.

Más adelante veremos un tipo de tile especial (tile tipo *puerta*) en el que nos apoyaremos para la conexión entre habitaciones.

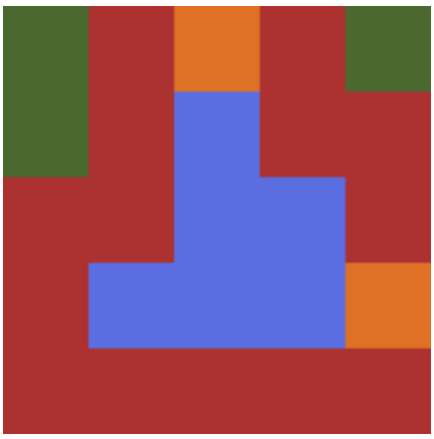

Figura 3.2: Representación gráfica de un mapa de tiles

	Sólido	Solapable
Pared	✓	X
Interno	X	X
Externo	X	✓
Puerta	X	X

Figura 3.3: Propiedades de los tiles

3.2. Habitaciones.

Analizaremos los elementos relevantes para la representación de las habitaciones. Atendiendo a la especificación de que las habitaciones *se pueden repetir*, se ha dividido la representación en dos componentes:

- *Prefab*. Representación que engloba los elementos comunes para un modelo de habitación. Sirve, entre otras cosas, para almacenar solamente un mapa de tiles por modelo de habitación.
- *Instancia*. Representación específica para una concretización o instancia de un modelo de habitación o prefab. Contendrá información sobre la posición de una habitación en el mapa y las puertas.

Ésto nos permite un ahorro de memoria, ya que guardamos información común a muchas habitaciones en un solo modelo, y procesamiento, que veremos en detalle en el capítulo 4 (INSERTAENLACE).

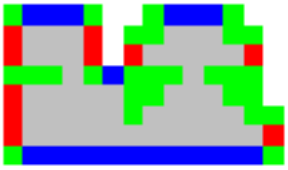

Figura 3.4: Representación gráfica de puertas potenciales verticales y horizontales

3.2.1. Puertas potenciales.

Antes de abarcar los dos componentes empleados para representar las habitaciones, vamos a introducir el concepto de *puerta potencial*, que será necesario para explicar una de las motivaciones del *prefab*, además de tener que mantener solo un mapa de tiles por modelo.

Con el concepto de puertas potenciales nos referimos a las posibles conexiones de un modelo de habitación. Cualquier tile que cumpla la restricción de ser puerta potencial, puede convertirse en puerta para establecer una conexión entre habitaciones en el transcurso de la generación en el sistema.

Así, consideraremos que un tile $t(x, y)$ será una puerta potencial de la habitación R teniendo en cuenta las dos orientaciones posibles.

- *Horizontal* si se cumple:
  - Existe un tile *pared* a la izquierda y derecha del tile.
  - Existe un tile *interior* arriba y un tile *exterior* abajo, o viceversa.
- *Vertical* si se cumple:
  - Existe un tile *pared* arriba y abajo del tile.
  - Existe un tile *interior* a la izquierda y un tile *exterior* a la derecha, o viceversa.

En la figura 3.5 se puede ver la definición formal del concepto de *puerta potencial*, y en la figura 3.4, una representación gráfica, teniendo en cuenta ambos casos.

$$t \in R, \begin{cases} outer(x+1,y), & inner(x-1,y), & wall(x,y\pm 1) & vertical \\ outer(x-1,y), & inner(x+1,y), & wall(x,y\pm 1) & vertical \\ outer(x,y+1), & inner(x,y-1), & wall(x\pm 1,y) & horizontal \\ outer(x,y-1), & inner(x,y+1), & wall(x\pm 1,y) & horizontal \end{cases}$$

Figura 3.5: Definición formal de una puerta potencial y sus casos

Según la complejidad de las habitaciones, con este concepto conseguiremos ahorrarnos casos en los que será imposible conectar habitaciones. Obviamente, habrá casos en los que aún siendo un tile una puerta potencial, es imposible conectarla con otra habitación por motivos de solapamiento entre las mismas. Aún así, podemos ahorrarnos parte de las comprobaciones.

3.2.2. Prefabs.

Como se ha mencionado anteriormente, los prefabs representarán el modelo de un tipo de habitación, común a todas las instancias o concretizaciones del mismo. Las motivaciones de este componente son de eficiencia, tanto de memoria como de procesamiento.

En cuanto a memoria, solo tendremos que guardar un mapa de tiles, ya que las instancias correspondientes a un prefab, guardarán la misma representación física. Además, solo será necesario guardar la lista de puertas potenciales en el prefab, ya que también será común a todas las instancias.

El cómputo de las puertas potenciales también conlleva un coste, y gracias a este componente, se realizará una sola vez en la carga.

3.2.3. Instancias.

La instancia representa una concretización de un prefab. Una vez tenemos los modelos de habitación necesarios, se crearán las instancias pertinentes a partir de cada modelo. En estas instancias, incluiremos la información necesaria para situar las habitaciones en el mapa, así como para

poder manejar las conexiones entre puertas de las distintas instancias de habitaciones y poder realizar estimaciones sobre el mapa.

Para conectar dos instancias de habitación, necesitaremos una puerta por cada instancia, de forma que ambas puertas sean contiguas para que el jugador pueda cruzar de una habitación a otra. En la figura 3.6 se puede observar la definición formal de ésto.

$$p1(x1, y1) \in R1, p2(x2, y2) \in R2, \begin{cases} p1 = p2 + (0, 1) & \text{conexión horizontal} \\ p1 = p2 - (0, 1) & \text{conexión horizontal} \\ p1 = p2 + (1, 0) & \text{conexión vertical} \\ p1 = p2 - (1, 0) & \text{conexión vertical} \end{cases}$$

Figura 3.6: Conexión entre dos instancias de habitación

Así, guardaremos una lista de las puertas presentes en la instancia.

Otro dato importante a mantener en a instancia es una referencia al prefab, de forma que podamos realizar los cómputos necesarios que corresponden al modelo de habitación.

Por último, necesitaremos conocer la posición de la instancia en el mapa. Esta posición no se usará mientras la instancia no esté colocada en el mapa. Asimismo, se establecerá el valor de este dato, cuando coloquemos la instancia en el mapa.

Cabe destacar que a efectos lógicos, dos instancias de un mismo prefab son consideradas como habitaciones *distintas*. La utilización del prefab es solo una medida de conveniencia para evitar la duplicación de información.

3.3. Puertas.

Una vez elegido un par de tiles de puertas potenciales de dos instancias de habitación distintas que conectaremos, tenemos que añadir cierta información a las puertas, que será necesaria para el funcionamiento del sistema.

- *Instancia de Habitación propietaria* de la puerta
- *Puerta conectada* miembro de la instancia de habitación conectada.
- La *habitación conectada* se puede extraer a partir de la *puerta conectada*.
- *Posición local* relativa a la instancia de habitación a la que pertenece la puerta.
- *Tipo* de puerta. Se refiere a la orientación: horizontal o vertical.

Con toda esta información, podemos tratar las puertas como elementos independientes, ya que se tiene acceso a todos los componentes necesarios para cualquier cómputo o acción que desee realizarse sobre la conexión que representa esta puerta.

3.4. Mapa.

Por último, el modelo que usaremos para el mapa contendrá información física y lógica de las habitaciones y su disposición en el escenario.

El modelo empleado para el mapa, empleará un mapa de tiles donde se colocarán las habitaciones de forma física. El mapa de tiles nos servirá para comprobar solapamiento a la hora de colocar habitaciones para conectarlas. Además, una vez completo el mapa, tendremos el resultado final de la generación en este mapa de tiles.

El mapa de tiles empleado en el mapa, según las especificaciones, se supone lo suficientemente grande como para albergar cualquier distribución de habitaciones. Se ha elegido hacer uso de un mapa de tiles lo suficientemente grande para cumplir este requisito, pero podría haberse elaborado un mapa autoajustable, de forma que, en caso de ser necesario, se expandiría el tamaño.

Resultará práctico mantener las conexiones entre las instancias de habitaciones presentes en el mapa. Para ello, se ha hecho uso de una matriz superior, tal que en cada posición $(R1, R2)$, tendremos una estimación de la distancia entre las habitaciones $R1$ y $R2$. Consideraremos un valor muy alto como la ausencia de conexión. Esta matriz superior nos será de mucha utilidad a la hora de computar propiedades del mapa de forma rápida.

También será útil mantener una lista con el conjunto de todas las puertas potenciales de las habitaciones en el mapa. Se hablará en detalle de ésto en el próximo capítulo.

Capítulo 4

Estrategia constructiva.

Hasta ahora hemos visto la representación de las distintas entidades que entran en juego. En este capítulo introduciremos parte de la lógica que seguirá el sistema. Destacar que, aunque el enfoque final ha sido reducir el problema a una modificación de búsqueda, el sistema está pensado de forma que sea flexible a la hora de utilizar otro tipo de estrategias.

4.1. Algoritmo de generación

A la metodología que hemos utilizado la hemos denominado *estrategia constructiva*, cuyo origen radica en la forma de generar el mapa. Como se puede ver en el listado 4.1, el sistema construirá el mapa por pasos. de forma que en cada paso se hará una elección de una habitación y se colocará en el mapa. Así, el sistema dará por concluida la generación cuando no queden habitaciones en la lista inicial.

Listado 4.1: Algoritmo constructivo para generar mapas

```
1  Mapa GenerarMapa( List<Habitacion> habitaciones, InterfazSeleccion mapSolver
       ) {
2    Mapa mapa = MapaVacio();
3    while( !habitaciones.isEmpty() ) {
4      List<Movimiento> movimientos = GenerarMovimientos( mapa, habitaciones );
5      Movimiento elegido = mapSolver.ElegirMovimiento( mapa, movimientos );
6      mapa.InsertarHabitacion( elegido.habitacion, elegido.posicion );
7      habitaciones.remove( elegido.habitacion );
8      GuardarMovimiento( elegido );
9    }
10   EstablecerRecorridoPrincipal( mapa );
```

```
11    return mapa;
12  }
```

En el listado 4.1 se introducen dos conceptos nuevos: *movimientos* e *interfaz de selección de movimiento*, y ambos son clave para el entendimiento del funcionamiento del sistema.

A grandes rasgos, se elige un par $(habitacion, posicion)$ y se inserta en el mapa (línea 6). Posteriormente, se elimina la habitación de la lista de habitaciones, ya que se acaba de colocar (línea 7). Por último, guardamos el movimiento (línea 8). Más adelante veremos la utilidad de ésto.

Antes de devolver el mapa, establecemos el recorrido principal (línea 10), es decir, elegimos la habitacion inicial y final. Para ello, se ha empleado el algoritmo de *Floyd-Warshall*, ayudándonos de la *matriz superior de conexiones entre habitaciones* comentada en el capítuloX (INSERTARENLACE). El objetivo algoritmo *Floyd-Warshall* (APENDICEINSERTARENLACE) es, dado un grafo, establecer los caminos mínimos entre todos los pares de nodos posibles. Así, para establecer el recorrido principal, elegiremos el par de nodos cuya distancia mínima sea mayor.

4.2. Movimientos

Un movimiento $M_i(R_i, P_i)$ está constituido por:

- R_i: *instancia de habitación* a colocar en el mapa
- P_i: *posición* del mapa donde colocaremos dicha habitación

Así, en cada paso de colocación de una habitación, se generarán todos los movimientos posibles a realizar (listado 4.1, línea 4). La lista de movimientos posibles dependerá del estado del sistema, como veremos a continuación.

4.2.1. Cómputo de posibles movimientos.

Consideraremos que el *estado del sistema* depende de, y se ve afectado por:

- Instancias de habitación restantes en la lista inicial

- Modelo de instancias de habitaciones colocadas en el mapa

- Posición de las instancias de habitaciones colocadas

Como se observa en la figura 4.1, dado un estado del sistema $S(map, RR)$, la lista de posibles movimientos se computará a partir de las *posibles conexiones* entre las *puertas potenciales de las habitaciones colocadas en el mapa* y las *puertas potenciales de las habitaciones restantes*. El concepto de *puerta potencial* se discutió en el CAPITLOX (insertar enlace).

El cómputo de las posibles conexiones entre el mapa y una de las habitaciones restantes, se define en la figura 4.2. En la figura 4.3 se puede ver una ilustración de los tiles implicados. Destacar que el desplazamiento para testear la posible conexión entre una habitación de las restantes y una del mapa, solamente se puede realizar sobre la habitación restante, ya que las del mapa ya han sido colocadas y están fijas.

El cómputo de todos los posibles movimientos queda como la unión de los conjuntos que dan como resultado calcular las posibles conexiones entre todas las habitaciones restantes y las presentes en el mapa.

Así, el cómputo de todas las posibles conexiones entre una habitación en la lista de habitaciones restantes y una habitación del mapa consiste en calcular el conjunto de todas las posibles colocaciones de la habitación de la lista de restantes contigua a la habitación del mapa.

$$S(map, RR), RR = habitacionesRestantes$$

$$movimientos = \bigcup_{r \in RR} posiblesConexiones(map, r)$$

Figura 4.1: Cómputo de lista de movimientos dado un estado del sistema

$$posiblesConexiones(map, r) =$$

$$\bigcup_{p \in rooms(map)} \{cx(map, r, p, u, v, w) : \neg col(map, r, w)\}$$

$$w = v - u + \begin{cases} (-1, 0) \\ (+1, 0) \\ (0, -1) \\ (0, +1) \end{cases}$$

$$u \in r, v \in map, w \in map$$

$$puertaPotencial(u, r), puertaPotencial(v, p)$$

$col(map, room, t)$: colisión al colocar $room$ en map en el tile t

$cx(map, r, p, u, v, w)$: conectar $u \in r$ y $u \in p$ colocando r en tile $w \in map$

Figura 4.2: Cómputo de las posibles conexiones entre una habitación de las restantes y el mapa

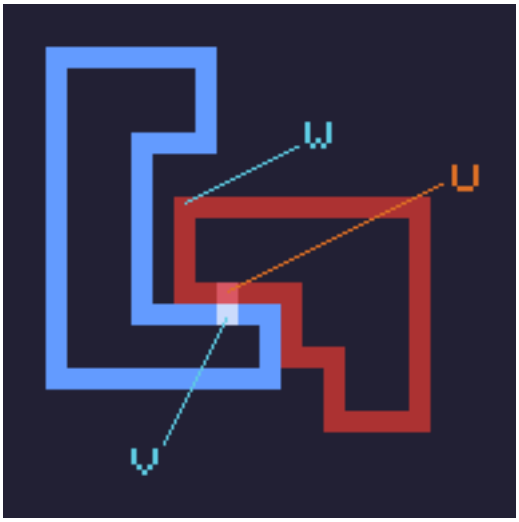

Figura 4.3: Ilustración de posible conexión entre habitación y mapa

4.3. Interfaz de selección de movimiento

Como se ha comentado anteriormente, la *interfaz de selección de movimiento* es otro elemento importante del sistema, y en ella radica parte de la flexibilidad del sistema. En el listado 4.2 se muestra el esqueleto de dicha interfaz, cuyo único método de interés es el relacionado con la elección del movimiento.

Listado 4.2: Interfaz de selección de movimiento

```
1 interface InterfazSeleccion {
2   Movimiento ElegirMovimiento( Mapa mapa, List<Movimiento> movimientos );
3 }
```

Una vez generados todos los movimientos posibles, se delega la decisión de elegir un movimiento a la *interfaz de selección de movimiento*. En esta interfaz radica uno de los puntos de flexibilidad del sistema. Mediante componentes que implementen esta interfaz, podemos idear otras formas de elegir un movimiento a partir de la lista de movimientos posibles.

En este proyecto se han creado dos interfaces de selección de movimiento:

- *Interfaz aleatoria*. Elige un movimiento de forma aleatoria de entre todos los posibles.
- *Interfaz basada en búsqueda*. Asigna una puntuación a cada movimiento y elige la mejor.

La interfaz aleatoria es simple y no tiene interés computacional. En el listado 4.3 se muestra la implementación de dicha interfaz por completitud, pero se ha realizado solo para testeo.

Listado 4.3: Implementación aleatoria de la interfaz de selección de movimiento

```
1 class RandomMovementSelector implements IMovementSelector {
2   Movimiento ElegirMovimiento( Mapa mapa, List<Movimiento> movimientos ) {
3     int indiceMovimiento = random( 0, movimientos.size() - 1);
4     Movimiento elegido = movimientos.get( indiceMovimiento );
5     return elegido;
6   }
7 }
```

Capítulo 5

Interfaz basada en búsqueda.

Abarcaremos en cierta profundidad la *interfaz de selección basada en búsqueda*. Se han omitido algunos detalles como la forma de crear y manejar las puertas, ya que se considera trivial y fuera del objetivo de la práctica. Además, algunos concepto como el concepto de *movimiento* no se traduce literalmente en el código, pero la idea esencial se conserva.

En el capítulo siguiente veremos la relación de este acercamiento con el método de búsqueda.

5.1. Estrategia basada en búsqueda

Como se observa en el listado 5.1, la forma de seleccionar el movimiento incluye el cálculo de una puntuación numérica que llamaremos *fitness*. Para obtener el *fitness*, aplicaremos una función que denominaremos como *función guía*, y devolverá un valor numérico indicando la calidad del movimiento.

Listado 5.1: Interfaz de selección de movimiento basada en búsqueda

```
1 class BestSearchMovementSelector implements IMovementSelector {
2   Movimiento ElegirMovimiento( Mapa mapa, List<Movimiento> movimientos ) {
3     List<Float> fitnesses;
4     for( Movimiento m : movimientos ) {
5       fitnesses[m.index] = FuncionGuia( m );
6     }
7     Movimiento elegido = ObtenerMejor( movimientos, fitnesses );
8     return elegido;
9   }
```

```
10 }
```

Aplicando la *función guía* a todos los movimientos, se computará el *fitness* para cada uno. Por último, elegiremos el movimiento cuyo *fitness* sea mejor. Convendremos que la calidad del *fitness* será directamente proporcional al mismo, es decir, cuanto mayor *fitness*, mejor movimiento.

5.2. Interfaz de cómputo de fitness

Para el cómputo del fitness, se ha empleado una interfaz que definimos en el listado 5.2, añadiendo flexibilidad a este componente. De esta forma, podemos también idear formas nuevas de computar el fitness y llevarlas a la práctica fácilmente.

Listado 5.2: Interfaz de cómputo de fitness

```
1 interface IFitnessSolver {
2   float FuncionGuia( Movimiento m );
3 }
```

En el listado 5.4 se ha sustituido la llamada a la función guía por la interfaz de cálculo de fitness. De esta forma, añadimos un componente de flexibilidad en esta parte, permitiendo la implementación de diferentes formas de calcular el fitness.

Listado 5.3: Interfaz de selección de movimiento basada en búsqueda

```
1  class BestSearchMovementSelector implements IMovementSelector {
2    IFitnessSolver fitnessSolver;
3    Movimiento ElegirMovimiento( Mapa mapa, List<Movimiento> movimientos ) {
4      List<Float> fitnesses;
5      for( Movimiento m : movimientos ) {
6        fitnesses[m.index] = fitnessSolver.FuncionGuia( mapa, m );
7      }
8      Movimiento elegido = ObtenerMejor( movimientos, fitnesses );
9      return elegido;
10   }
11 }
```

En las líneas 2 y 6 se observa el cambio sustancial que representa la interfaz de cálculo de fitness.

Se han elaborado dos implementaciones de la interfaz de cómputo de fitness. Una de ellas es sencilla, y define el fitness una puntuación numérica de *una sola propiedad del mapa*, en concreto, la distancia del camino principal (lo analizaremos más adelante (ENLACE A TAMAÑO CAMINO PPAL). Lo importante a destacar de esta implementación es que el fitness se define como un solo valor, mientras que en la segunda implementación veremos como computamos el fitness a partir de múltiples propiedades del mapa.

5.2.1. Cómputo de fitness múltiple

El propósito de esta implementación de la interfaz de cómputo de fitness es poder utilizar diferentes propiedades del mapa que influyan en la selección del movimiento a través del fitness. En el listado **??**, se ha empleado un número constante de 3 propiedades del mapa por utilidad, pero podría elaborarse para generalizar para aceptar un número cualquiera de propiedades.

Listado 5.4: Interfaz de selección de movimiento basada en búsqueda

```
1  class MultiFitnessSolver implements IFitnessSolver {
2    IFitnessCombinator fitnessCombinator;
3    float FuncionGuia( Mapa mapa, Movimiento m ) {
4      float fitnesses = new float[3];
5      fitnesses[0] = ComputarPropiedad0( mapa, m );
6      fitnesses[1] = ComputarPropiedad1( mapa, m );
7      fitnesses[2] = ComputarPropiedad2( mapa, m );
8      fitnessCombinator.Combinar( fitnesses );
9    }
10 }
```

Se observa en el listado **??** que se emplea otra interfaz para dar flexibilidad a la hora de combinar las múltiples propiedades. Se han elaborado dos tipos de combinadores de fitness:

- *Combinador parametrizado*, que computa el fitness final como una suma ponderada de todas las propiedades.

$$\sum_{f \in F} f * k(f)$$

$F =$ conjunto de propiedades, $k(f) =$ ponderación de la propiedad f

- *Combinador adaptativo parametrizado*. Al igual que el anterior, se computa como una suma ponderada de todas las propiedades. La diferencia es que el parámetro de ponderación es variable, de forma que la mejor propiedad del último movimiento elegido baja y el resto aumenta en dos factores que denominamos *attack* y *decay* respectivamente.

Con el combinador adaptativo parametrizado conseguimos una personalización más ajustada de la influencia de cada propiedad en el mapa. En el capítulo de experimentación (ENLACE A CAPI EXP) veremos como afecta a la generación.

5.3. Adaptación de las especificaciones del mapa al generador

En esta sección analizaremos las propiedades del mapa empleadas en el cómputo del fitness. Son totalmente independientes de la implementación de la interfaz de cómputo de fitness, ya que éstas solo establecen si se tiene en cuenta una o más propiedades.

5.3.1. Tamaño del camino principal

Para calcular el tamaño del camino principal dado un estado intermedio del sistema, emplearemos el estado del mapa sin tener en cuenta las habitaciones restantes. Como se comentó en (ENLACE A REPRESENTACION/uppermatrix), se mantiene una matriz de conexiones entre habitaciones con una estimación de la distancia entre las habitaciones conectadas. Actualizaremos la matriz cada vez que se coloque una habitación en el mapa.

Como se comentó en (ENLACE A FW ANTES), mediante esta matriz, podemos aplicar el algoritmo de *Floyd-Warshall* (apendice FW) para calcular la distancia mínima entre todos los pares de nodos. Elegiremos el par de nodos cuya distancia mínima sea máxima como habitaciones inicial y final, pudiendo computar así, una estimación del tamaño del camino principal.

5.3.2. Caminos no principales y bifurcaciones

En esta sección se analizan dos propiedades distintas en conjunto por estar relacionadas en la forma del cómputo de las mismas.

Si una habitación es un callejón sin salida, tendrá un solo enlace a la habitación que lleva hasta ella. Si se corresponde a un camino sin bifurcación, tendrá dos enlaces: uno hacia la habitación de la que viene el jugador, y otro a la que se tiene que dirigir por necesidad, al no haber otro enlace. Diremos en este caso que constituye un camino *sin* bifurcación. Atendiendo a todo esto, se puede considerar que si una habitación tiene más de dos enlaces, existen $n - 1$ caminos a los que se puede dirigir un jugador, sin contar la habitación de la que proviene, y por ello, estimamos que esta habitación contribuye a que existan bifurcaciones en el mapa.

Para establecer una estimación de *cuanto* tiene de caminos no-principales, se ha hecho un conteo de las habitaciones que no constituyen el camino principal. Debido a que en principio todas las bifurcaciones del mapa que no sean camino principal influirían a la estimación de caminos no-principales, se omite el conteo para la propiedad de *caminos no principales* cuando dicha habitación influye en la propiedad de *bifurcaciones*

Para la estimación de ambas propiedades, se ha empleado el algoritmo de *FloodFill* que se puede ver en el apéndice (ENLACE A APENDICE FLOODFILL). De esta forma, recorremos todas las habitaciones sin repetir ninguna y realizaremos el conteo de habitaciones que influyen en cada propiedad.

5.4. Eficiencia

Según las pruebas y el profiling, el cuello de botella de nuestra aplicación se encuentra en dos partes principalmente:

- Tiempo de cómputo del algoritmo Floyd-Warshall para el cálculo del tamaño del camino principal por cada movimiento
- Excesiva cantidad de posibles movimientos

Coste de Floyd-Warshall.

El algoritmo Floyd-Warshall se emplea para obtener el camíno mínimo entre todos los pares de nodos de un grafo. Su coste es de n^3 donde $n = |V|$ (número de vértices en el grafo). Si pensamos por un momento la cantidad de movimientos que puede darse en un estado intermedio de la generación, impacta de manera desorbitada.

Una primera solución fue usar el algoritmo de Dijkstra, con coste n^2 ($nlog(n)$ si usamos cola de prioridad). Para ello, no podemos tener en cuenta todos los nodos del grafo, ya que el algoritmo de Dijkstra calcula los caminos mínimos tomando un nodo de salida, y por ello se probó empleando como nodo de salida el que tuviera una distancia euclídea estimada mayor a cualquier otro nodo.

Esto daba resultados pésimos, ya que la puntuación del camino principal no es para nada real si lo comparamos con el cómputo que se hace en Floyd-Warshall. Es por ello que se ha decidido no recortar en este aspecto por la fiabilidad. Aún así, se adjunta una implementación del algoritmo de Dijkstra por si un usuario decidiera que no le importa tanto el fitness calculado para el camino principal.

Gran cantidad de posibles movimientos.

El segundo aspecto a tener en cuenta es el excesivo número de movimientos al que podemos llegar a enfrentarnos. Para ello, se han tomado varias estrategias que solventan este problema de manera considerable y no afectan prácticamente a la calidad de las soluciones. Aún así, para mejores tiempos, se puede modificar los parámetros.

5.4.1. Fitness caché

Experimentalmente, se ha comprobado que el *cuello de botella* de nuestra aplicación reside en el cómputo del fitness en su totalidad. Si recordamos las especificaciones (ENLACE A QUE TIENE QUE SER PARA MOVIL), el sistema tiene que funcionar en sistemas móviles. Para resolver este requisito, se ha creado una interfaz que nos ayudará a modular el tiempo de ejecución, sacrificando calidad en cuanto a las propiedades calculadas,

pero dando menores tiempos de ejecución.

Listado 5.5: Interfaz de selección de movimiento basada en búsqueda con mejora de caché

```
1  class BestSearchMovementSelector implements IMovementSelector {
2    IFitnessSolver fitnessSolver;
3    IFitnessCache fitnessCache;
4    Movimiento ElegirMovimiento( Mapa mapa, List<Movimiento> movimientos ) {
5      List<Float> fitnesses;
6      for( Movimiento m : movimientos ) {
7        Fitness f = fitnessCache.Get(m);
8        if( f != null ) {
9            fitnesses[m.index] = fitnessSolver.FuncionGuia( mapa, m );
10           fitnessCache.Cachear( m, fitnesses[m.index] );
11         } else {
12           fitnesses[m.index] = f;
13       }
14     }
15   }
16 }
```

En el listado 5.5 vemos las modificaciones realizadas para permitir el empleo de la caché. Ahora, antes de realizar el cómputo del fitness, comprobamos mediante la interfaz si está el cálculo de dicho movimiento está cacheado. En caso positivo, se usa el valor cacheado, y en caso negativo, se calcula el fitness y se guarda en la caché.

Como se ha mencionado antes, esto conlleva a una peor fiabilidad del valor asociado al movimiento, ya que no se tiene en cuenta actualizaciones del mapa, pero podemos crear cachés nuevas que cada cierto número de pasos haga un recómputo de todos los movimientos en caché. Veremos más posibilidades en el capítulo de trabajo futuro (ENLACE A MEJORAS DE FITNESS CACHE).

Así, se han realizado dos implementaciones de la caché:

- *Dummy*. No cachea nada. El sistema se comporta como si no hubiera caché.
- *Always*. Cachea siempre y devuelve el valor cacheado.
- *Refresher*. Vacía los movimientos cacheados cada N habitaciones insertadas.

En el capítulo de trabajo futuro (ENLACE A MEJORAS DE FITNESS CACHÉ) estableceremos las pautas para crear cachés modulables con una implementación rápida.

5.4.2. Prefab Manager

Como se ha mencionado varias veces, cada habitación a colocar o colocada en el mapa, es una instancia de un modelo de habitación. Por ello, el PrefabManager se encarga de evitar que se comprueben los movimientos teniendo en cuenta todas las instancias, sino los modelos. De esta forma, ahorramos movimientos con una capacidad informativa idéntica.

5.4.3. Puertas potenciales

Las puertas potenciales afectan al número de movimientos posibles. Si reducimos este número, podemos agilizar de manera importante el proceso. Por ejemplo, si tenemos habitaciones muy grandes con muchas puertas potenciales, pero solo tomamos 3 de este subconjunto, no habrá que comprobar todos en cada paso del algoritmo.

Para ello, se han planteado tres estrategias de selección de puertas, de las cuales se han implementado dos:

- Generación aleatoria de un subconjunto de puertas
- Generación de 1 de cada N puertas
- Selección de puertas potenciales manual

La selección de puertas manual no se ha implementado, pero sería directo hacerlo indicando las puertas desde el editor de habitaciones.

La generación aleatoria de un subconjunto de puertas según un factor en el rango $[0, 1]$, que indica el porcentaje del total de puertas potenciales de una instancia de habitación que se eligirán de forma aleatoria.

La generación de 1 de cada N puertas, que se ha llamado *divisor*, emplea un factor en el rango $[0, 1]$, que indica el porcentaje de puertas que se

descartarán. Es decir, si el factor es 0, se elegirán todas las puertas. Si el factor es 1, se eligirá solamente 1 puerta de todas las posibles.

5.4.4. Divisor de movimientos

Parámetro en el rango $[0, 1]$ que indica el porcentaje de movimientos que se tendrán en cuenta en cada paso. Debido a que se baraja aleatoriamente la lista de posibles movimientos, este parámetro permite una calidad de las soluciones bastante buena, no afectándola demasiado.

Capítulo 6

Correspondencia con un problema de búsqueda.

Hasta ahora hemos visto los elementos y el procedimiento del sistema para generar una solución válida. En este capítulo, analizaremos los elementos de un problema de búsqueda y los identificaremos en el sistema propuesto. Destacar que este trabajo, se ha necesitado un componente de aleatoriedad, por lo que el sistema construido no es totalmente fiel al método de búsqueda como tal, pero sí se basa de forma coherente en el mismo.

6.1. Método de búsqueda

Cuando se nos propone un problema en el que podemos conocer el *espacio de soluciones*, se puede aplicar un método de búsqueda para la obtención de una de ellas que se ajuste al enunciado del problema. Así, el problema de búsqueda se puede constatar como encontrar una solución o estado concreto entre un conjunto de soluciones (el llamado *espacio de estados*) que satisfaga el enunciado del problema.

Para transitar entre estados, y realizar la búsqueda de una solución adecuada, definiremos las *acciones sobre estados*. De esta forma, realizaremos dichas acciones sobre los estados para obtener nuevos estados definidos en el *espacio de estados*, y así poder indagar sobre el mismo para encontrar una solución factible para nuestro enunciado.

Así, analizaremos el problema de búsqueda desde el punto de vista de *agentes* que realizan dicha búsqueda para encontrar una solución factible. En nuestro caso, podemos identificar el *agente* como el sistema de generación que se ha elaborado.

6.2. Modelización del problema

El método de búsqueda se aplica a la resolución de problemas para los que existe un *espacio de estados*. Para resolver el problema, se emplea un modelo del mismo, generalmente para simplificar y eliminar complejidad innecesaria. A partir de este modelo, se elaborará una estructura de datos que será la base para la resolución del modelo. Según la adecuación del modelo al problema real, así se adecuará la solución que encontremos, ya que la misma será una solución al modelo del problema, y no al problema en sí. Una elección de un modelo inadecuado, puede resultar en soluciones ineficientes.

En nuestro caso, se comentó la estructura de datos para la modelización del problema en el capítulo X (ENLACE A CAPITULO DE REPR. DEL MAPA). A partir de esta representación, se elaboró un sistema que iteraba sobre ella para ir construyendo una solución (ENLACE A CAPITULO DE ESTRATEGIA CONSTRUCTIVA), y es en la interfaz de construcción donde reside el resto de la similitud del sistema con una búsqueda.

6.3. Búsqueda informada

En este tipo de métodos, podemos encontrar una clasificación que divide entre *búsqueda informada* y *búsqueda no informada*. La *búsqueda no informada* parte de que no se conoce información acerca del entorno en el que se realiza la búsqueda.

Por contrapartida, la *búsqueda informada* se constata cuando sí podemos conocer en alguna medida el entorno en el que buscamos nuestra solución. La adecuación de la solución al enunciado del problema, dependerá del objetivo que se inste en el enunciado. Así, la búsqueda procurará encontrar una solución tomando este objetivo como baza para la obtención de la misma. Estos objetivos son los que denominaremos *bondad*, *fitness* o *heurística*

de la solución, y son los que nos guiarán en la búsqueda para la obtención de una solución adecuada al enunciado.

En este capítulo trataremos el método de *búsqueda informada*, debido a que en nuestro problema, partimos de una serie de objetivos bien definidos y computables sobre los estados que componen el espacio de estados.

6.4. Problemas resolubles mediante búsqueda

Para que un problema sea abarcable por un método de búsqueda necesitamos que cumplan varias propiedades. Dichas propiedades, definiran de forma concisa un entorno en el que podamos utilizar el método, que requieren que el problema esté bien definido según las mismas.

El problema debe ser *discreto*, es decir, el espacio de estados debe estar *bien definido* y ser *finito*. Además, esto conlleva que el número de *acciones* posibles a realizar en un estado para la obtención de estados subyacentes, sea también *finito*.

El problema debe ser *accesible*. Esto es, el agente debe poder determinar en que estado se encuentra, así como el estado que debe alcanzar según los objetivos definidos en el enunciado del problema.

El problema debe ser *estático*. Para ello, no podemos introducir variables que cambien en el tiempo, de forma que un estado tendrá las mismas propiedades independientemente del momento de la búsqueda en el que lo visitemos.

El problema debe ser *observable*. El agente debe conocer las variables que influyen en el modelo del problema en todo momento para poder evaluar cada estado.

El problema debe ser *determinista*. El estado solo cambiará a consecuencia de las *acciones* permitidas que ejecute el agente, y cada estado consecutivo a un estado previo debe estar determinado solamente por el estado en el que se encuentra y la acción realizada.

6.5. Elementos de un problema de búsqueda

Para poder efectuar la metodología de búsqueda en el objetivo de resolver un problema, es necesario tener presentes diversos elementos que ya hemos nombrado previamente Pasaremos a definir e identificar en nuestro sistema elaborado para la resolución del problema que nos atañe:

- *Modelo de solución del problema*. En nuestro caso, el modelo del problema a emplear será el mapa sobre el que iremos colocando habitaciones, así como el conjunto de habitaciones restantes por insertar en el mapa.

- *Espacio de estados*. Definiremos como espacio de estados, el conjunto de posibilidades de colocación de habitaciones de la lista inicial en el mapa, ya sea el conjunto completo de las mismas o parte.

- *Estado inicial*. El estado del que partiremos, en principio será un mapa vacío, aunque como veremos más adelante, se podrá partir de un mapa previamente construido.

- *Función sucesor* o *acciones*. Las acciones que puede realizar nuestro agente es lo que anteriormente hemos llamado *movimientos*, y consiste en la colocación de una habitación en una posición concreta del mapa.

- *Objetivo*. En nuestro sistema, el concepto de objetivo tiene dos componentes:

    1. Objetivo de finalizado del algoritmo. Este objetivo se cumplirá cuando no existan más habitaciones que introducir en el mapa.
    2. Propiedades requeridas de la solución deseada. Este componente tiene cierto aspecto *multimodal* que comentaremos en la siguiente sección, y se refiere a la función guía que influye en la forma de colocar las habitaciones para cumplir los objetivos de maximizar el camino principal y fomentar caminos alternativos

Como vemos, se pueden identificar todos los elementos de un problema de búsqueda en el sistema que se ha elaborado.

6.6. Multimodalidad

Normalmente, el elemento multimodal suele añadir una complejidad extra al sistema. En nuestro caso, se ha procurado dejar este elemento lo suficientemente flexible como para poder añadir tanto nuevos objetivos, como para redefinir la forma en la que se relacionan los mismos gracias a la interfaz IFitnessCombinator (ENLACE).

Como se comentó anteriormente (ENLACE A 5.2.1), nuestro sistema soporta la adición de diferentes objetivos para la guía en la búsqueda de una solución. Esto añade un componente multiobjetivo al problema que se ha definido de forma flexible para fomentar la afinación de este aspecto, como por ejemplo el cómputo de fitness adaptativo parametrizado.

Además, en nuestro problema, el componente multiobjetivo *no es una restricción fuerte*, sino una guía para como se debe construir la solución de forma que cumpla el enunciado.

6.7. Posibilidad de backtracking

El árbol de búsqueda es otro elemento importante a considerar en este tipo de problemas. Es un grafo de tipo árbol que se irá construyendo de forma implícita, donde el nodo raíz es el estado inicial, las aristas representan las distintas acciones que realizamos sobre dicho estado inicial, desembocando en nodos que representan el estado tal cual se ha dejado después de aplicar dicha acción.

En nuestro caso, el árbol de búsqueda es bastante simple (figura 6.1), ya que no existe backtracking, sino que confiamos en que cada movimiento (acción) elegido será bueno gracias a la función guía. Así, el componente *constructivo* del que hablamos previamente (ENLACE), podemos identificarlo en la búsqueda como que siempre se decide en cada paso y no hay vuelta a posibles estados anteriores.

Esto es así porque un componente de backtracking podría encarecer el algoritmo en cuanto a tiempo de ejecución, además de que podemos decir que trabajamos en el sistema siempre con el mismo mapa de tiles para poder comprobar las colisiones entre los mapas.

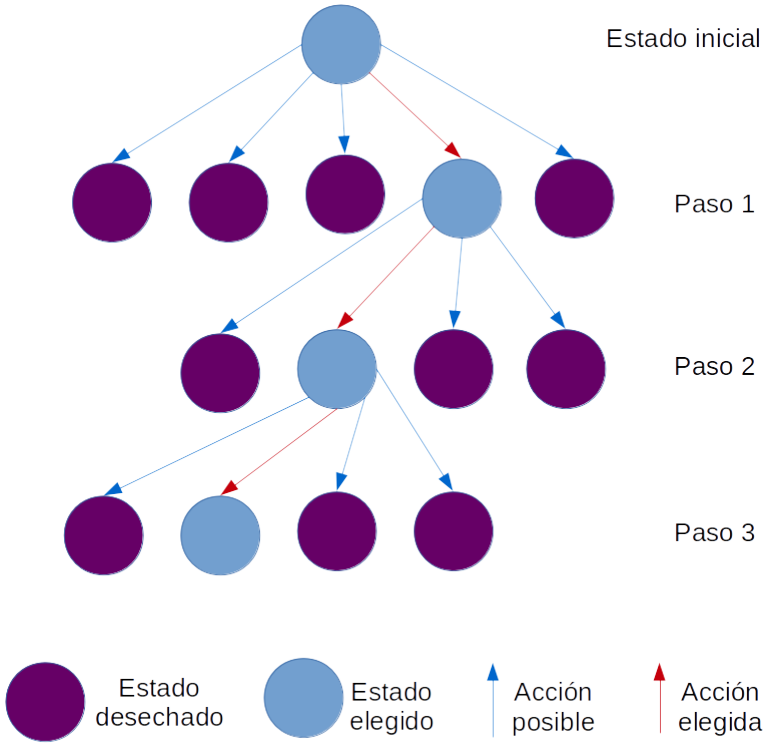

Figura 6.1: Árbol de búsqueda para el método planteado

Aún así, como se comentó anteriormente (ENLACE A GUARDADO DE ESTADOS), en cada paso del algoritmo se guarda el movimiento que se realiza. Con ésto, se ha implementado la forma de reconstruir un mapa a partir de una serie de movimientos consecutivos. Con esto, comentaremos en el capítulo de trabajo futuro (ENLACE A CAPITULO), una forma factible de poder realizar backtracking.

Para la elaboración de este proyecto no se vió necesario, ya que los resultados que se obtienen con el sistema tal cual está, son bastante buenos y adecuados al enunciado, además de que añadiría un componente de complejidad extra.

Capítulo 7

Experimentación.

En este capítulo veremos ejemplos demostrando tanto datos de tiempo como muestras de algunos ejemplos realizados. Para los ejemplos de mapas se han empleado solamente dos habitaciones.

7.1. Flexibilidad y posibilidades.

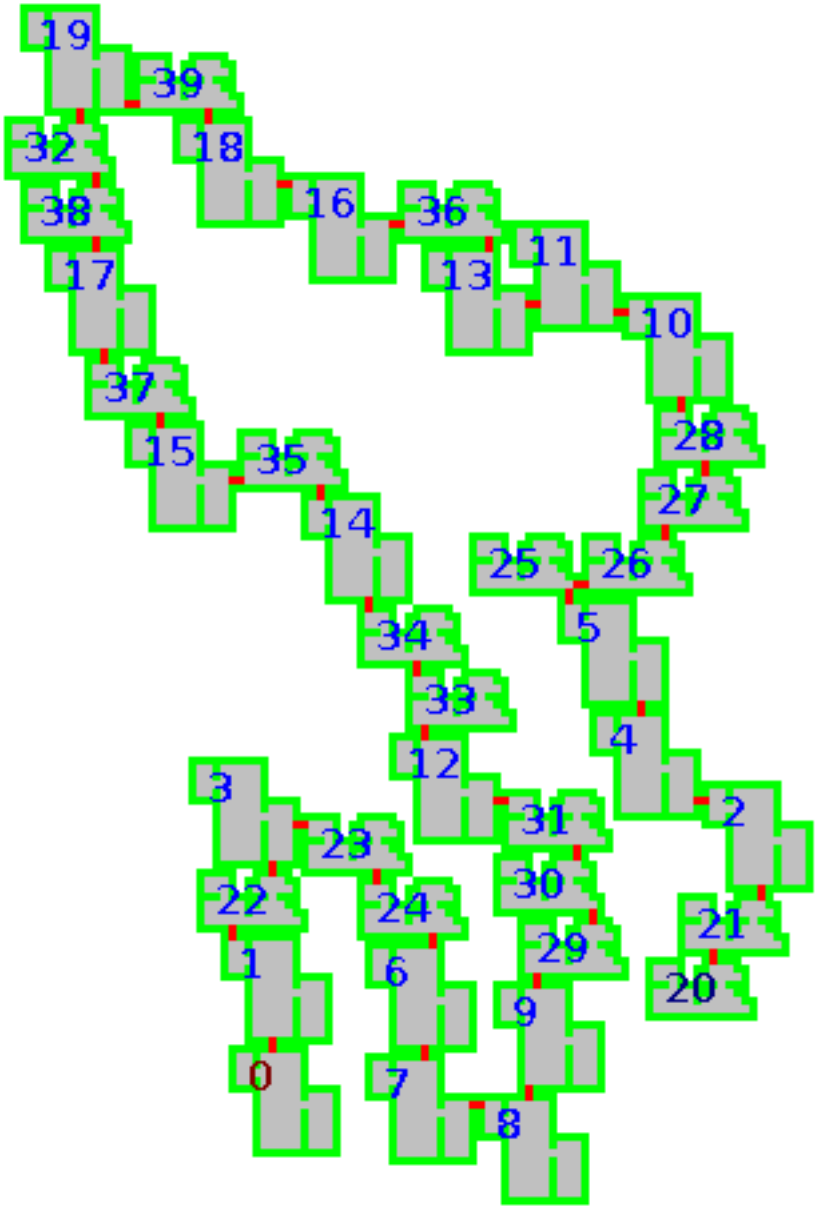

Figura 7.1: Combinador parametrizado. Parámetros: camino principal (100), resto (0)

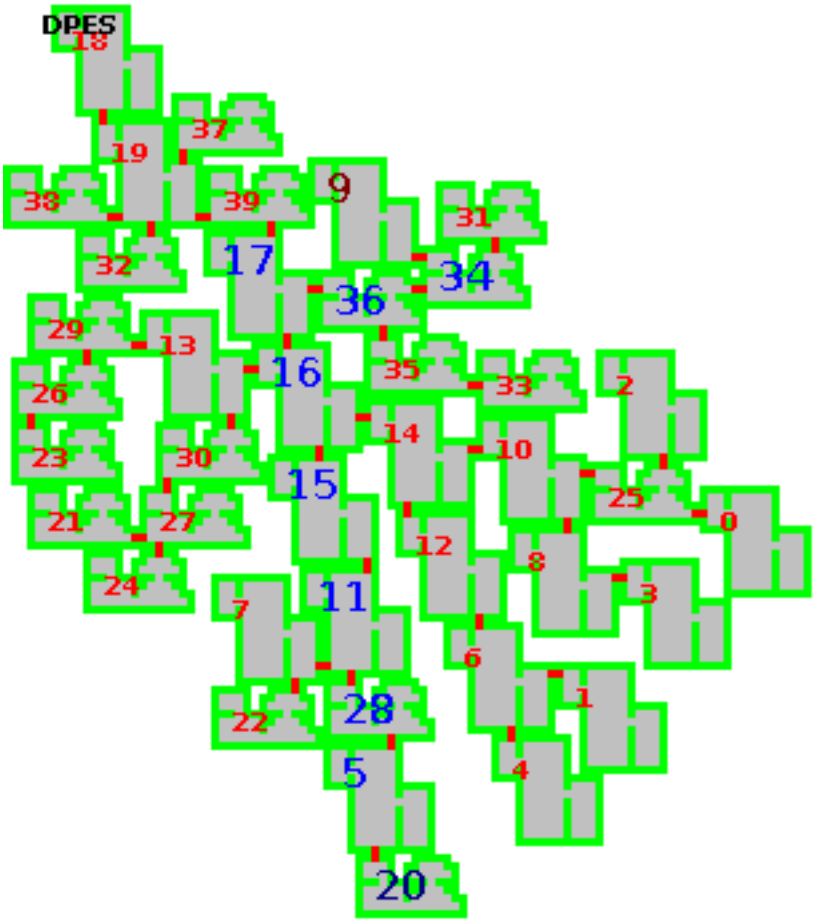

Figura 7.2: Combinador parametrizado. Parámetros: camino alternativo (100), resto (0)

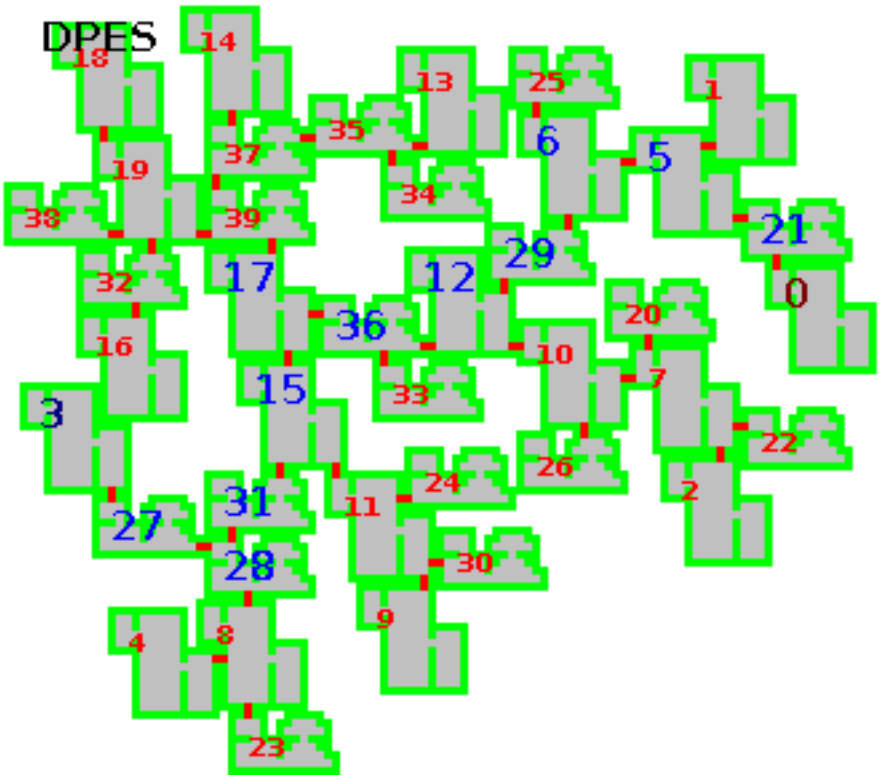

Figura 7.3: Combinador parametrizado. Parámetros: branching (100), resto (0)

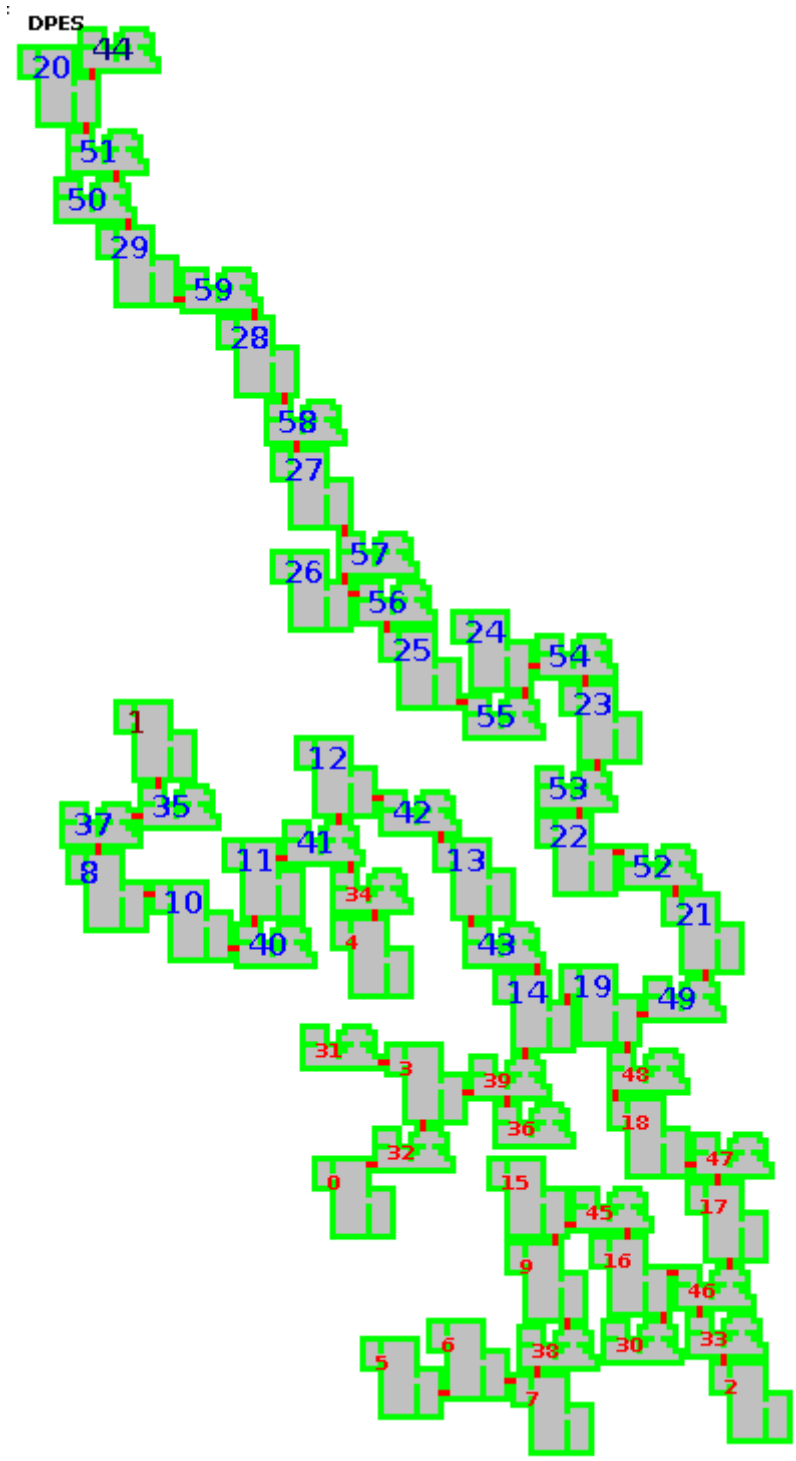

Figura 7.4: Combinador parametrizado. Parámetros: camino principal (5.8), camino alternativo (1), branching (0)

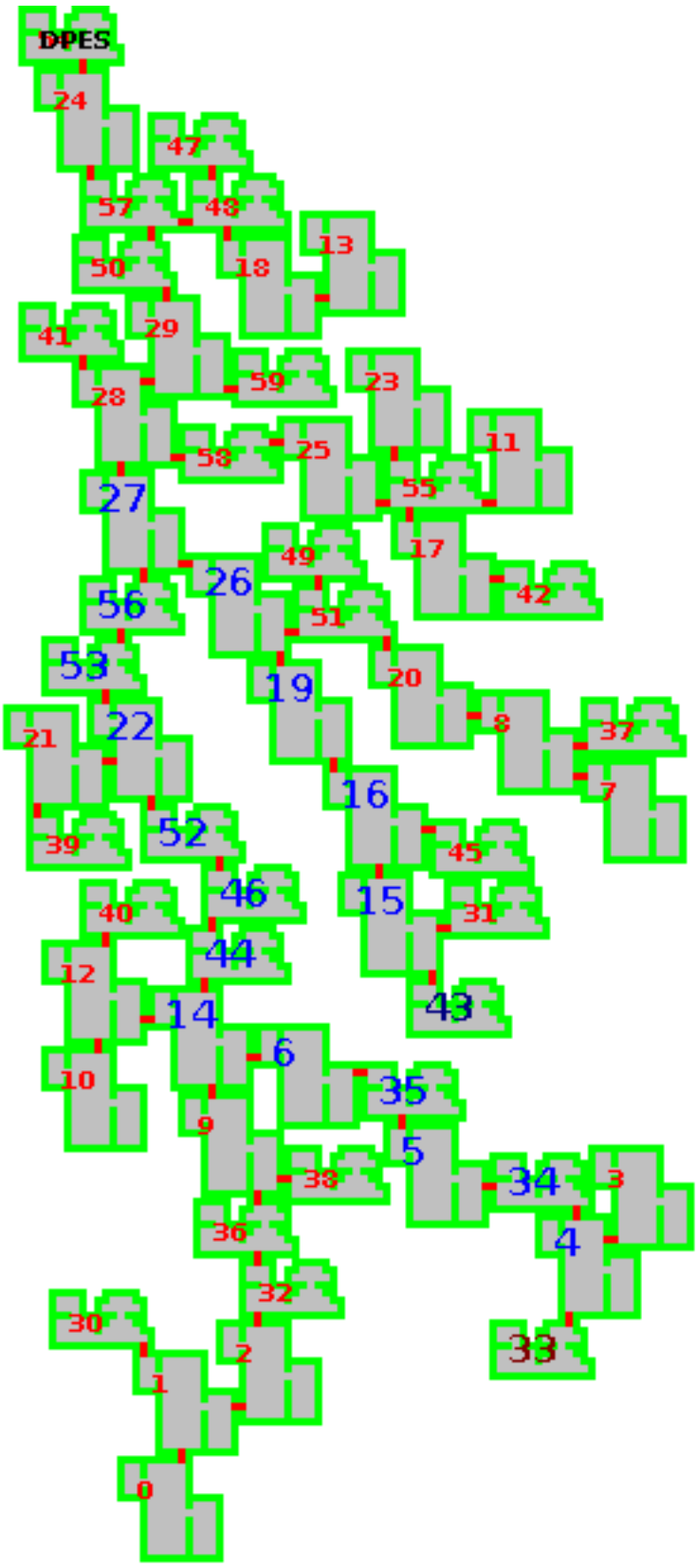

Figura 7.5: Combinador parametrizado. Parámetros: camino principal (5.8), camino alternativo (1), branching (1)

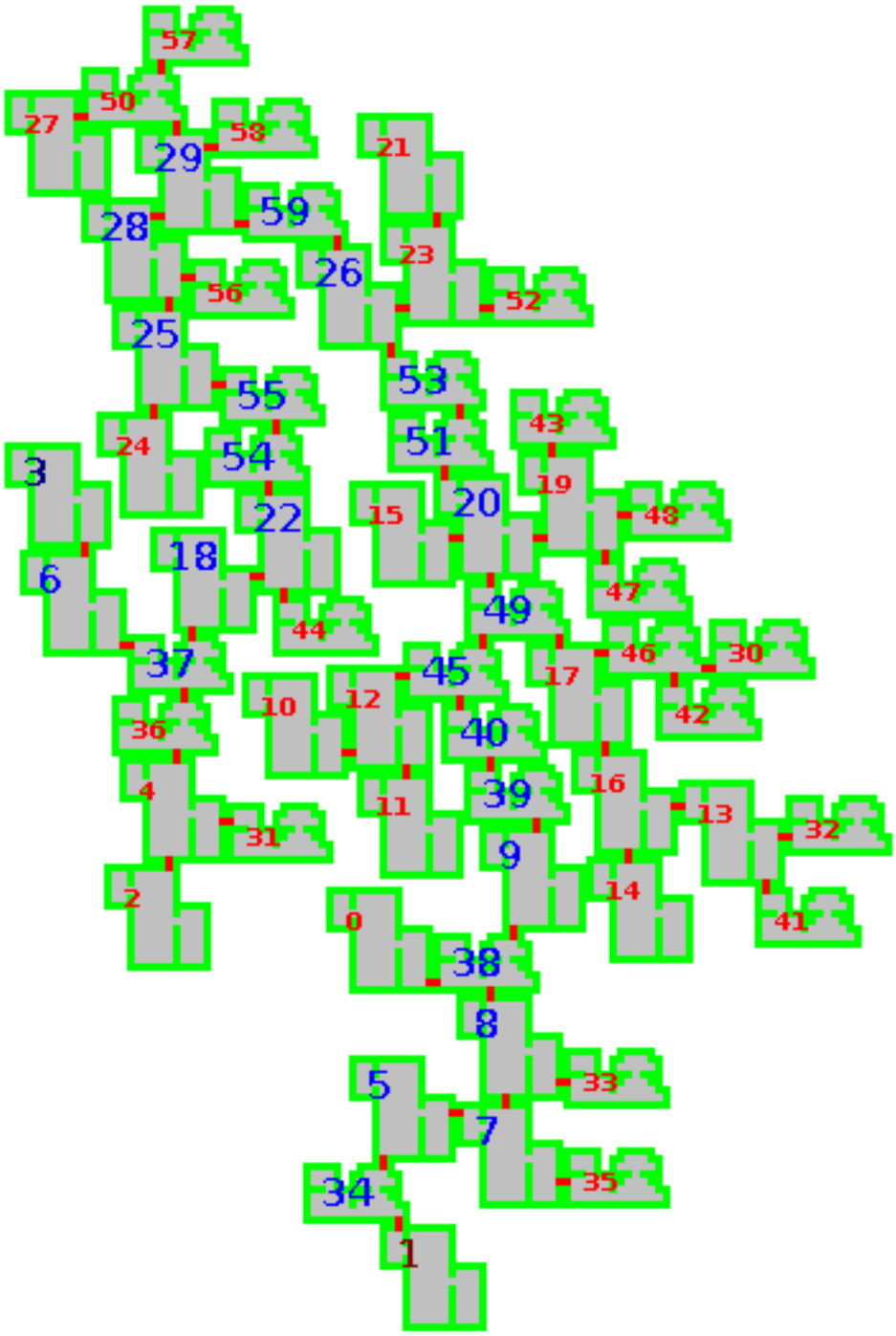

Figura 7.6: Combinador adaptativo. Attack/Decay (1/0.5). Parámetros: camino principal (10), camino alternativo (1), branching (1)

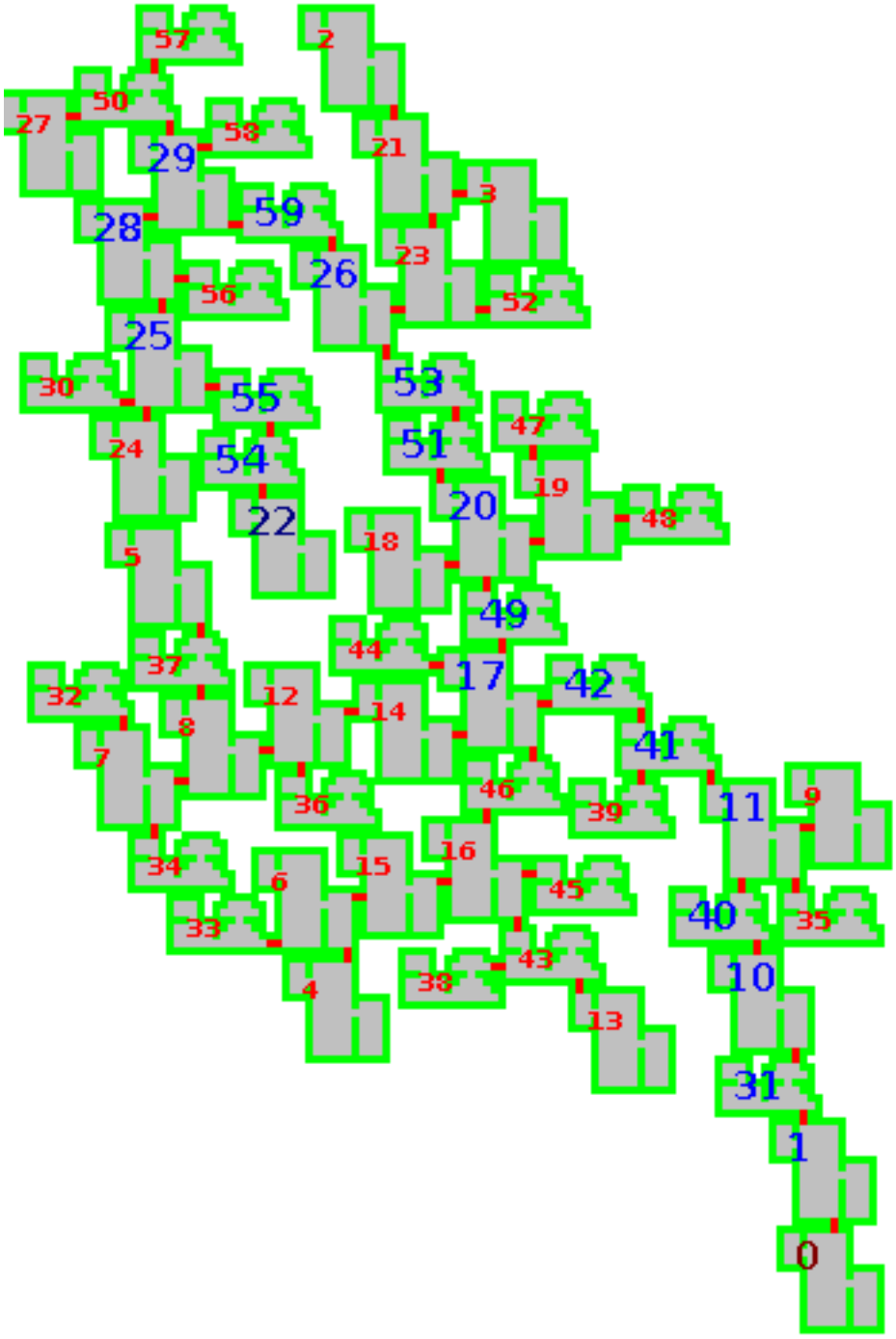

Figura 7.7: Combinador adaptativo. Attack/Decay (1/0.5). Parámetros: camino principal (10), camino alternativo (10), branching (10)

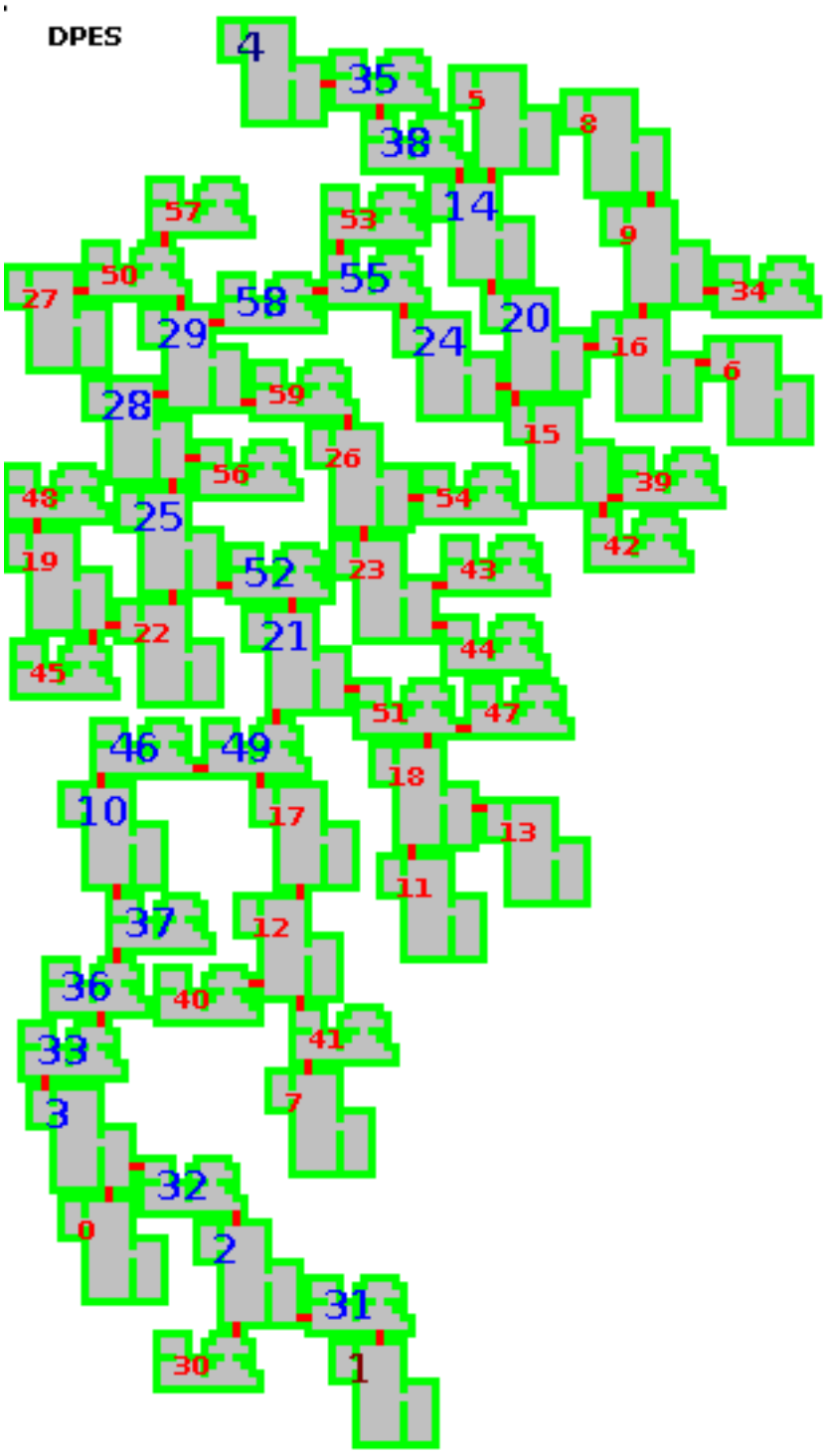

Figura 7.8: Combinador adaptativo. Attack/Decay (1.1/0.9). Parámetros: camino principal (10), camino alternativo (10), branching (10)

7.2. Eficiencia

Recordemos que, aunque no se restringía un tamaño al escenario, se estima como tamaño máximo un escenario de 64x64 tiles. Aún así, para estresar el algoritmo, se han elaborado mapas con tamaños mayores. Por ejemplo, en uno de ellos el tamaño de habitación de 20x20, y en el mejor caso, con habitaciones de este tamaño cuadradas, podríamos tener 9 en un mapa de 64x64, pero veremos tiempos bastante buenos con una generación de este tipo hasta para escenarios de 30 habitaciones.

Como no se ha implementado en móvil, se ha considerado que un tiempo es bueno, cuando es menor de un segundo. Si es menor de dos segundos, también se considerará como aceptable. Aún así, habría que hacer una prueba real, que debido a que Java es la plataforma de elección para el desarrollo a Android, no sería complejo. En un caso real de un juego de móvil además, se presupone menor complejidad debido al sistema donde se ejecuta, y se puede entender que un tiempo aceptable de espera a la generación es hasta 10 segundos, y en esto se fundamenta la aceptación de menor de dos segundos como solución buena.

Destacar que las mediciones realizadas para comprobar el impacto de algunos parámetros se han usado sin tener en cuenta los demás. Al final, podremos ver una medición realizada sobre una configuración que se ha conseguido con tiempos muy buenos y calidad notable.

Las pruebas se han realizado en un portátil HP Compaq Presario V6000 con las siguientes características:

Modelo procesador	AMD Athlon 64 X2
Núcleos procesador	2 (1 en uso)
Velocidad procesador	1.7 Ghz
Caché procesador	512 KB L2
Memoria RAM	1 GB

7.2.1. Impacto del factor divisor de movimientos

Como se comentó anteriormente (ENLACE A ESTO), el factor de divisor de movimientos permite elegir un subconjunto de todos los movimientos posibles para cada paso de la generación.

Se han realizado pruebas con la configuración para el resto de parámetros como se indica en la tabla 7.1, modificando el parámetro de divisor de movimientos para los valores 0.75, 0.85 y 0.95.

En la figura 7.9 se puede apreciar la influencia de este parámetro en los tiempos de generación. Normalmente, suelen dar mapas de buena calidad para los parámetros dados.

Ha de tenerse en cuenta que cuanto mayor son los posibles movimientos, menos influye el parámetro en la calidad de los mapas generados. Si aumentamos el tamaño de las habitaciones, podremos con seguridad aumentar el parámetro, ya que habrán muchos posibles movimientos asociados a cada habitación.

Property	Value
DoorGenType	ALL
SolverType	BestSearch
CacheType	NO CACHE

Cuadro 7.1: Configuración para divisor de movimientos

Tam. habs.	Modelos	Inst./modelo	Total habs.	0.75f	0.85f	0.95f
10	2	20	40	0.1763	0.1967	0.3667
10	2	30	60	0.4945	0.6254	1.4392
10	4	20	80	1.2336	1.6429	4.0735
10	4	30	120	4.8805	6.6694	18.1853

Cuadro 7.2: Influencia del divisor de movimientos

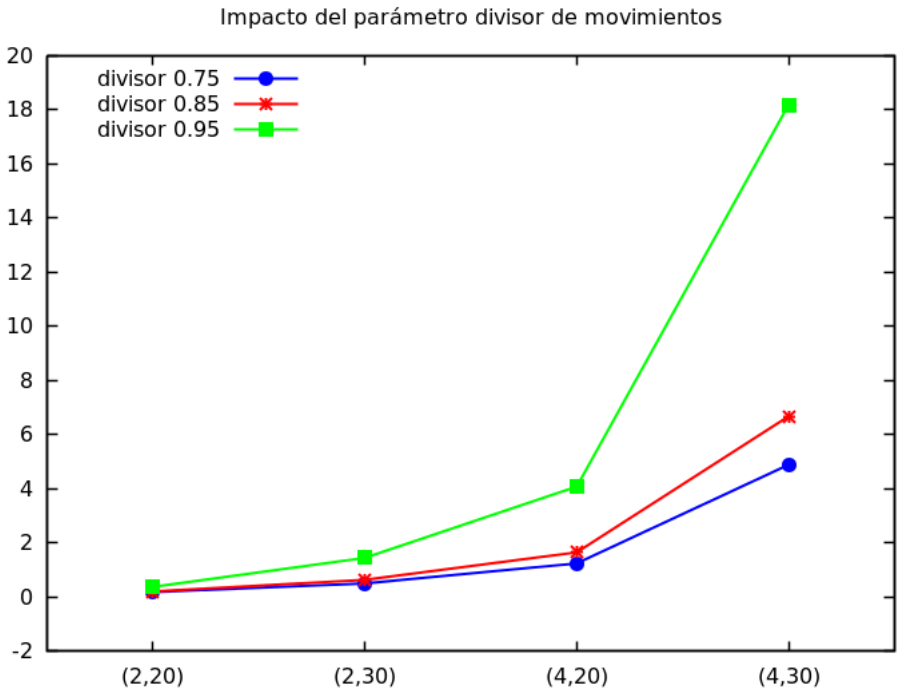

Figura 7.9: Gráfica de comparación de parámetro divisor de movimientos

7.2.2. Impacto de la generación de puertas aleatoria

Gracias a este parámetro, podemos reducir el número de puertas potenciales de cada habitación. Así, se ha probado con un valor de 0.4, 0.6 y 0.8 para este aspecto. Cualquier valor es bueno, si tenemos en cuenta que el divisor de movimientos reducirá los posibles movimientos. Es decir, si elegimos una sola puerta potencial de todas las habitaciones y damos un valor muy alto del divisor de movimientos, obtendremos mapas de baja calidad, ya que estaremos limitando demasiado la exploración de posibilidades.

Property	Value
DoorGenType	RANDOM
SolverType	BestSearch
BestSearch DPE Divisor	1.0
CacheType	NO CACHE

Cuadro 7.3: Configuración del test de impacto de generación de puertas aleatorias

Tam. habs.	Modelos	Inst./modelo	Total	0.8f	0.6f	0.4f
10	4	8	32	4.6552	3.3017	1.7974
10	4	10	40	9.8173	8.2366	4.1857
10	6	8	48	29.2931	20.9267	15.4530
10	6	10	60	69.2747	57.3447	34.6857

Cuadro 7.4: Test de impacto de generación de puertas aleatorias

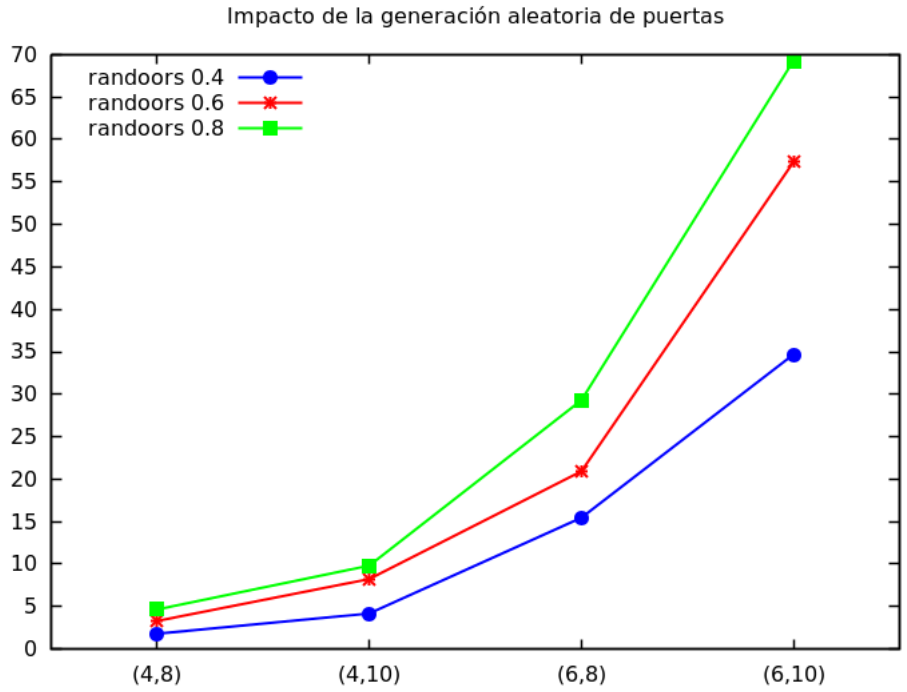

Figura 7.10: Gráfica de comparación de parámetro de generación de puertas aleatorias

7.2.3. Impacto del Caché Refresher

Este parámetro trabaja muy bien con la generación. En principio, si empleamos una caché que cachea siempre, obtenemos generaciones mediocres, sin llegar a ser excesivamente malas. Añadiendo este componente, podemos aumentar la calidad de los mapas de forma considerable y obteniendo tiempos bastante buenos.

En la figura 7.11 se puede ver como se mejoran los tiempos empleando este parámetro.

Property	Value
DoorGenType	ALL
SolverType	BestSearch
BestSearch DPE Divisor	1.0
CacheType	REFRESHER

Cuadro 7.5: Configuración de test de impacto de caché refresher

Tam. habs.	Modelos	Inst./modelo	Total	N=2	N=5	N=10
6	4	10	40	1.3768	1.0284	0.7653
6	4	15	60	6.7213	5.5230	3.4248
6	6	10	60	8.7300	6.7349	5.5666
6	6	15	90	43.7862	35.1948	26.3091

Cuadro 7.6: Test de impacto de caché refresher

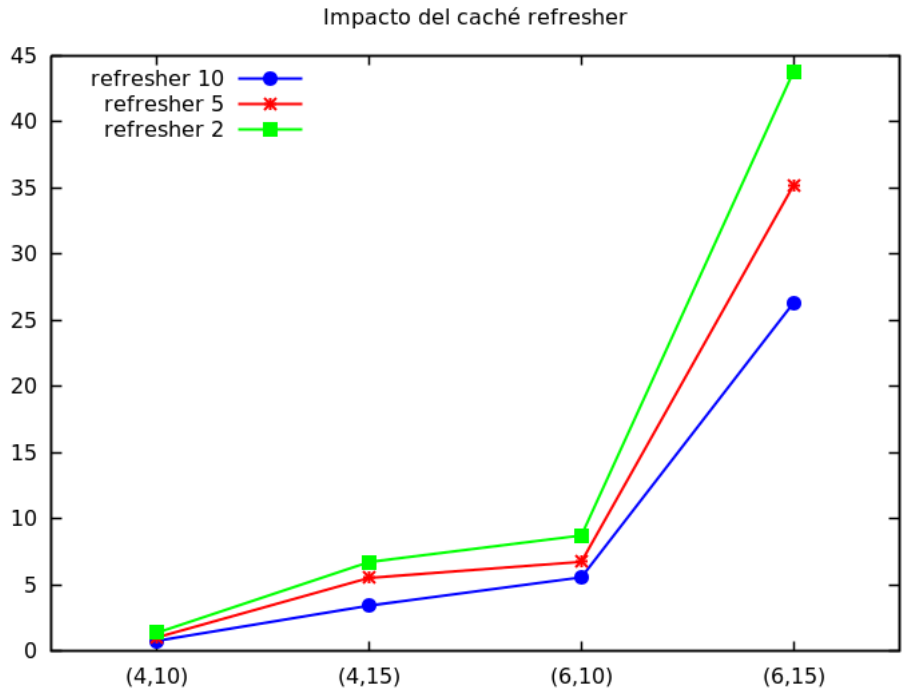

Figura 7.11: Gráfica de comparación de parámetro de caché refresher

7.2.4. Ejemplo real optimizado

Se ha elaborado una configuración optimizada para ejemplos que consideraremos reales de habitaciones desde 6x6 hasta 10x10, teniendo en cuenta que el tamaño del mapa puede rondar los 64x64 tiles. La configuración puede verse en la tabla 7.7.

En la figura **??** puede verse la evolución de los tiempos conforme aumentamos el número de modelos y de instancias por modelo para un tamaño de habitaciones de 6x6, y en la figura **??** para un tamaño de 10x10. Se observa como se obtienen tiempos muy buenos para ambos casos.

Property	Value
DoorGenType	RANDOM
Refresher cache divisor	10
SolverType	BestSearch
BestSearch DPE Divisor	0.9
CacheType	REFRESHER
Random doors param	0.5

Cuadro 7.7: Configuración para test real en dispositivos móviles

Tam. habs.	Modelos	Instancias/modelo	Total habs.	Tiempo
6	5	1	5	0.0007
6	5	2	10	0.0029
6	5	3	15	0.0073
6	5	4	20	0.0168
6	10	1	10	0.0050
6	10	2	20	0.0266
6	10	3	30	0.0679
6	10	4	40	0.1601
6	15	1	15	0.0157
6	15	2	30	0.0905
6	15	3	45	0.3026
6	15	4	60	0.6859
6	20	1	20	0.0398
6	20	2	40	0.2525
6	20	3	60	0.7766
6	20	4	80	2.0803

Cuadro 7.8: Test real en dispositivos móviles

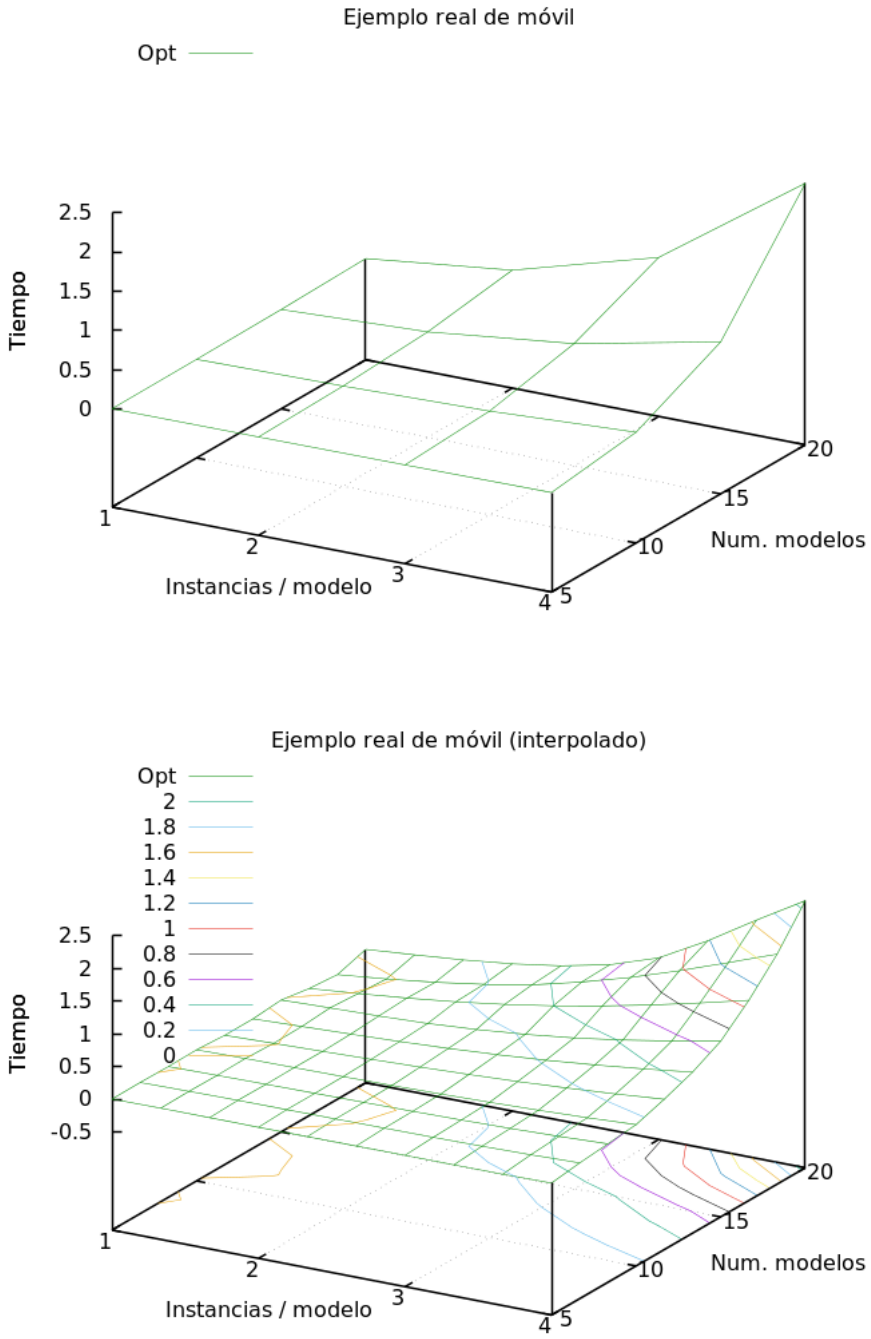

Figura 7.12: Gráfica de ejemplo óptimo real en móviles

Tam. habs.	Modelos	Instancias/modelo	Total habs.	Tiempo
10	5	1	5	0.0035
10	5	2	10	0.0127
10	5	3	15	0.0246
10	10	1	10	0.0257
10	10	2	20	0.1216
10	10	3	30	0.1667
10	15	1	15	0.1143
10	15	2	30	0.3776
10	15	3	45	0.8163

Cuadro 7.9: Test optimizado de tamaño medio

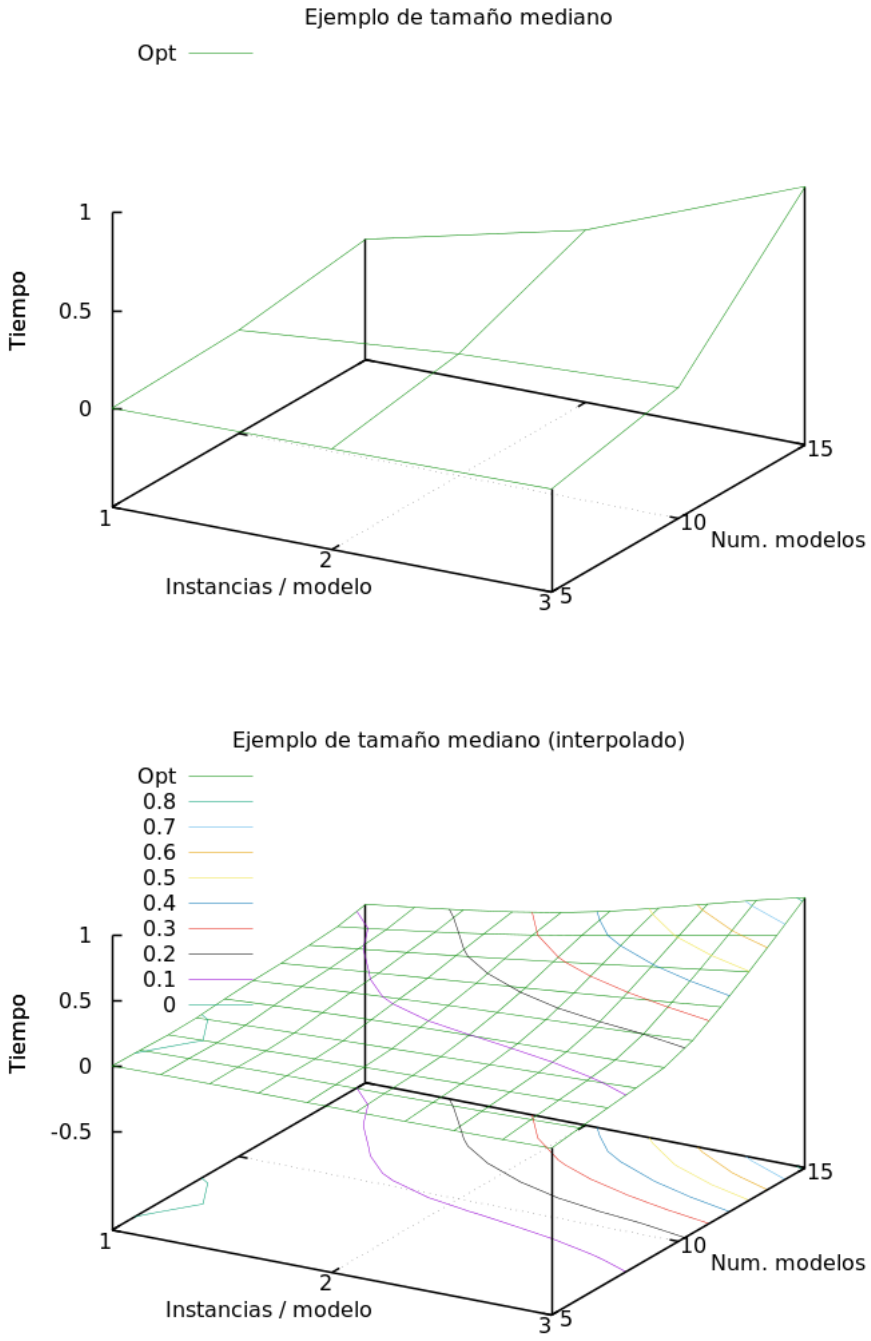

Figura 7.13: Gráfica de ejemplo óptimo de tamaño medio

7.2.5. Comparación ejemplo optimizado

Si comparamos la configuración optimizada con ejemplos sin caché y cacheando siempre (pero no modificando el resto de parámetros), podemos ver como mejoramos bastante los tiempos. Además, obtenemos mapas con mejor calidad que si cacheáramos siempre.

Property	NoCache	Always	Opt
DoorGenType	ALL	ALL	RANDOM
Random doors param	-	-	0.5
CacheType	-	ALWAYS	REFRESHER
Refresher cache divisor	-	-	10
SolverType	BestSearch	BestSearch	BestSearch
BestSearch DPE Divisor	1.0	1.0	0.9

Cuadro 7.10: Configuración para test de comparación

Tam. h.	Modelos	Inst./modelo	Total	NoCache	Always	Opt
8	4	2	8	0.0333	0.0174	0.0030
8	4	4	16	0.2818	0.0649	0.0123
8	4	6	24	1.2533	0.1949	0.0354
8	4	8	32	3.3727	0.4231	0.0780
8	6	2	12	0.1696	0.0666	0.0107
8	6	4	24	1.7436	0.2856	0.0481
8	6	6	36	8.6428	1.0331	0.1302
8	6	8	48	19.8587	2.7444	0.3155
8	8	2	16	0.5825	0.2523	0.0210
8	8	4	32	7.3917	1.1891	0.1030
8	8	6	48	33.7262	3.6202	0.3438
8	8	8	64	107.0144	9.6228	0.8999
8	10	2	20	1.6964	1.0235	0.0643
8	10	4	40	24.2987	3.1605	0.2243
8	10	6	60	113.6830	11.2674	0.9700
8	10	8	80	417.6313	28.2073	2.1928

Cuadro 7.11: Test de comparación

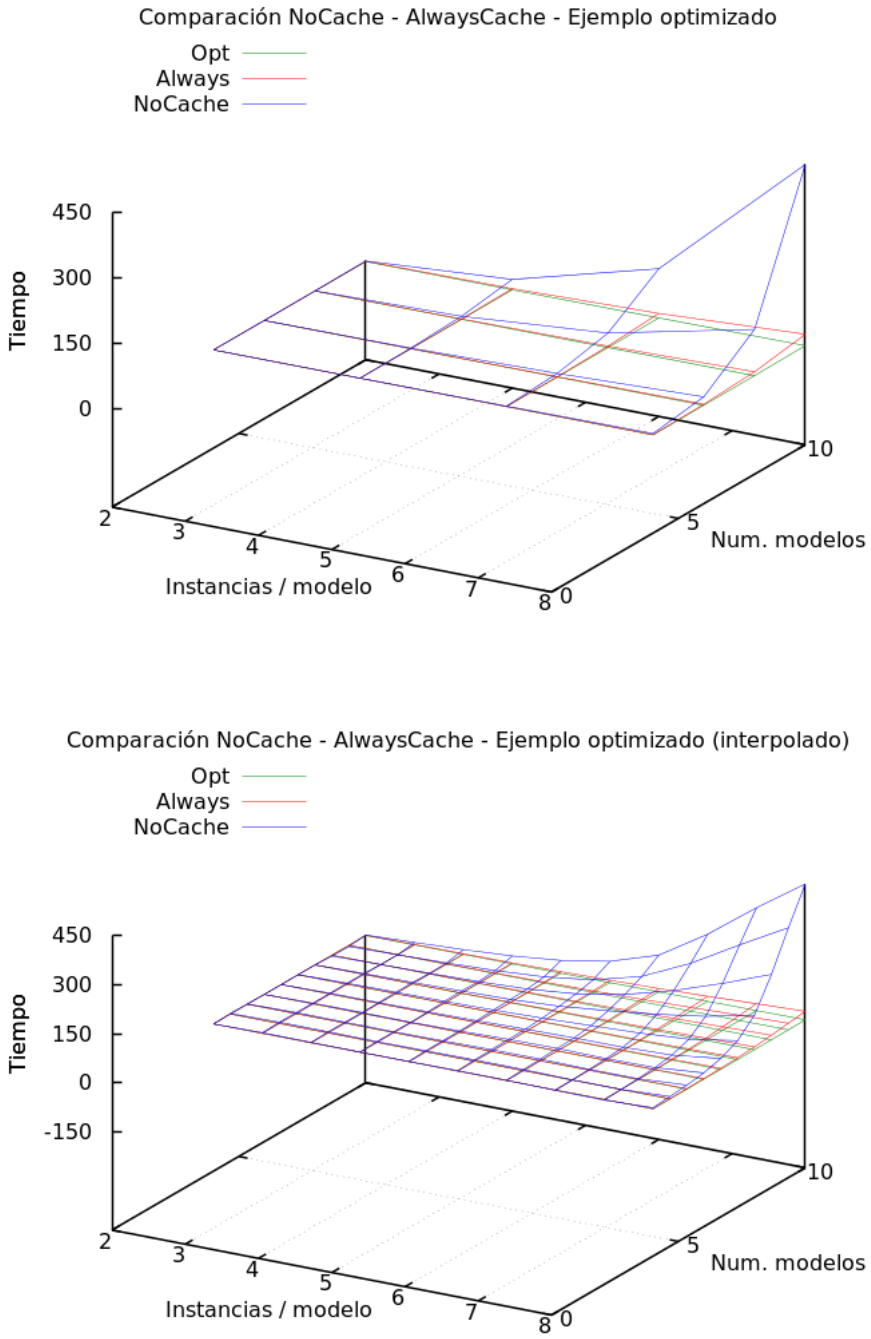

Figura 7.14: Gráfica de comparación entre AlwaysCache, NoCache y ejemplo optimizado

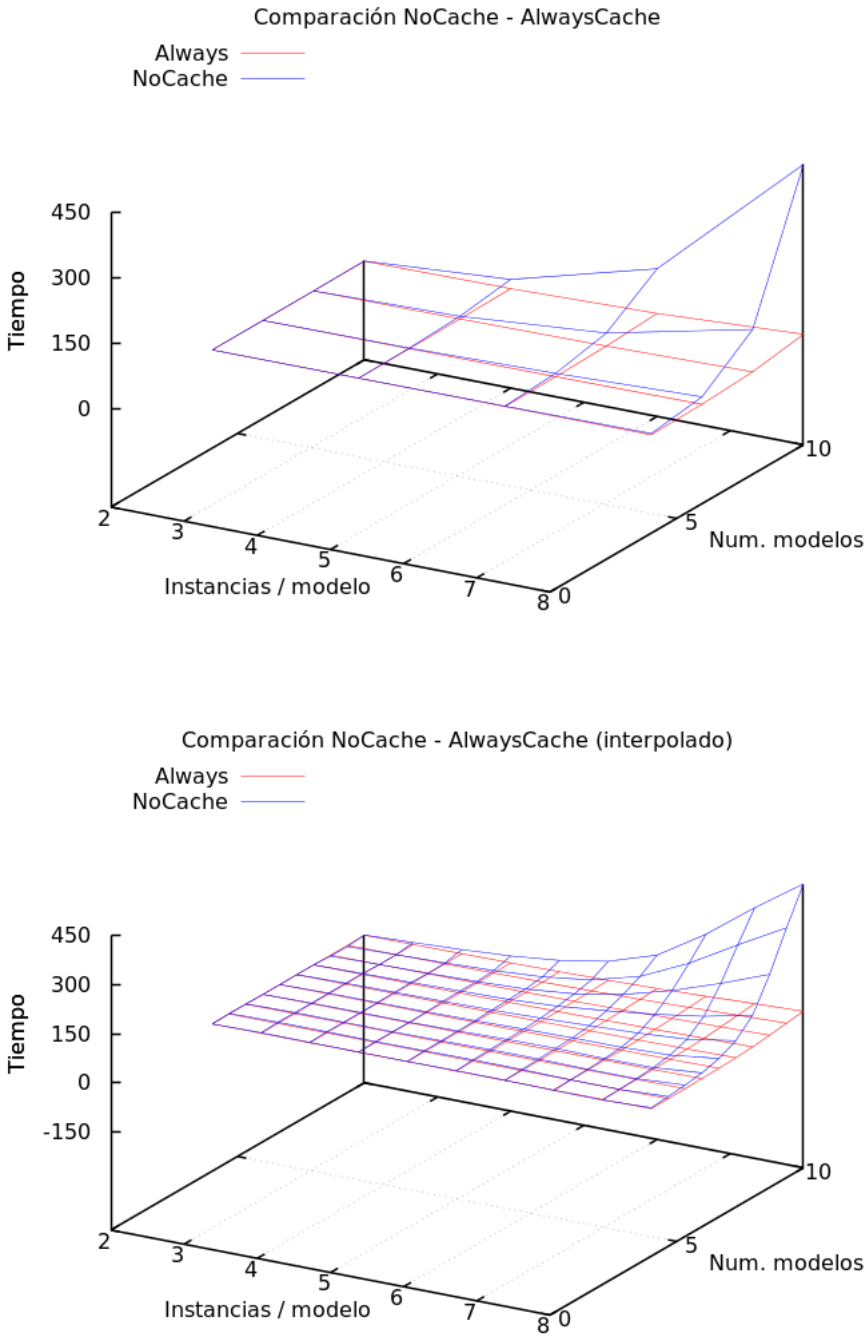

Figura 7.15: Gráfica de comparación entre AlwaysCache y NoCache

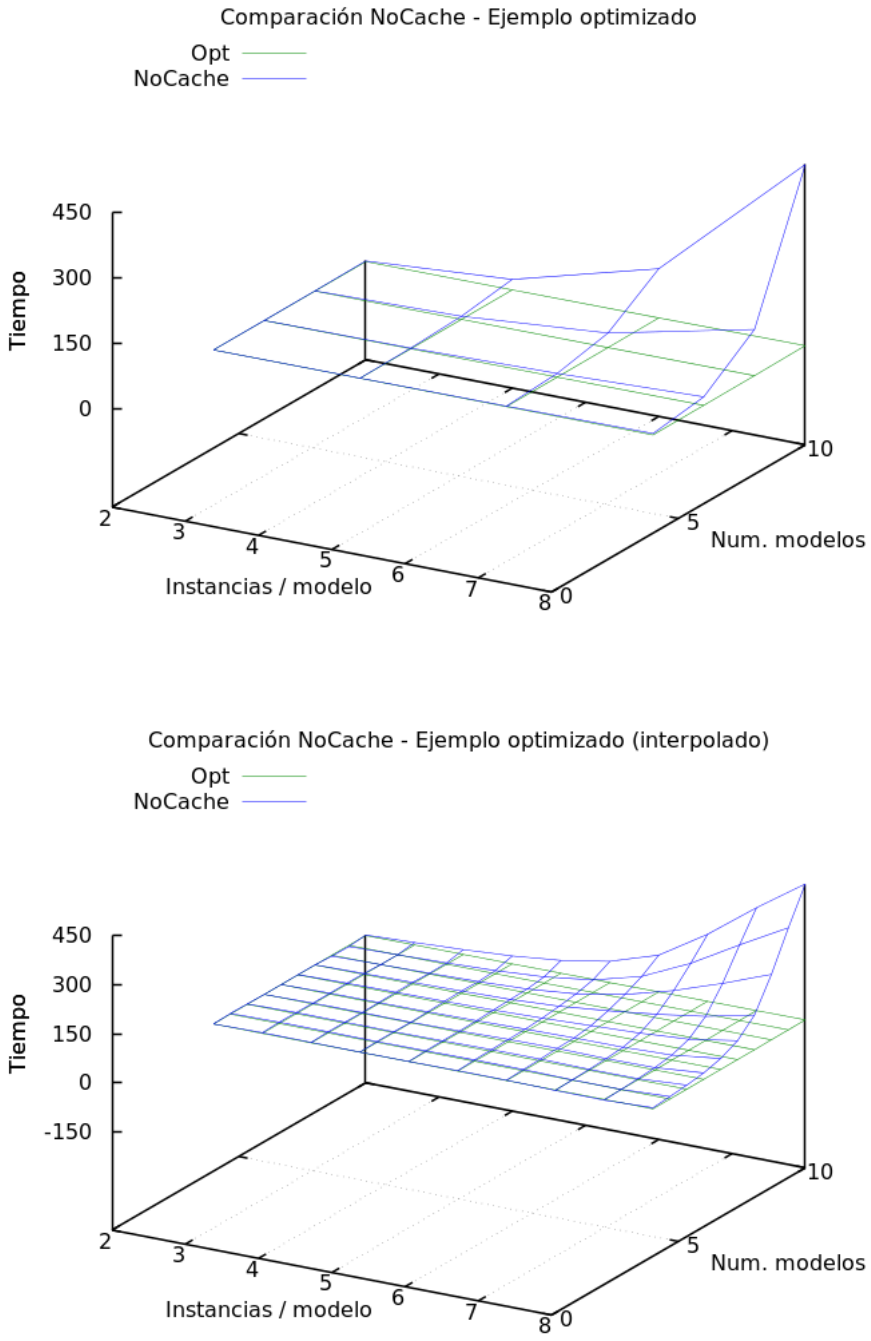

Figura 7.16: Gráfica de comparación entre NoCache y ejemplo optimizado

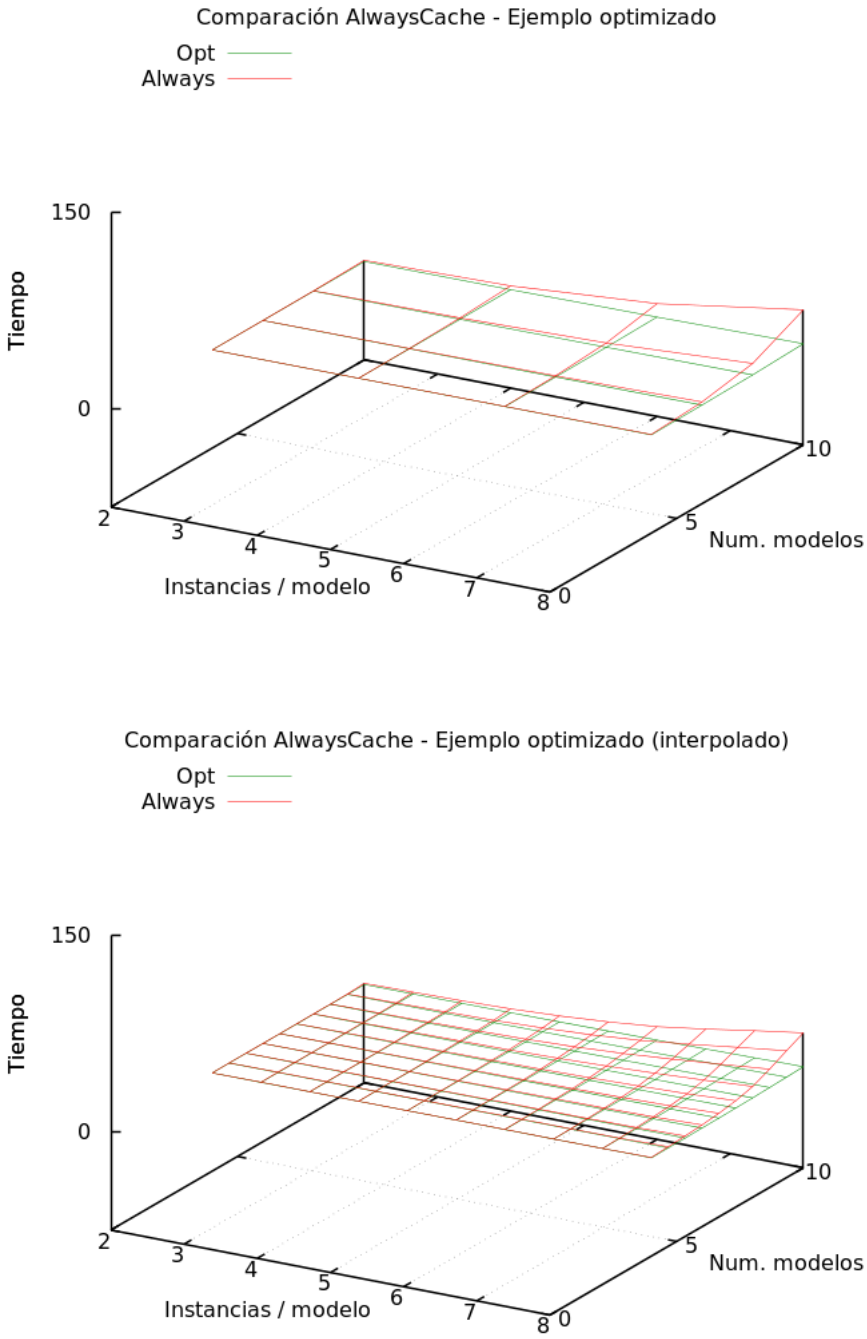

Figura 7.17: Gráfica de comparación entre AlwaysCache y ejemplo optimizado

7.2.6. Muchos tipos de habitación

Por último, hemos querido hacer un ejemplo donde la diversidad de modelos es muy alta. Es decir, se empleará un número alto de modelos y bajo de instancias por modelo.

En la gráfica **??**, podemos ver un caso que tarda un tiempo excesivo de 7 segundos. Para ello, se ha realizado una modificación en la configuración, obteniendo un arreglo del mismo que puede observarse en la figura **??**.

Property	Value
DoorGenType	RANDOM
Refresher cache divisor	10
SolverType	BestSearch
BestSearch DPE Divisor	0.9
CacheType	REFRESHER
Random doors param	0.5

Cuadro 7.12: Configuración test con variabilidad en habitaciones

Tam. habs.	Modelos	Instancias/modelo	Total habs.	Tiempo
6	20	1	20	0.0395
6	20	2	40	0.2505
6	30	1	30	0.1479
6	30	2	60	0.9846
6	40	1	40	0.4307
6	40	2	80	2.9426
6	50	1	50	0.9980
6	50	2	100	7.6388

Cuadro 7.13: Test con variabilidad en habitaciones

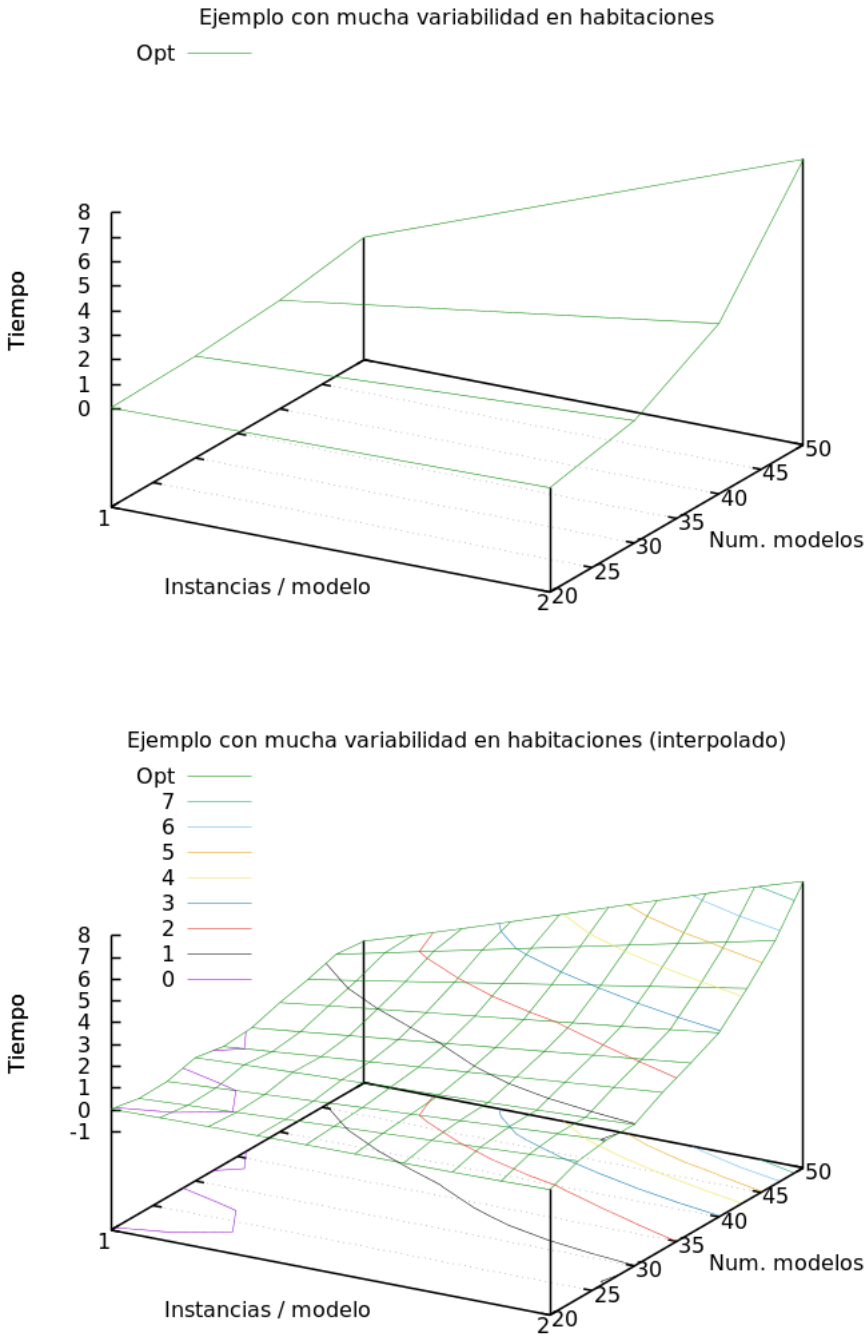

Figura 7.18: Gráfica de ejemplo variable con muchos modelos de habitaciones

Property	Value
DoorGenType	RANDOM
SolverType	BestSearch
BestSearch DPE Divisor	0.5
CacheType	ALWAYS
Random doors param	0.3

Cuadro 7.14: Configuración para el arreglo del último caso en test con variabilidad en habitaciones

Tam. habs.	Modelos	Instancias/modelo	Total habs.	Tiempo
6	20	1	20	0.1615
6	20	2	40	0.1523
6	30	1	30	0.0494
6	30	2	60	0.2139
6	40	1	40	0.1108
6	40	2	80	0.6791
6	50	1	50	0.2222
6	50	2	100	1.3180

Cuadro 7.15: Arreglo del último caso en test con variabilidad en habitaciones

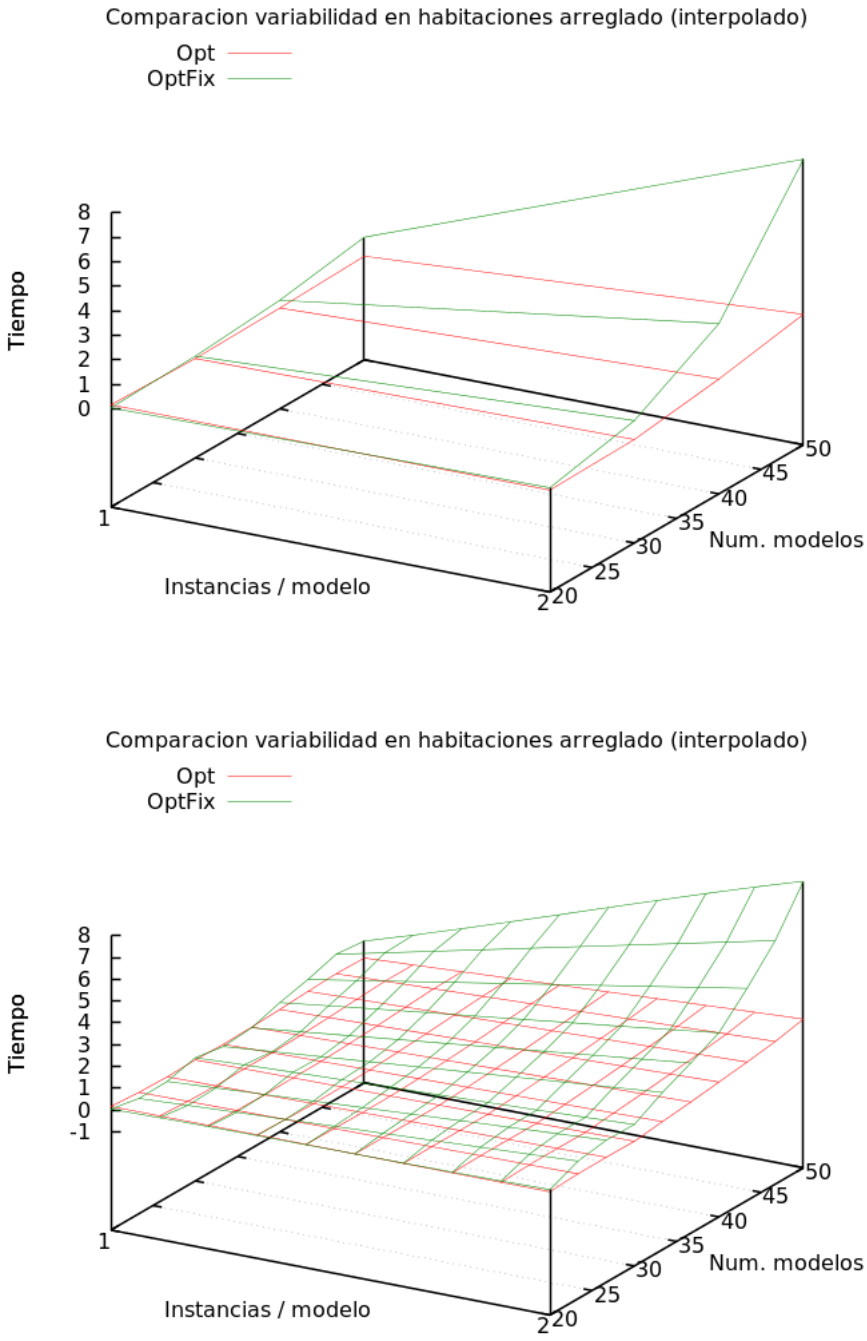

Figura 7.19: Gráfica de comparación entre ejemplo variable y arreglado

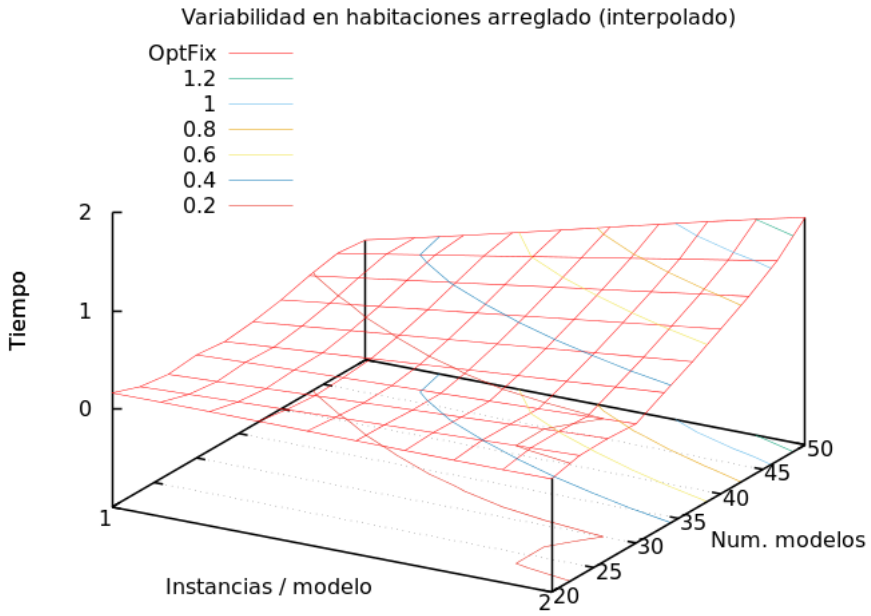

Figura 7.20: Arreglo de ejemplo variable

Capítulo 8

Trabajo futuro.

Comentaremos algunas posibles mejoras para el sistema, enfocadas principalmente a una posible versión comerciable del sistema que ofrezca algunas características extra y añadan atractivo para potenciar su posible comercialización.

8.1. Mejora del editor de habitaciones.

La implementación del editor de habitaciones es bastante rudimentaria. Podría haberse hecho más amigable al usuario, pero para un esbozo del sistema era suficiente. Además, normalmente, en los mapas de tiles se emplean varias capas para añadir detalles. Se podría añadir esta característica para que los usuarios pudieran añadir capas extra en las habitaciones.

8.2. Mapa de tamaño autoajustable

En el enunciado del problema, se supone que el mapa es lo suficientemente grande como para albergar cualquier distribución de habitaciones. Para solventar este requerimiento, se ha empleado un mapa de tiles grande, pero sería más correcto utilizar una implementación que vaya autoaumentando el tamaño de la matriz que representa el mapa conforme se

vaya necesitando más espacio para albergar habitaciones.

8.3. Fitness extra

Se ha implementado un sistema lo suficientemente flexible como para añadir otros objetivos de guía en la construcción del mapa. Esto da al usuario la ventaja de poder elaborar sus propios objetivos.

Podría añadirse dos objetivos de *horizontalidad* y *verticalidad*. Estos objetivos darían más puntuación a un movimiento, si este promueve que la disposición final sea horizontal o vertical. El cómputo numérico de este objetivo podría hacerse computando la recta horizontal o vertical media, y calculando la suma de las desviaciones de todas las habitaciones con respecto a dicha recta

Otra medida objetivo a tener encuenta podría ser una medida que estuviera relacionada con la dispersión o condensación de la disposición de las habitaciones. Esto podría realizarse calculando la desviación típica con respecto a la posición media de todas las habitaciones, obteniendo así una medida representativa de esta característica.

8.4. Flexibilidad en elección de puertas potenciales

Otro componente interesante a añadir en el sistema sería uno dedicado a la afinación de la elección de puertas potenciales. Por ejemplo, para un plataformas, podemos querer que las puertas potenciales sean solamente horizontales y no verticales. Quizá para un cierto tipo de juego, solo nos interese cierto subconjunto de puertas potenciales, y no todas, de forma que podríamos añadir un selector de puertas al editor de habitaciones y emplear dicho subconjunto elegido por el usuario en la ejecución del sistema.

8.5. Portar a móvil

Debido a los tiempos de ejecución obtenidos en la experimentación y a los distintos componentes relacionados con la eficiencia del tiempo de ejecución, sería posible portar el sistema a móviles para el uso en juegos en estos dispositivos. Un elemento importante en este aspecto es el denominado *fitness caché* que se comentó en el capítulo (ENLACE A CAPITULO), que nos permite sacrificar calidad de la generación a cambio de mejores tiempo de eficiencia.

8.6. Otros fitness caché

Relacionado con la sección anterior, podríamos elaborar varias implementaciones de la interfaz del fitness caché que controlaran mediante parámetro la eficiencia en cuanto a tiempo de ejecución del sistema.

Una posibilidad sería cachear solamente cada N intentos. Así, podríamos configurar este parámetro N para calibrar la relación calidad/tiempo de ejecución del sistema. Se podría hacer también con una probabilidad de que se cachee o no. Con esto, tendríamos bastante flexibilidad a la hora de ajustar la eficiencia al dispositivo en el que corra el sistema.

Otra posibilidad sería regenerar la caché cada N pasos del algoritmo. De esta forma, podemos controlar la relación calidad/tiempo de ejecución de forma más homogénea para todos los posibles movimientos.

8.7. Backtracking con guardado de movimientos

Como se comentó en el capítulo anterior, el sistema no está dotado de backtracking ya que no se vio en necesidad de ello, pero podría implementarse. Esto es gracias a que guardamos la lista de movimientos que construye una solución para poder obtener el mapa de tiles asociado a dicha solución. Así, cada N pasos, podríamos guardar la lista de movimientos generados para poder reanudar la generación en ese punto. También podríamos simplemente tener en cuenta si hay varias soluciones mejores, es decir, soluciones que están dotadas de la misma puntuación, o dando

un cierto margen de diferencia. De esta forma, también podríamos reanudar la generación a partir de ellas.

Capítulo 9

Conclusiones

Para terminar, elaboraremos las conclusiones que se han extraído de la elaboración y desarrollo de esta memoria, desde la parte de investigación hasta la implementación de un sistema capaz de generar escenarios de tiles.

9.1. Camino recorrido

9.1.1. AnsProlog

Inicialmente, el proyecto se enfocó a utilizar *AnsProlog* [**?**], un lenguaje declarativo orientado a la resolución de problemas *NP-Hard* [**?**] principalmente. Recientemente, se ha investigado bastante en la generación procedimental de contenido para videojuegos empleando este sistema [**?**].

Se investigó esta línea de forma práctica. Se elaboraron tests de prueba [**?**] para validar la factibilidad del uso de este lenguaje, pero finalmente se decidió despojarse de este acercamiento, ya que para poder utilizar el motor para procesar dicho lenguaje, habría hecho falta portar el propio motor al sistema donde fuera a ser utilizado, en nuestro caso, Java.

Por ello, se optó por tomar una línea más imperativa al acercamiento del problema.

9.1.2. Algoritmos genéticos

Después de elaborar la representación, se pensó en una forma de enfocar el problema usando algoritmos genéticos. Para ello, se procuraba una disposición inicial aleatoria de las habitaciones, para posteriormente computar todas las posibles conexiones entre habitaciones.

Una vez obtenida las posibles conexiones entre habitaciones, como modelo de datos del algoritmo genético se emplearía una ristra de bits, donde cada una de las posibles conexiones se vería representada por un bit que indicaría si esa conexión está activa o no.

El problema es que las posibilidades eran nimias. Era incluso posible computar todas las posibilidades, siendo la resolución directa de esta forma, y además limitando la flexibilidad a solo intervenir en la elección de puertas.

9.1.3. Búsqueda

Finalmente, este fue el acercamiento que más factibilidad tenía, y el que se desarrolló. La adición de elementos extra a la generación y la opción de no seguir religiosamente el método de búsqueda han sido claves para poder obtener flexibilidad a la hora de generar los escenarios.

9.2. Conclusiones finales

El campo de la generación procedimental de contenido es vasto y amplio. En este proyecto solo se ha enfocado uno de las ramas del mismo, concretamente enfocándonos a escenarios 2D de tiles.

La popularidad de este tipo de métodos esta totalmente en auge, ya que gracias a ello ahorramos en recursos y damos un cierto margen de dinamismo a los videojuegos.

Bibliografía

[1] LucasArts Wikipedia page

[2] Occlusion culling

[3] Techopedia LOD page

[4] Voxel wikipedia

[5] Quadtree wikipedia

[6] Octree wikipedia

[7] Brush wikipedia

[8] Quake wikipedia

[9] Indie game developers

[10] RogueLike en Wikipedia

[11] List of most expensive video games to develop.

[12] Gamedev Job Roles at CreativeSkillSet

[13] Video Game Development Wikipedia page

[14] DemoScene wikipedia

[15] http://www.amazon.com/Texturing-Modeling-Third-Edition-Procedural/dp/1558608486

[16] http://libnoise.sourceforge.net/tutorials/tutorial3.html

[17] http://www-cs-students.stanford.edu/ amitp/game-programming/polygon-map-generation/

[18] http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap

[19] Explicación del algoritmo de generación de mazmorras empleado en TinyKeep.

[20] TheGameKitchen