

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



“DISEÑO DE ALGORITMO DE GENERACIÓN PROCEDURAL ENFOCADO A VIDEOJUEGOS”

CRISTÓBAL SCHEIHING ORREGO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Elizabeth Montero
Profesor Correferente: Erika Rosas

Agosto - 2019

AGRADECIMIENTOS

Me gustaría agradecer a la profesora Elizabeth Montero por guiarme durante todo este proceso que duro varios meses. Gracias a su guía y apoyo siento que este trabajo ha resultado mucho mejor de lo que tenía inicialmente y he aprendido mucho.

También quiero agradecer a mi familia y amigos por todo el apoyo incondicional que me dieron durante estos meses de trabajo, sin ellos no hubiera sido lo mismo.

RESUMEN

Resumen— La generación automática de mapas para videojuegos es un problema crucial para poder ofrecer escenarios creativos, innovadores y que otorguen experiencias distintas cada vez que se vuelven a jugar los videojuegos. Dentro de las formas en que se puede abarcar este problema está la generación de contenido procedural o PCG por sus siglas en inglés, la cual entre todas las cosas que puede generar, se encuentra el generar mapas para videojuegos.

Dentro de las formas en que se ha usado PCG en el pasado para abarcar este problema se encuentran: autómatas celulares, gramática generativa, algoritmos genéticos, basado en restricciones, entre otros. De estos enfoques se decidió utilizar un algoritmo genético, ya que entrega los mejores resultados de lo que se quiere lograr.

Los algoritmos genéticos trabajan sobre una población de soluciones que son sometidas a las fases de evaluación, elitismo, selección, cruzamiento y mutación y esto se repite durante varias generaciones para ir mejorando las soluciones en cada una.

Los parámetros usados fueron elegidos para tener el mayor control posible sobre los resultados finales, de manera de ajustar cada tipo de celda presente en el mapa. Los resultados obtenidos fueron satisfactorios para las instancias pequeñas y medianas, sin embargo fue en las instancias más grandes donde el algoritmo necesito de un mayor tiempo de cómputo para entregar los resultados deseados.

Palabras Clave— IA., GCP., AG.

ABSTRACT

Abstract— The automatic generation of maps for videogames is a crucial problem to be able to offer creative, innovative scenarios and that they can give different experiences each time the videogame are played again. Among the ways in which this problem can be covered is the procedural content generation or PCG by its acronym in English, which among all the things that can generate, is to generate maps for video games.

Among the ways in which PCG has been used in the past to cover this problem are: cellular automata, generative grammar, genetic algorithms, constraint-based, among others. From these approaches it was decided to use a genetic algorithm, since it delivers the best results of what we want to achieve with the algorithm.

The genetic algorithms work on a population of solutions that are subjected to the phases of evaluation, elitism, selection, crossing and mutation and this is repeated over several generations to improve the solutions in each one.

The parameters used were chosen to have the greatest possible control over the final results, in order to adjust each type of cell present in the map. The results obtained were satisfactory for the small and medium instances, however it was in the larger instances where the algorithm needed a longer computing time to deliver the desired results.

Keywords— AI., PCG., GA.

GLOSARIO

- **PCG:** Procedural Content Generation - Generación de Contenido Procedural.
- **FPS:** First Person Shooter - videojuego de disparos en primera persona.
- **RPG:** Role Playing Game - Juego de rol.
- **RTS:** Real Time Strategy - videojuego de estrategia en tiempo real.
- **Dungeon:** Calabozo o Mazmorra.
- **NPC:** Non-Player Character - Personaje no jugable.
- **Algoritmo:** Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.
- **Unity:** Plataforma de desarrollo de videojuegos.
- **IA:** Inteligencia Artificial.
- **GA:** Genetic Algorithm - Algoritmo Genético.

ÍNDICE DE CONTENIDOS

RESUMEN	II
ABSTRACT	III
GLOSARIO	IV
ÍNDICE DE FIGURAS	VII
ÍNDICE DE TABLAS	VIII
INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	2
1.1 Situación Actual	2
1.2 Tipos de videojuegos y PCG	3
1.3 Objetivos	5
1.3.1 Objetivo general	5
1.3.2 Objetivos específicos	5
CAPÍTULO 2: MARCO CONCEPTUAL	7
2.1 Discusión bibliográfica	7
2.2 Propiedades y Taxonomía	10
2.3 Algoritmos	12
2.4 Otras áreas	19
CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	22
3.1 Algoritmo genético	22
3.2 Propuesta	22
3.3 Representación	24
3.4 Inicialización	25
3.5 Selección	27
3.6 Cruzamiento	28
3.7 Mutación	28
3.8 Función de Evaluación	29
3.8.1 Factibilidad	30
3.8.2 Calidad	32
3.9 Elitismo	33
3.10 Ejemplo final	33
3.11 Parámetros de entrada	33
3.12 Variantes	34
3.13 Fase 1	34
3.14 Fase 2	34

3.15 Fase 3	35
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	37
4.1 Indicadores	38
4.2 Instancias/Casos de pruebas	38
4.3 Mapas de tamaño pequeño	41
4.4 Mapas de tamaño mediano	47
4.5 Mapas de tamaño grande	53
CAPÍTULO 5: CONCLUSIONES	61
REFERENCIAS BIBLIOGRÁFICAS	63

ÍNDICE DE FIGURAS

1	Árbol de definición del problema.	3
2	Ejemplo de un mapa de un dungeon generado con BSP.	8
3	Ejemplo de distintos patrones de disparos.	9
4	Ejemplo del juego Beneath Apple Manor.	10
5	Mapa generado con autómatas celulares.	14
6	Grafo de jugabilidad generado con Generative grammar.	15
7	Mapa generado a partir del grafo de la figura 6.	15
8	Estructura del árbol traducida a estructura del mapa.	16
9	Ejemplo de mapa generado con algoritmo genético.	17
10	Geometría de mapa 3D (derecha) a partir de un grafo generado (izquierda) . .	18
11	Ejemplo de un dungeon aleatorio generado.	20
12	Ejemplo de decodificación de un árbol en un mapa.	25
13	Ejemplo de Inicialización de una solución.	27
14	Ejemplo de cruzamiento	29
15	Ejemplo de mutación.	30
16	Ejemplo de traducción de árbol a mapa	33
17	Comparación entre las 3 fases con configuración estándar con mapas de tamaño 4.	42
18	Ejemplo del resultado de un mapa con configuración estándar con mapas de tamaño 4.	43
19	Comparación entre las 3 fases con configuración laberinto con mapas de tamaño 4.	45
20	Ejemplo del resultado de un mapa con configuración laberinto con mapas de tamaño 4.	46

21 Comparación entre las 3 fases con configuración estándar con mapas de tamaño 7.	48
22 Ejemplo del resultado de un mapa con configuración estándar con mapas de tamaño 7.	49
23 Comparación entre las 3 fases con configuración laberinto con mapas de tamaño 7.	51
24 Ejemplo del resultado de un mapa con configuración laberinto con mapas de tamaño 7.	52
25 Comparación entre las 3 fases con configuración estándar con mapas de tamaño 9.	54
26 Ejemplo del resultado de un mapa con configuración estándar con mapas de tamaño 9.	55
27 Comparación entre las 3 fases con configuración laberinto con mapas de tamaño 9.	57
28 Ejemplo del resultado de un mapa con configuración laberinto con mapas de tamaño 9.	58

ÍNDICE DE TABLAS

1 Parámetros usados en la configuración estándar.	39
2 Parámetros usados en la configuración laberinto.	40
3 Parámetros del algoritmo genético.	40

INTRODUCCIÓN

En este escrito se presenta el diseño e implementación de una solución para el problema de generación automática de mapas para videojuegos. Este problema consiste en que los videojuegos hoy en día necesitan ser creativos, innovadores y ser capaces de ofrecer experiencias distintas cada vez que se juegan. El diseño de niveles y mapas es crucial para este propósito y la creación manual de estos mapas resulta poco atractiva y costosa.

Después de realizar una investigación en el estado del arte sobre los videojuegos y los algoritmos existentes para resolver esta problemática, se decidió desarrollar una generación de contenido procedural (PCG) usando un Algoritmo Genético como base. Los algoritmos genéticos imitan el proceso de evolución darwiniano de los seres vivos. Un AG es un proceso repetitivo que en cada fase realiza procesos de evaluación, elitismo, selección, cruzamiento y mutación. Los mejores individuos de una población tienen mayores posibilidades de sobrevivir y reproducirse. En otras palabras un algoritmo genético evalúa los individuos de una generación, selecciona los mejores según los resultados de la evaluación, selecciona individuos de esta (no necesariamente los mejores, para que exista variedad) y los somete a un proceso de transformación en donde cruza distintas partes de individuos y luego los muta, cambiando datos de estos para obtener una nueva generación y así sucesivamente. Luego de acabar con este proceso se genera un mapa en base al mejor individuo encontrado.

Por lo tanto, el presente trabajo consiste en el diseño e implementación de un PCG utilizando un Algoritmo Genético para la resolución del problema de generación automática de mapas para videojuegos. Posterior a esto se realizan pruebas para verificar que la solución propuesta es capaz de resolver el problema entregando resultados satisfactorios. Este trabajo está estructurado en los siguientes capítulos:

- **Capítulo 1 - Definición del problema:** Se presenta el problema de la generación automática de mapas, la situación actual en que se encuentran los videojuegos, se presentan los principales tipos de videojuegos que existen y que técnicas PCG se han profundizado. Se presenta un árbol del problema y los objetivos de la memoria.
- **Capítulo 2 - Marco Conceptual:** Se presenta el estado del arte sobre que es el PCG, que conlleva y los algoritmos que se pueden usar para trabajar con él.
- **Capítulo 3 - Propuesta de Solución:** Se presentan las características y el diseño del algoritmo propuesto para la resolución del problema.
- **Capítulo 4 - Validación de la solución:** Se presentan los experimentos y resultados obtenidos al implementar la solución propuesta.
- **Capítulo 5 - Conclusiones:** Se presentan las conclusiones del trabajo realizado y se habla de posibles trabajos a futuro.

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

Se introduce la situación actual en la que se encuentran los videojuegos, qué características son fundamentales para su éxito y algunos de los tipos de videojuegos que existen. Además, se describe que tipo de elementos necesitan ser creados en un videojuego y donde aparece el PCG (Procedural Content Generation - Generación de Contenido Procedural) en este tema. En este capítulo se presenta el contexto general del problema y los objetivos del trabajo.

1.1. Situación Actual

En los videojuegos, mejorar la experiencia de juego es un factor fundamental para su éxito, por lo que el diseño de niveles y mapas se vuelve un factor crítico. Las exigencias de los jugadores actuales son entornos de juegos creativos, innovadores y que sean capaces de ofrecerles experiencias diferentes cada vez que juegan. El problema de esto está en que la creación manual de mapas en general resulta poco atractiva y costosa, ya que en los juegos actuales existe una gran cantidad de mapas y contratar a diseñadores de niveles se vuelve muy costoso, con lo que se terminan usando los recursos destinados a mejorar otros aspectos del videojuego en la creación de mapas y conlleva a que el crear el videojuego se vuelva costoso en su totalidad.

Por otro lado, el problema con estos mapas hechos a mano es que algunos de ellos no quedan bien, con grandes espacios sin nada interesante o poco atractivos y al estar 'definidos', cuando se vuelven a jugar, los mapas son exactamente igual por lo que el jugador se aprende los mapas, posición de objetos y enemigos, por lo que la experiencia de juego solo se puede vivir una vez.

Con esto dicho, se puede ver en la figura 1 un resumen a modo de árbol de la definición del problema.

El problema con que los mapas tengan grandes espacios vacíos (Causas) conlleva a que el mapa sea poco creíble y por lo tanto el jugador no se puede sumir en la atmósfera del juego y termina empeorando su experiencia al jugar (Efectos). Por otra parte, el que exista una gran cantidad de mapas en los videojuegos (Causas) conlleva a que se genere una disminución en los recursos que se podrían usar para mejorar otros aspectos como la jugabilidad, la música, la historia o cualquier otro aspecto del videojuego (Efectos).

Finalmente, todo esto hace que la existencia de mapas poco atractivos y costosos sea un problema crítico que se debe solucionar.

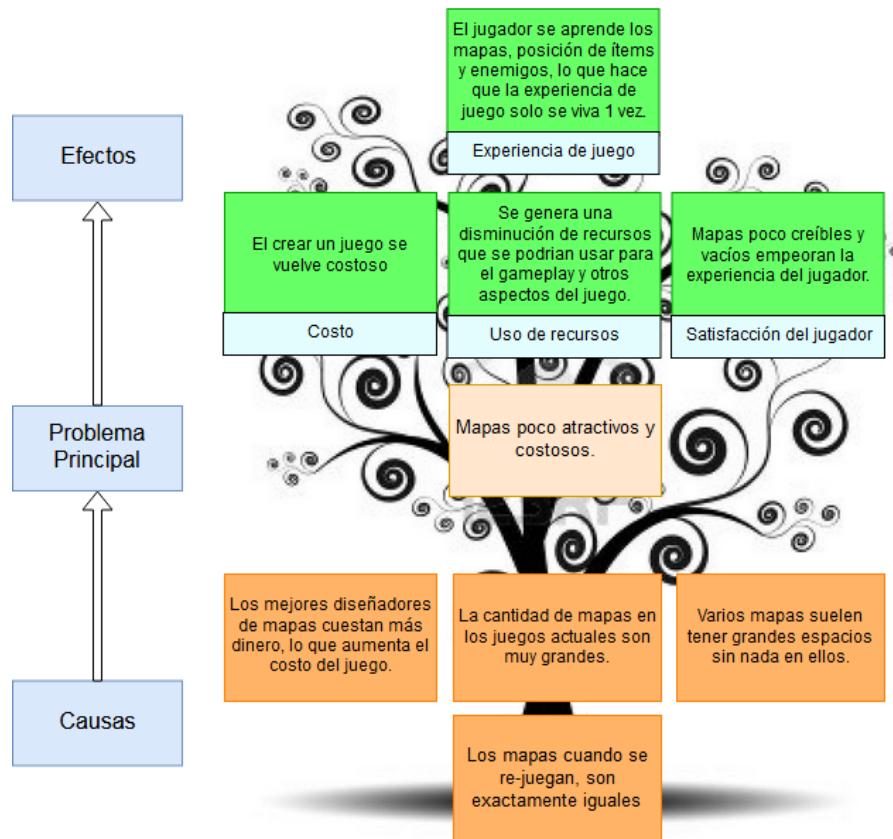


Figura 1: Árbol de definición del problema.

Fuente: Elaboración propia.

1.2. Tipos de videojuegos y PCG

¿Qué se entiende por videojuego? La explicación más general probablemente es la de un juego electrónico en donde uno o más jugadores interactúan mediante un control con un dispositivo que muestra imágenes de vídeo. Esta definición queda corta para lo que se entiende hoy por videojuego, una explicación más actual sería la siguiente: Los videojuegos recrean entornos y situaciones virtuales en los que el videojugador puede controlar a uno o varios personajes (o cualquier otro elemento de dicho entorno), para conseguir uno o varios objetivos dentro de unas reglas determinadas.

Existen diversos tipos de juegos que necesitan crear una gran variedad de mapas, entre ellos se encuentran los siguientes:

- FPS (First Person Shooter - videojuego de disparos en primera persona): Los FPS son videojuegos en donde el jugador observa el mundo desde la perspectiva del personaje protagonista y por lo general solo tienes armas de disparo para vencer el juego. Algu-

nos ejemplos de este tipo son el Quake, Doom y Counter-Strike. El propósito de los FPS por lo general es vencer a otros jugadores o avanzar en la historia mientras le disparas a los enemigos que aparecen y es en estos 2 puntos donde aparece la importancia de los mapas ya que estos tienen que estar diseñados de cierta forma que entretenga a los jugadores. Por ejemplo los mapas pueden tener lugares donde uno pueda disparar y esconderse al mismo tiempo de los otros jugadores o que tenga ciertas zonas en donde el jugador no sepa si se encontrara con un enemigo o un objeto al dar la vuelta en la esquina.

- **RPG (Role Playing Game - Juego de rol):** Los RPG tienen su origen en los juegos de sobremesa como Dungeons & Dragons en donde el jugador toma control de un personaje o grupo de personajes inmersos en un mundo detallado. Una de las características que usualmente se ven en los RPG es el crecimiento de estadísticas de los personajes, como la fuerza, la agilidad, etc. Algunos ejemplos de estos son los Final Fantasy, World of Warcraft, etc. El propósito de los RPG es avanzar en la historia mientras se derrotan enemigos para subir de nivel y así subir las estadísticas de los personajes. Al avanzar en la historia uno suele recorrer el mundo y por tanto recorrer diversos mapas, estos suelen ser muy detallados por lo que hay un enfoque en la calidad artística del mapa.
- **RTS(Real Time Strategy - Estrategia en tiempo real):** Los RTS son juegos de estrategia en los que no hay turnos entre jugadores, todo ocurre de forma continua. En este tipo de juegos los mapas suelen ser pequeños pero detallados en el aspecto gráfico y suele tener recursos para poder construir edificios y unidades de combate. Algunos ejemplos de RTS son el Age of Empire y el StarCraft. Si bien en los RTS la forma en que se componen los mapas no es crítica, si lo es la posición de los objetos, que en este caso serían los recursos ya que la obtención de estos es vital para poder ganarle a los otros jugadores y por lo tanto, la forma que tienen los mapas para poder ubicar estos recursos es de suma importancia.
- **Aventura:** Los videojuegos de aventura son videojuegos que se enfocan en la investigación, exploración y solución de puzzles, con la interacción con otros personajes y un enfoque en la historia. Algunos ejemplos son The Legend of Zelda y The Secret of Monkey Island. El propósito de los juegos de aventura es contar una historia mientras se avanza resolviendo los puzzles por lo que la ubicación de objetos, personajes y estructura de los mapas es crucial para el éxito de estos.
- **Acción:** Los videojuegos de acción son juegos en donde el jugador debe usar su velocidad, destreza y tiempos de reacción para superar obstáculos, ya sea en juegos de pelea, plataforma u otros subgéneros de los videojuegos de acción. Algunos ejemplos de este tipo son el Kung-Fu Master y River City Ransom. El propósito de los videojuegos de acción es saber reaccionar ante los enemigos que aparecen y lograr vencerlos mientras se avanza en el mapa, es por esto que el diseño de los mapas y la ubicación de los enemigos es crítica a la hora de hacer un videojuego de acción.
- **Mazmorras¹:** Los videojuegos de mazmorras son juegos en los que destaca la explora-

¹Tanto los de tipo Dungeon crawl como los de Rougelike

ción de calabozos o zonas cerradas en donde el jugador debe intentar salir de estos, ya sea mediante la resolución de acertijos, explorando o venciendo enemigos. Algunos ejemplos de este tipo son el Diablo y Wizardry. El propósito de los videojuegos de mazmorras por lo general es avanzar en estos calabozos, bajando pisos para encontrar un objeto o vencer un enemigo poderoso ó subir pisos para poder salir de la mazmorra. Es en este tipo de juegos donde el diseño de mapas es crítico para el juego, ya que el foco de este tipo de videojuego esta en los distintos mapas que el jugador recorre, los enemigos que se enfrenta y los objetos que encuentra.

Además los videojuegos necesitan crear todo tipo de elementos, entre ellos mapas (Ciudades, mundos, dungeons, etc.), objetos, enemigos, NPC, ubicación de objetos, misiones, historia, etc.

Todas estos elementos se pueden generar utilizando PCG y la diferencia entre generar cualquiera de estos elementos recae en la lógica detrás del algoritmo PCG, no es lo mismo programar un algoritmo para generar los objetos, que hacer uno para generar la historia de cada NPC ya que estos implican distintas consideraciones a la hora de diseñar el algoritmo. Por otro lado, generar el mapa de una ciudad (que contiene edificios, personas, etc.) es distinto a generar el mapa de un dungeon (Calabozo o Mazmorra, el cual contiene, laberintos, enemigos, tesoros, etc.), por lo que saber en qué enfocar el PCG también es un problema en sí, por la gran variedad de alternativas que se pueden considerar.

1.3. Objetivos

A partir del problema expuesto para este trabajo de memoria, se consideran los siguientes objetivos:

1.3.1. Objetivo general

Generar de manera automatizada mapas más interesantes o atractivos, de manera pro-activa utilizando PCG.

1.3.2. Objetivos específicos

- Crear un algoritmo que genere de manera procedural la ubicación de las estructuras en el dungeon siguiendo ciertas reglas para mantener un nivel de dificultad.
- Obtener distribuciones del mapa distintos cada vez que se juega una nueva partida utilizando un algoritmo genético.

- Crear un prototipo de videojuego en primera persona, en donde validar el algoritmo que será implementado.

CAPÍTULO 2

MARCO CONCEPTUAL

En este capítulo se expone el estado del arte del PCG, en donde primero se menciona que es el PCG y las cosas que este conlleva, hablando de sus propiedades y taxonomías para luego pasar a los algoritmos más usados con PCG para la generación de mapas hoy en día y por último se mencionan otras áreas en donde el PCG es utilizado.

2.1. Discusión bibliográfica

¿Qué es PCG? Según [Shaker *et al.*, 2016] PCG se puede definir como un algoritmo que crea contenido en un juego con una intervención indirecta o limitada del usuario. En otras palabras PCG se refiere a un software de computador que puede crear contenido por sí mismo o en conjunto con diseñadores o jugadores.

Un punto importante a considerar es el de 'contenido', con esto se refiere a los elementos que suele tener un videojuego: mapas, niveles, reglas, objetos, posición de enemigos, texturas, historias, misiones, etc.

Otro punto importante a considerar es el término 'juego', con esto se refiere a videojuegos, juegos de computador, juego de cartas, puzzles, etc. Es importante destacar que el sistema que generara contenido tenga en consideración las reglas, diseño y costumbres del juego que se generara.

Por último, es un requerimiento clave que el contenido generado sea jugable, es decir, se debe poder completar el nivel, se debe poder subir las escaleras generadas, se debe poder usar las armas generadas, etc.

A continuación se listan algunas de las aplicaciones que son consideradas PCG:

- Una herramienta de software que crea dungeons para juegos de acción/aventura como "The Legend of Zelda" sin ninguna intervención humana - cada vez que se corre el programa, un nuevo nivel es creado. Un ejemplo de esto es la técnica BSP (binary space partitioning) la cual divide recursivamente un espacio en 2 sub-espacios, lo más común es usar celdas cuadradas y generar un mapa a partir de un ancho w y un alto h . En la figura 2 se puede ver un ejemplo de un mapa creado usando esta técnica.



Figura 2: Ejemplo de un mapa de un dungeon generado con BSP.
Fuente: [Shaker *et al.*, 2016].

- Un sistema que crea nuevas armas en un juego de disparos espacial en respuesta a lo que realizan colectivamente los jugadores, de manera que las armas que se encuentran los jugadores sean versiones evolucionadas de las que otros jugadores encuentran divertidas de usar. Un ejemplo de esto se habla en [Hastings *et al.*, 2009] en donde las armas que un jugador encuentra son armas que han ido evolucionando de armas que otros jugadores han usado en el pasado, con evolución no se refiere a la fuerza de los disparos, sino que a la forma en que salen las partículas del arma. En la figura 3 se pueden ver algunos ejemplos de las distintas formas en que salen los disparos en el juego.

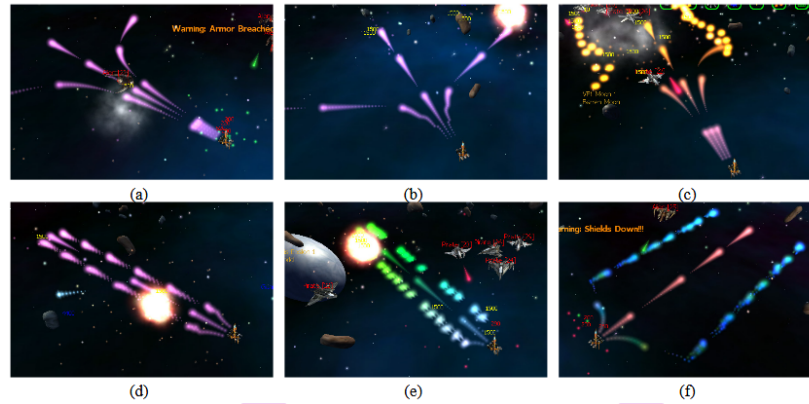


Figura 3: Ejemplo de distintos patrones de disparos.
Fuente: [Hastings *et al.*, 2009].

- Un programa que genera un juego de tablero completo, balanceado y jugable por sí mismo, puede ser cualquier juego de tablero, pudiendo usar un juego de tablero existente como base. Un ejemplo de este es METAGAME [Pell, 1992] que está basado en juegos de tipo ajedrez.
- Un motor de juego que rápidamente puede poblar el mundo de un juego con vegetación. Uno de los más famosos es el middleware SpeedTree, el cual ha sido usado en diversos juegos grandes, por lo que se puede poblar tanto mapas pequeños como grandes.
- Una herramienta de diseño gráfica que permite al usuario diseñar mapas para un juego de estrategia, mientras se va evaluando continuamente el mapa diseñado y sus propiedades de jugabilidad, para así sugerir mejoras en el mapa para hacerlo más balanceado e interesante. Un ejemplo de esto se puede encontrar en [Shaker *et al.*, 2012].

¿Desde cuándo existe el PCG? El problema de generar los mapas para los videojuegos existe desde sus inicios y por lo tanto también lo es el PCG, en donde el primer juego en utilizarlo fue 'Beneath Apple Manor' en 1978, el cual consiste en obtener la manzana dorada al final del dungeon, contando con varios pisos en los cuales hay enemigos, puertas bloqueadas, objetos, etc. Cada elemento se genera de manera aleatoria al entrar en él, por lo que se puede decir que el PCG surge en esta fecha con este juego.



Figura 4: Ejemplo del juego Beneath Apple Manor.
Fuente: Google Images.

2.2. Propiedades y Taxonomía

En [Shaker *et al.*, 2016] exponen sobre las propiedades y taxonomía que tienen los algoritmos de PCG en general ya que estos son pensados como soluciones a problemas de generación de contenido. Las propiedades deseables o requeridas para una solución varían según el problema que se quiere abarcar pero en [Shaker *et al.*, 2016] se listan las propiedades más comunes:

- **Velocidad** - Los requerimientos de velocidad varían enormemente, desde un máximo de tiempo de generación de milisegundos a meses, dependiendo si (entre otras cosas) el contenido a generar es logrado durante el tiempo de juego o durante el desarrollo del juego.
- **Confiabilidad** - Se refiere a que mientras que algunos generadores hacen que un jugador dispare desde la cadera en vez desde la pistola, otros son capaces de garantizar que el contenido que generan satisface un criterio de calidad dado. Esto es más importante para algunos contenidos que otros, por ejemplo un dungeon sin salida o entrada es un fallo catastrófico mientras que una flor que se ve rara solo se verá rara, sin necesariamente romper el juego.
- **Controlabilidad** - Frecuentemente hay una necesidad de que el contenido generado sea controlable en algún sentido, de manera que un humano o algoritmo (como un mecanismo adaptativo de jugador) pueda especificar algunos aspectos del contenido a generarse. Existen diversas dimensiones de control, por ejemplo uno puede pedir un auto que pueda tomar curvas cerradas y que tenga múltiples colores, o uno puede pedir un nivel que induce una sensación de misterio y premia a los perfeccionistas, etc.

- **Expresividad y diversidad** - Usualmente existe una necesidad de generar diversos sets de contenido, para evitar que el contenido luzca como si todos son variaciones mínimas en un tema sobreexplotado. Medir expresividad no es un tema trivial en sí mismo y diseñar generadores de niveles que generen contenido diverso sin comprometer la calidad de este es aún menos trivial.
- **Creatividad y credibilidad** - En la mayoría de los casos, uno no quiere que el contenido parezca generado por un PCG. Existen numerosas formas de que el contenido generado se vea generado de manera distinta al generado por un humano.

Si bien no es necesario que se abarquen todas estas características o hasta podrían no centrarse en ninguna al realizar el algoritmo, lo que hacen los PCG por lo general es incluir una combinación de estos, como por ejemplo velocidad versus controlabilidad.

Con la variedad de problemas de generación de contenido y sus métodos disponibles, es de gran ayuda el tener una estructura que muestre las diferencias y similitudes entre los acercamientos a estos, por lo que a continuación se hablara de las taxonomías de PCG, las cuales están listadas en [Shaker *et al.*, 2016]:

- **Online versus offline** - Las técnicas de PCG se pueden usar para generar contenido online mientras el jugador está jugando el juego, permitiendo que se genere contenido con infinitas posibilidades, haciendo que el juego tenga una re-jugabilidad infinita y abriendo la posibilidad de generar contenido adaptativo. Por otro lado, se puede usar para generar contenido offline durante el desarrollo del juego o antes de empezar una sesión del juego. El uso del PCG offline es particularmente útil para la generación de contenido complejo como lo son el crear el ambiente o los mapas.
- **Necesario versus opcional** - PCG puede ser usado para generar contenido necesario para completar niveles en un juego, o puede ser usado para generar contenido auxiliar que puede luego ser descartado o intercambiado por otro contenido. La mayor diferencia en estos dos ámbitos es que el necesario debe siempre estar correcto mientras que el opcional no.
- **Grado y dimensión de control** - La generación de contenido creado por PCG puede ser controlado de distintas maneras. El uso de una semilla aleatoria es una forma de ganar control sobre la generación de espacio, otra forma es usar un set de parámetros para obtener control sobre un número de dimensiones.
- **Genérico versus adaptativo** - La generación de contenido genérica se refiere al paradigma del PCG donde el contenido es generado sin tener en cuenta el comportamiento del jugador. Al contrario el contenido adaptativo generado es personalizado o centrado en el jugador de manera que cuando el jugador interactúa con el juego, esto es analizado y el contenido es creado en base a sus comportamientos anteriores.

- Estocástico versus determinista - El PCG determinista permite la re-generación del mismo contenido dado el mismo punto de partida y los mismos parámetros, mientras que en el PCG estocástico la re-creación de un mismo contenido normalmente no es posible.
- Constructivo versus generar y probar - En el PCG constructivo, el contenido es generado en una sola iteración, como es normalmente hecho en juegos tipo roguelike². Por otro lado, las técnicas de PCG de generar y probar, alternan en generar y probar en un ciclo, repitiendo hasta que se generan soluciones satisfactorias
- Generación automática versus autoría mixta - Hasta hace relativamente poco tiempo, PCG permitía una entrada limitada de parte del diseñador del juego, quien normalmente ajusta los parámetros del algoritmo para controlar y guiar el contenido generado mientras que el propósito principal del PCG se encuentra en la generación de variaciones infinitas de contenido jugable. Sin embargo, ha aparecido un nuevo paradigma interesante que se centra en incorporar diseño y/o datos de entrada del jugador a través del proceso de diseño. En este paradigma de iniciativa mixta, el diseñador o el jugador cooperan con el algoritmo para generar el contenido deseado.

Si bien existen varias dimensiones, estas no son excluyentes, los juegos en los que se utilizan no deben necesariamente estar dentro de una sola categoría, de hecho es normal que estén en más de una, por ejemplo el juego "Left 4 Dead" (Valve Corporation, 2008) usa PCG adaptativo para ajustar la dificultad según cómo juega la persona lo cual lo hace de manera online, es decir mientras está jugando [Booth, 2009].

2.3. Algoritmos

En esta sección se describen los algoritmos más usados en la generación de mapas utilizando PCG y al final se mencionan algunos ejemplos de que otras cosas se pueden hacer usando PCG.

El PCG utiliza algoritmos que en un principio no estaban pensados para usarse en videojuegos y realizar su generación de contenido, uno de los primeros reconocidos es el de Perlin Noise y el algoritmo Diamond-square [Pedreño Moya, 2014] que genera un mapa de alturas, pero existen muchos otros que se pueden usar para la generación de estos, como se menciona en [van der Linden *et al.*, 2014], los algoritmos mencionados en él se enfocan en generación de dungeons, se basan en la dimensión y grado de control y se separan en categorías con distintos acercamientos, entre los cuales se encuentran:

- autómatas celulares: [Johnson *et al.*, 2010] Un autómata celular es un modelo matemático para un sistema dinámico que evoluciona de manera discreta. Un autómata

²Juegos de un jugador de mazmorras generadas aleatoriamente en donde suele haber muerte permanente, es decir, una vez que el jugador muere, se debe empezar desde el principio.

celular consiste en una grilla de celdas, en donde cada una tiene un número finito de estados, por ejemplo prendido y apagado.

El objetivo de los juegos en los que se usa autómatas celulares es el de avanzar por el mapa hasta llegar a un objetivo, usualmente bajar o subir un piso o salir de éste, donde el mapa en estos juegos suele tener un enfoque de laberinto, pero sin ser estrictamente uno.

En el caso del algoritmo, este consiste en una cuadrícula de celdas con dimensiones finitas, en donde cada celda tiene una referencia a las celdas a su alrededor (las cuales son su vecindario) y parten con un estado inicial en $t=0$. Para calcular el estado en las siguientes iteraciones se aplican un número de reglas a la celda actual y a sus vecinos. Luego de varias iteraciones se genera un mapa en la cuadrícula el cual depende del tipo de reglas que se utilicen y el estado inicial de la cuadrícula.

Por lo tanto, la representación de autómatas celulares consiste en una cuadrícula de celdas y sus estados, en el caso de [Johnson *et al.*, 2010] se utilizaron 3 estados: Piso, muralla, roca. En donde uno se puede mover por el piso, pero no por las paredes y rocas.

En este caso, se parte cambiando de manera aleatoria un porcentaje de las celdas de piso a roca, respecto a la cantidad que haya de ambas y parte el algoritmo, utilizando 2 reglas:

- Una celda es una roca si el valor del vecindario es mayor o igual a T ($T=5$)
- Una celda roca que tiene un vecino piso es una celda muralla

Se usaron 4 parámetros para poder controlar la generación de mapas y son:

- r : Porcentaje inicial de rocas
- n : Número de iteraciones
- T : Límite de vecindario que define a las rocas
- M : Número de vecinos

Si bien el tener pocos parámetros puede parecer algo bueno, también es su debilidad por ser poco personalizable. Con este algoritmo se generan mapas como el que se muestra en la figura 5. Acá se puede observar que a pesar de ser poco personalizable, se pueden crear mapas bastante interesantes y diversos para un videojuego al no tener una forma "cuadrada" o con figuras simétricas como suelen ser los mapas de los videojuegos normalmente.

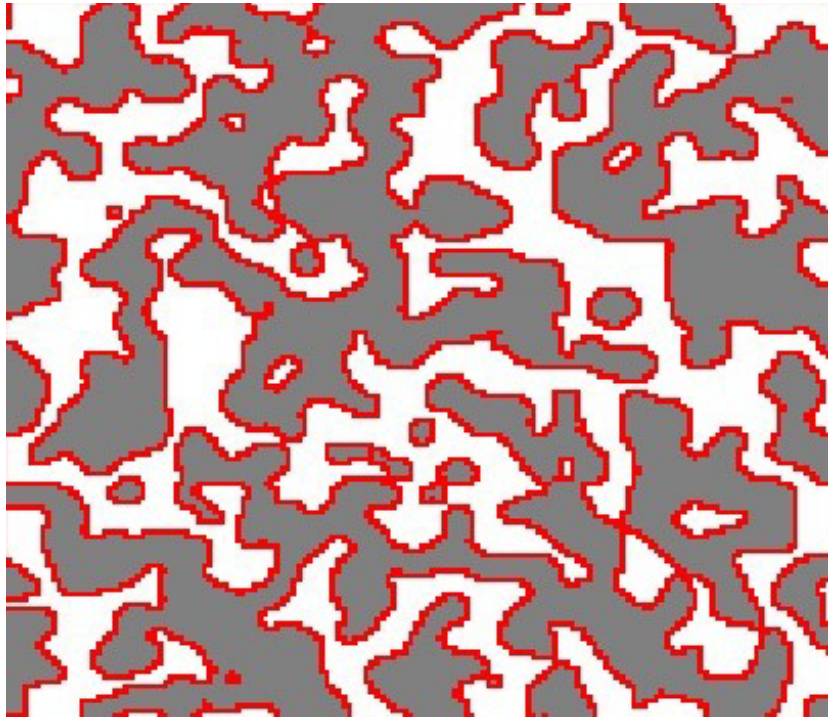


Figura 5: Mapa generado con autómatas celulares.

Las rocas y murallas son representadas con rojo y blanco respectivamente. El gris corresponde a áreas de piso.

Fuente: [Johnson *et al.*, 2010].

- Generative grammar: [Van der Linden *et al.*, 2013] Este tipo de algoritmo originalmente se basa en la creación de frases a través de una selección finita de una lista recursiva de reglas transformacionales, las cuales contienen palabras como símbolos terminales. En base a esta gramática generativa, nacieron otros tipos de gramática, como por ejemplo de grafos, forma e incluso de jugabilidad.

Por lo general, todos los tipos sirven, hay mapas generados con grafos, con forma e incluso con una combinación de ellos, en el caso de [Van der Linden *et al.*, 2013] los diseñadores tienen que especificar, a priori, las restricciones de diseño en un vocabulario orientado a la jugabilidad, las cuales consisten en las acciones que el jugador puede realizar, sus secuencias, relaciones y contenido, por ejemplo: pelear con un dragón. Estas restricciones resultan directamente en una gramática de jugabilidad (grafo de jugabilidad), luego se utilizan algoritmos para pasar el grafo a un mapa de juego y por último se utiliza un método procedural para generar la geometría de las salas y los pasillos.

Lo bueno de este algoritmo es que al ser genérico, se puede usar para todo tipo de videojuegos ya que está basado en la jugabilidad y no en el juego en sí. Con este algoritmo se generan grafos como los de la figura 6 y mapas como los de la figura 7 a partir del grafo. En el grafo de la figura 6 se pueden ver las acciones que el jugador puede realizar, partiendo por recorrer el mapa o pelear contra un enemigo para así poder

obtener una llave y abrir la puerta para seguir avanzando con las siguientes acciones claves, con el fin de poder ganar el juego. Luego, como se dijo anteriormente, se utiliza un algoritmo para pasar este grafo a un mapa y se utiliza un método procedural para generar las salas y pasillos que se pueden observar en la figura 7. En este tipo de mapas lo importante es la distribución que tienen ya que deben cumplir con los pasos a seguir en el grafo, es decir, no puede estar la sala de la puerta antes que la sala en donde se obtiene la llave para abrirla porque haría injugable el juego al no poder avanzar en él.

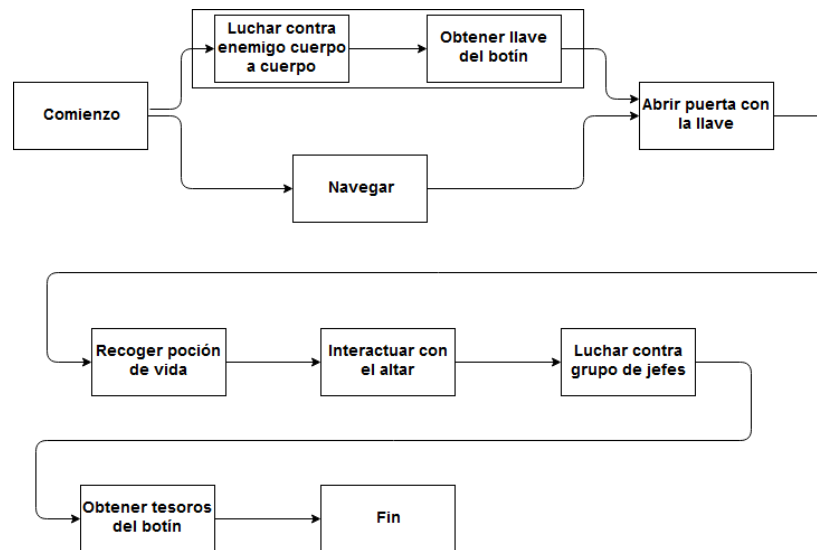


Figura 6: Grafo de jugabilidad generado con Generative grammar.
Fuente: Traducido de [Van der Linden *et al.*, 2013].

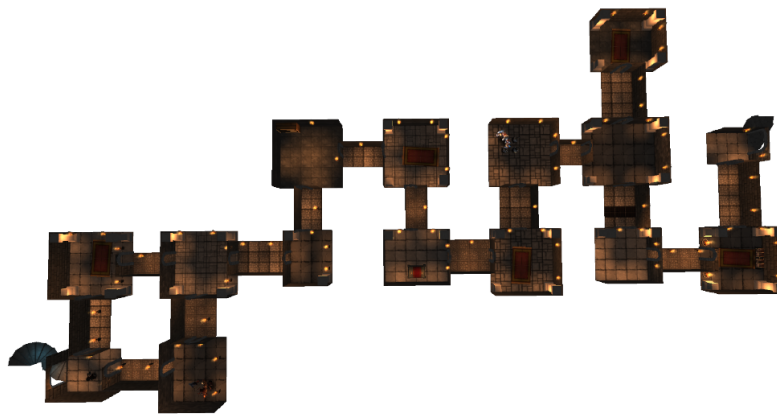


Figura 7: Mapa generado a partir del grafo de la figura 6.
Fuente: [Van der Linden *et al.*, 2013].

- Genetic algorithm: [Valtchanov y Brown, 2012] Este tipo de algoritmo son algoritmos evolutivos basados en la búsqueda de una solución óptima a un problema de optimización. Los algoritmos genéticos necesitan de una representación genética (usualmente

llamada genes o cromosomas) para representar las posibles soluciones y una función de aptitud para medir que tan buenas son las soluciones obtenidas. El algoritmo se usa iterativamente, en donde en cada iteración se calcula la aptitud de las soluciones obtenidas y luego se utilizan dos métodos para alterar las soluciones, una se llama mutación y se encarga de hacer pequeños cambios dentro de una misma solución. El otro método se llama cruzamiento y lo que hace es tomar partes de dos soluciones y mezclarlas para crear una nueva solución (crea un cromosoma hijo).

Estos métodos usualmente seleccionan los cromosomas según un puntaje de aptitud que tiene cada uno el cual se basa en la capacidad de cada cromosoma de solucionar el problema. Estos métodos permiten tanto la exploración como la explotación del espacio de búsqueda.

Este proceso continua hasta que haya pasado cierto tiempo o se haya obtenido cierto nivel de aptitud.

En [Valtchanov y Brown, 2012] usaron algoritmos genéticos para crear mapas de 'dungeon crawler'. En este trabajo usaron una estructura de árbol para representar a los cromosomas, en donde los nodos son las habitaciones, de las cuales existen dos tipos, habitaciones comunes y corrientes y habitaciones en donde ocurre un evento del juego, y las conexiones con los hijos del árbol son los pasillos como se puede ver en la figura 8.

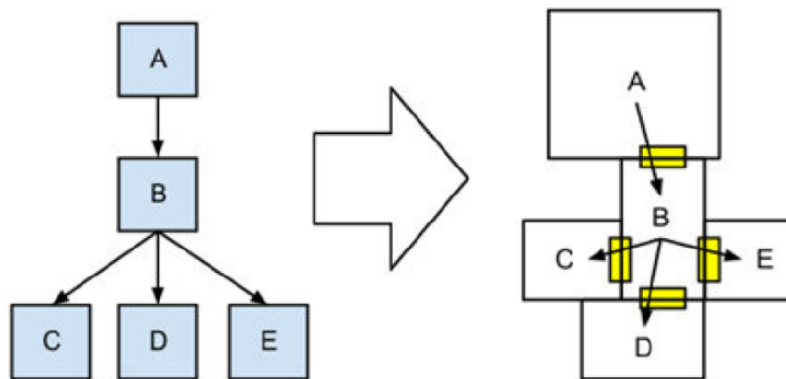


Figura 8: Estructura del árbol traducida a estructura del mapa.

Fuente: [Valtchanov y Brown, 2012].

Para el método de cruzamiento se intercambiaron de manera aleatoria dos sub-ramas entre dos copias de los padres. Para el método de mutación se utilizaron tres formas distintas en las que un árbol puede ser modificado: Hacerlo crecer, recortarlo y cambiarlo (Seleccionan X nodos y les cambian valores, los cuales son los valores de las puertas (si se puede abrir o no) y asigna un tipo de celda aleatoriamente). El método de cruzamiento tenía un 70 % de probabilidad de ser aplicado mientras que los tres de mutación cada uno tenía un 50% de probabilidades y el de cambiar se aplicaba siempre si ningún otro método se aplicaba en la iteración.

Para la selección se utiliza una selección por torneo en donde divide a los cromosomas de manera aleatoria.

Para la función de aptitud se utilizó uno que promovía habitaciones pequeñas y bien agrupadas que están conectadas eficientemente por pasillos con hasta tres habitaciones especiales situadas cerca del perímetro. Con este algoritmo se logran crear mapas como el de la figura 9. En la figura se pueden apreciar las habitaciones especiales, las cuales están con colores distintos y son especiales porque en ellas hay eventos del juego. Los parámetros utilizados son: 2000 generaciones y un máximo de 500 habitaciones, la cual no se alcanzó por el pequeño espacio que se utilizó.

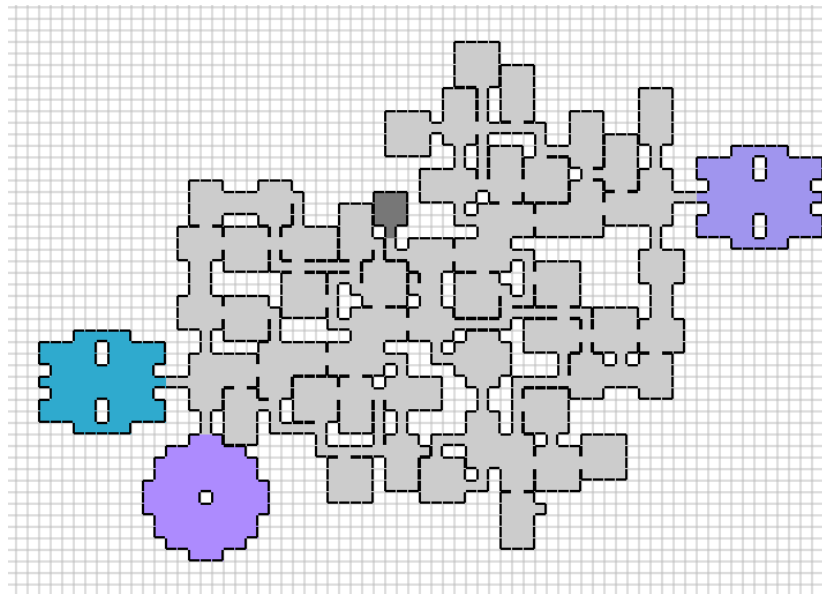


Figura 9: Ejemplo de mapa generado con algoritmo genético.
Fuente: [Valtchanov y Brown, 2012].

- **Constraint-based:** [Roden y Parberry, 2004] Este tipo de algoritmo, como su nombre lo indica, se basa en restricciones, las que, por lo general, son representadas como reglas con ciertos parámetros, por ejemplo la distancia entre nodos en un grafo.

En [Roden y Parberry, 2004] primero crearon un grafo 3D indirecto para representar la estructura del mapa en donde los nodos representan una porción de la geometría del nivel. Para generar este grafo se utilizaron restricciones en la topología y propiedades en los nodos (como distancia y adyacencia entre ellos). Algunas de las topologías clásicas que se usaron fueron árboles, anillos y estrellas, con las cuales luego se juntaron para crear topologías híbridas. Luego usando geometría pre-fabricada, se renderizó cada nodo. La entrada que se utilizó para el algoritmo fue el de un grafo de una topología inicial y un conjunto de reglas tales como mínima/máxima distancia entre nodos terminales, conexiones fijas entre nodos adyacentes y restricciones para hacer que el jugador visite los nodos en cierto orden.

Con este algoritmo se logró crear mapas como el de la figura 10. Se puede notar en la figura que esta es 3D, teniendo una dimensión de profundidad que no tienen las otras técnicas y ejemplos vistos hasta ahora.

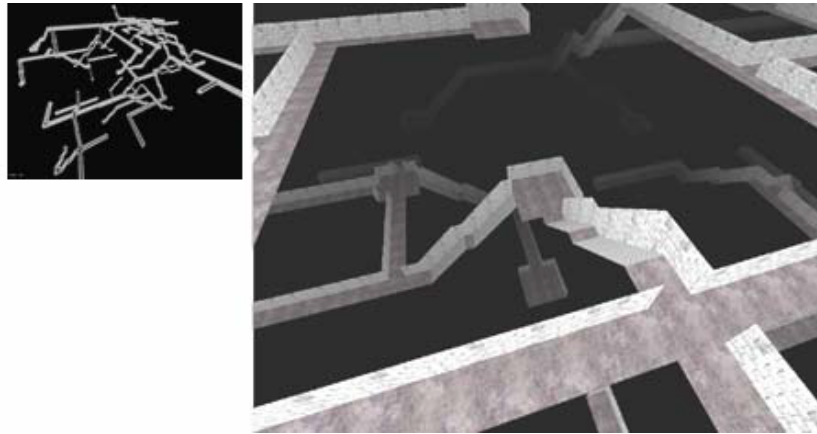


Figura 10: Geometría de mapa 3D (derecha) a partir de un grafo generado (izquierda)
Fuente: [Roden y Parberry, 2004].

Por otro lado, en [Buck, 2003] se propone un generador de dungeon aleatorio, el cual utiliza como base un algoritmo de generación de laberintos (de los cuales existen varios, el que ellos usan se llama Hunt and Kill [H-a, 2009]) para luego utilizar ese resultado como base para su propio algoritmo para generar el dungeon. Este consta de dos pasos principales. Lo primero es revisar cada celda en el laberinto, si la celda es el final de un pasillo sin salida, se borra la celda. Esto se repite un número de veces según un parámetro llamado *escasez*. El siguiente paso es remover las celdas sin salidas agregando ciclos al laberinto, lo cual se hace según un parámetro llamado *'deadends removed'* el cual es un porcentaje para indicar si una celda sin salida debe ser removida o no, por cada celda sin salida, se analiza la probabilidad de si se remueve la celda o no.

Luego de crear el calabozo, se agregan las habitaciones, siguiendo 3 parámetros:

- Rn: Número de habitaciones
- Rw: Ancho máximo de las habitaciones
- Rh: Largo máximo de las habitaciones

Con estos parámetros, lo que hace el algoritmo es tratar de colocar habitaciones en cada espacio posible, en donde se superpongan el menor número de pasillos, pero que pase por al menos uno. El pseudocódigo para agregar las habitaciones se muestra en el algoritmo 1. Con este algoritmo se pueden obtener mapas como el de la figura 11. En la figura, los cuadrados pequeños en las intersecciones entre habitaciones y pasillos son puertas, la puerta de la habitación 6 está escondida, la cual está representada por un semicírculo negro. Es importante

mentonar esto ya que denota la capacidad del algoritmo de lograr colocar distintos tipos de puerta de manera satisfactoria.

Algoritmo 1 Random Dungeon

- 1: Asignar al puntaje "mejor" un valor de infinito (o un número muy alto)
 - 2: Generar una habitación tal que $Ancho_{min} \geq Rw \geq Ancho_{max}$ y $Altura_{min} \geq Rh \geq Altura_{max}$
 - 3: **for** Por cada celda C en el dungeon hacer **do**
 - 4: Colocar la esquina superior izquierda de la habitación en C y colocar el puntaje "actual" en 0.
 - 5: Por cada celda de la habitación que es adyacente a un pasillo, agregar 1 al puntaje actual.
 - 6: Por cada celda de la habitación que se sobrepone con un pasillo, agregar 3 al puntaje actual.
 - 7: Por cada celda de la habitación que se sobrepone a otra habitación, agregar 100 al puntaje actual.
 - 8: Si el puntaje actual es menor que el mejor puntaje, asignar al mejor puntaje el puntaje actual y asignar C como la mejor posición hasta ahora.
 - 9: **End for**
 - 10: Colocar la habitación en la mejor posición (donde se encontró el mejor puntaje)
 - 11: Por cada lugar donde la habitación es adyacente a un pasillo o habitación, añadir una puerta (Si no se quieren tener tantas puertas, se puede añadir un parámetro para determinar si se coloca una puerta o no).
 - 12: Repetir los pasos del 2 al 11 hasta que todas las habitaciones sean colocadas.
-

Cabe mencionar que la mayoría de estos algoritmos recién mencionados se utilizan para generar dungeons en 2D y casi ninguno en 3D. También existen otras categorías que si bien no generan dungeon como ellos los definen, pueden ayudar a la generación de estos, como lo son el uso de información del mundo real utilizando machine learning [Merrell *et al.*, 2010], en donde generan diseños 3D de edificios residenciales, enfocándose en los planos de la planta, por ejemplo, la organización interna de espacios dentro del edificio. Otra categoría es usando un híbrido entre Answer Set Programming (ASP) y Evolutionary Search [Togelius *et al.*, 2012] en donde el ASP se utiliza como el mapeo del genotipo al fenotipo de un algoritmo de Evolutionary Search, entre otros.

2.4. Otras áreas

Si bien se habla de generación de mapas para dungeons, los PCG no son exclusivos ni para la creación de dungeons ni para la de mapas. Se pueden utilizar para generar otros elementos, por ejemplo en [Pedreño Moya, 2014] se habla sobre cómo utilizar este tipo de algoritmos para generar mapas de ciudades en un JRPG (RPG japonés). En [Bhojan y Wong, 2016]

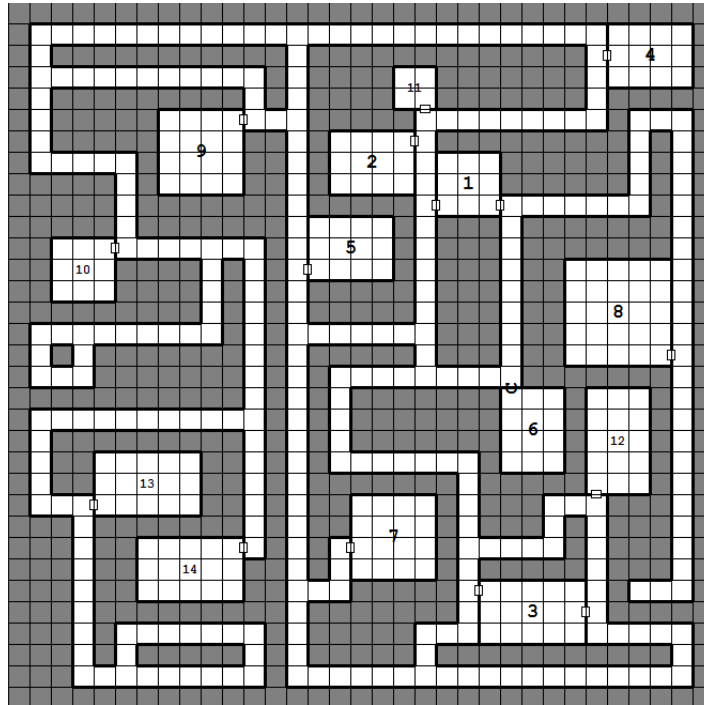


Figura 11: Ejemplo de un dungeon aleatorio generado.
Fuente: Resultado obtenido al hacer correr el programa de [Buck, 2003].

se habla sobre cómo generar mapas para juegos FPS de una manera innovadora. En [Cabrera *et al.*, 2015] sobre cómo utilizar PCG para mejorar la IA y la jugabilidad de un juego RTS, también se puede utilizar el PCG para generar el contenido del juego, como los enemigos, los objetos, etc. [Amato, 2017].

Los algoritmos aquí descritos no son los únicos que existen, hay algoritmos de varias categorías, por ejemplo para generar mapas, de conceptos, de generación de secuencias, Ontogenetic (Duplican el resultado final de un proceso físico sin emular los pasos intermedios) [Ont, 2009] y los Teleological (Simular un proceso físico que resulta en un output procedural deseado) [Tel, 2009], entre otros. Algunos ejemplos se pueden encontrar en [alg, 2014]

Resumen del capítulo

En este capítulo se vio el estado del arte del PCG, entre los temas importantes que se hablaron se encuentran algunas de las aplicaciones que son consideradas PCG y el inicio del PCG, que fue alrededor de 1978 con el juego Beneath Apple Manor. Se habló de las propiedades y taxonomías de los algoritmos PCG, los cuales son: Velocidad, confiabilidad, controlabilidad, expresividad y diversidad y por último creatividad y credibilidad, esto en el lado de las propiedades, las taxonomías son: Online versus offline, necesario versus opcional, grado y dimensión, genérico versus adaptativo, estocástico versus determinista, constructivo versus generar y probar y por último generación automática versus autoría mixta. Luego se habló

de los algoritmos más usados en la generación de mapas utilizando PCG, en donde los más destacables son: autómatas celulares, gramática generativa, algoritmo genético, basado en restricciones y la generación aleatoria de mapas. Por último, se habló de otras áreas en donde se puede usar PCG a parte de mapas de dungeons, como por ejemplo mapas de ciudades, mapas de juegos FPS, para mejorar la IA del juego, etc.

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

En el presente capítulo se expone la solución propuesta para resolver el problema de generación automática de mapas. Como algoritmo para el PCG se utilizó un GA (Genetic Algorithm - Algoritmo Genético) por el tipo de control que se tiene sobre los parámetros y por el tipo de mapas que se pueden lograr construir, como los de la figura 9. Este tipo de algoritmo comienza con una población inicial de soluciones, en nuestro caso mapas, la cual va evolucionando a través de generaciones utilizando procedimientos de evaluación, elitismo, selección, cruzamiento y mutación.

3.1. Algoritmo genético

Los algoritmos genéticos son algoritmos evolutivos basados en la teoría de evolución de Darwin propuestos para resolución aproximada de problema de optimización. Los algoritmos genéticos necesitan de una representación genética (usualmente llamada genes o cromosomas) para representar las posibles soluciones y una función de aptitud (evaluación o calidad) para medir que tan buenas son las soluciones obtenidas durante el proceso.

El algoritmo se usa iterativamente, en donde para cada iteración se calcula la aptitud de las soluciones obtenidas y se selecciona un subconjunto de soluciones consideradas más aptas para la resolución del problema estudiado. Luego, se utilizan dos métodos para alterar las soluciones, un método unario llamado mutación y un método binario llamado cruzamiento. La mutación se encarga de hacer pequeños cambios dentro de una misma solución y promueve la diversificación en el proceso de búsqueda. El otro método se llama cruzamiento y lo que hace es tomar partes de varias soluciones, dos en el caso binario, y mezclarlas para crear nuevas soluciones. El propósito de este operador es intensificar el proceso de búsqueda. Este proceso continua hasta que se alcance un cierto criterio de término. Los criterios de término más utilizado se relacionan con: tiempo máximo, nivel de aptitud mínimo, convergencia, entre otros.

3.2. Propuesta

La propuesta aquí presente, está enfocada a videojuegos del genero de mazmorras, como se vio en la sección 1.2. Este tipo de videojuegos está enfocado en la exploración, por parte del jugador, de calabozos o zonas cerradas en donde el jugador debe lograr salir de estos calabozos, obteniendo objetos y venciendo enemigos en el camino. La importancia de los mapas en este tipo de juegos es crucial, ya que estos no deben ser muy abiertos, deben

poseer diferentes estructuras como paredes, trampas, enemigos, etc. y deben generar en el jugador diferentes retos en cada nivel. Así, la idea es que el jugador no sepa si se encontrara una trampa, un objeto, un enemigo o incluso la salida al doblar en una esquina, generando así una incertidumbre con cada paso que se da. Un punto importante de esto es que los mapas deben ser distintos cada vez que se juegan ya que de lo contrario el jugador se aprendería las ubicaciones de las trampas, objetos, enemigos y de la salida. Esto de forma tal que la sensación de incertidumbre que se genera la primera vez que se juega se mantenga a pesar de que el mismo jugador decida jugar el juego en repetidas oportunidades.

Así, este trabajo de memoria se propone un algoritmo de PCG, utilizando un algoritmo genético para crear de manera procedural mapas de videojuegos de tipo mazmorra. Esta propuesta está basada en la de [Valtchanov y Brown, 2012], pero en nuestro caso, se construyen las habitaciones completas generando las celdas del mapa, mientras que en su propuesta, se construyen mapas completos generando habitaciones enteras pre-construidas. Por otro lado, el proceso de selección y el modelo de aptitud usados son distintos. En su propuesta, el método de selección utiliza una selección por torneo en donde divide las soluciones de manera aleatoria, mientras que en esta propuesta se utilizó un método de selección por torneo en donde las soluciones con mejor aptitud, tienen más probabilidades de ser seleccionadas. Por otro lado, la función de aptitud en su propuesta promueve habitaciones pequeñas y bien agrupadas mientras que en esta propuesta, la función de aptitud promueve la utilización de celdas pedidas y la alcanzabilidad dentro del mapa (solo en la fase 2 y 3). Los operadores de cruzamiento y mutación son prácticamente los mismos, con ciertas variaciones, por ser los adecuados para un árbol. El pseudocódigo de la propuesta se presenta en el algoritmo 2.

Algoritmo 2 Genetic Algorithm

```
1: bestFitness = 99999
2: Generaciones = N
3: Inicialización()
4: while True do
5:   Evaluación()
6:   Elitismo()
7:   if Generaciones < 0 OR bestFitness == 0 then
8:     Break
9:   Selección()
10:  Cruzamiento()
11:  Mutación()
12:  Generaciones - -
13: TransformarSolucionAMapa()
```

Como se puede observar en la línea 3, lo primero que se hace es inicializar la población, lo cual hace que se obtengan X soluciones distintas usando aleatoriedad. En este paso también se inicializan los parámetros. Una vez construido el conjunto de soluciones iniciales, se comienza el proceso iterativo en la línea 4. En cada iteración, se evalúan las soluciones en la línea 5 y se elige la con mejor aptitud como la mejor solución hasta el momento en el paso

del elitismo en la línea 6. A continuación se revisa el criterio de parada en la línea 7. En caso de corresponder a la última generación o en caso de que se ha alcanzado una aptitud perfecta, se termina el ciclo, pasando a la línea 13. En caso de que no se cumpla ninguna de las dos condiciones de parada, se pasa a los procesos de selección, cruzamiento y mutación en las líneas 9, 10 y 11 respectivamente. En el proceso de selección se eligen las soluciones que serán padres de la siguiente generación. Dichos padres son posteriormente sometidos a los procesos de cruzamiento y mutación. Luego de esto, las nuevas soluciones son sometidas a la evaluación y elitismo en las líneas 5 y 6 nuevamente, repitiendo así el ciclo. Una vez terminado el ciclo, la mejor solución obtenida es traducida al mapa del juego en la línea 13 y termina el algoritmo.

En este trabajo de memoria, se estudiaron u evaluaron tres versiones o fases del algoritmo, para analizar sus ventajas y desventajas en la generación de soluciones para el problema estudiado.

A continuación se explicaran las componentes básicas comunes para las tres versiones del algoritmo: Representación, inicialización, operadores de selección y transformación, función de evaluación y elitismo.

3.3. Representación

El cromosoma utilizado para representar las soluciones es un árbol triario, es decir, con 3 hijos en cada nodo. Cada nodo posee la siguiente información:

- **type:** Tipo de celda, donde 0 es una celda con espinas, 1 es una celda normal sin paredes, 2 es una celda con 1 pared, 3 es una celda con 2 paredes, 4 es una celda con una puerta y 5 es la celda donde está la salida del mapa.
- **coordenadas:** Vector bidimensional de las coordenadas 'x' y 'z' de la celda.
- **wall1, wall2, wall3, door:** Dirección en la que se sitúa la pared o puerta en la celda, la pared número 3 solo se utiliza si la celda está al borde del mapa.
- **neighbors:** Corresponde a una lista con los 3 nodos hijos a los que este apunta: Left, Center y Right.

Cabe mencionar que todos los tipos de celdas pueden tener paredes si están al borde, de manera de poder cerrar el mapa y solo existen 3 paredes como máximo por celda, ya que si tuvieran 4 no se podría acceder a ellas por ninguna dirección. Por otro lado, las puertas al ser colocadas en las mismas posiciones en donde se colocan las paredes, puede pasar que una pared y una puerta se coloquen en la misma posición, creando una puerta trampa, es decir, una puerta que no se puede abrir.

Para decodificar el mapa del juego a partir de la representación, se recorre el árbol en sentido pre-orden, es decir procesando el nodo actual y luego el de su izquierda, centro y por último el de la derecha. En cada paso se guardan las coordenadas visitadas, de manera que si hay más de un nodo con las mismas coordenadas, solo el primero se procesa. Esto conlleva a que los nodos de más a la izquierda tienen preferencia sobre los de más a la derecha.

En la figura 12 se muestra un ejemplo de cómo se recorre el árbol en sentido pre-orden. Las flechas verdes indican el flujo de lectura del procesamiento. En cada nodo se muestran las coordenadas en el mapa para una mejor visualización de la traducción árbol-mapa. Se

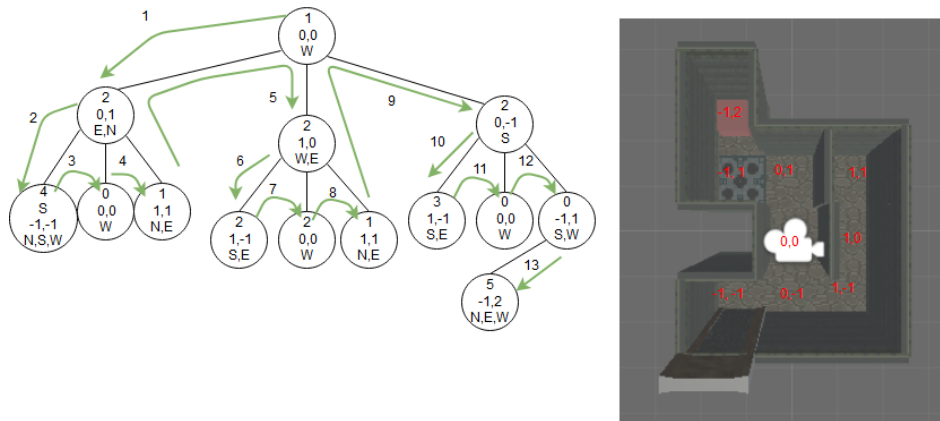


Figura 12: Ejemplo de decodificación de un árbol en un mapa.
Fuente: Elaboración propia.

puede apreciar en la figura que el primer nodo visitado es la raíz y es de tipo 1 y se ubica siempre en la coordenada (0,0). A continuación, siguiendo el orden de las flechas verdes, en el paso 1 se visita el nodo tipo 2 que se ubica en la coordenada (0,1). Luego en el paso 2 se visita el nodo tipo 4 que se ubica en la posición (-1,-1). Al ubicar el nodo siguiente (paso 3) se puede observar que el hijo del centro del nodo de la izquierda, es de tipo 0, espinas, pero este no es procesado ya que tiene las mismas coordenadas que el primer nodo visitado, el (0,0). El proceso continua hasta visitar el árbol completo, procesando las celdas que tengan la primera aparición de cada coordenada.

3.4. Inicialización

Como el GA necesita de una población inicial para trabajar, se crea una cantidad de soluciones igual al parámetro **Población inicial**. Cada una de las soluciones se genera de manera independiente de las otras para fomentar la diversidad del conjunto de soluciones inicial.

Primero que todo, se tiene en cuenta el parámetro **Tamaño Solución** para decidir cuál será la profundidad del árbol, el único nodo fijo es la raíz del árbol que está en las coordenadas (0,0) y es de tipo 1, ya que el jugador comienza en ese punto.

Por cada nodo, se elige un tipo al azar entre 0 y 4, donde 0 corresponde a una celda con espinas, 1 corresponde a una celda normal, 2 corresponde a una celda con una pared, 3 corresponde a una celda con dos paredes y 4 a una celda con una puerta. Solo existe uno nodo salida. Luego se escoge una dirección al azar y se colocan los tres hijos alrededor del padre, agregando las paredes si son del tipo 2 o 3 con direcciones aleatorias al igual que la dirección de la puerta si es de tipo 4. Por ejemplo la dirección seleccionada es Norte, Left se coloca un espacio arriba del padre, Center se ubica un espacio a la derecha del padre y Right se ubica un espacio abajo del padre. Este proceso se repite recursivamente hasta tener un árbol completo con profundidad **Tamaño Solución** - 1.

Luego de creada la población inicial, se agrega una salida a cada solución, en alguna coordenada que no se repita y se agrega al final del hijo más a la izquierda posible. La salida se podría colocar en cualquiera otra posición del árbol, pero se decidió esa posición por ser fácil de encontrar al recorrer el árbol. Finalmente se cierra el mapa, agregando paredes a las celdas del borde del mapa.

En la figura 13 se puede observar un ejemplo de cómo se va creando una solución paso a paso hasta agregar la salida, notar que no se agregó el paso de cerrar el mapa para no extender la figura más de lo deseado, pero los nodos mostrados ya contienen el paso de cerrar el mapa. A la derecha se muestra un ejemplo del mapa generado luego de terminar todo el proceso de inicialización. Se puede observar en la figura el paso a paso de la inicialización de la solución, primero se crea el nodo raíz, que siempre es una celda normal en la posición (0,0). Luego se crean sus tres hijos con datos aleatorios según la dirección que se generó también aleatoriamente. En el ejemplo de la figura 13 se generó la dirección sur, por lo tanto el nodo de la izquierda se coloca un espacio abajo del padre, el del centro un espacio hacia la izquierda y el de la derecha un espacio arriba. Estos nodos se muestran en la figura como las celdas de coordenadas (0,-1), (-1,0) y (0,1) respectivamente.

Luego, como esto se genera de manera pre-orden, el algoritmo se posiciona en el nodo de la izquierda (correspondiente a la celda (0,-1)) y vuelve a generar tres hijos con datos aleatorios posicionándolos alrededor de su padre según la dirección que se haya generado en esa iteración. A continuación se posiciona en el hijo de la izquierda (correspondiente a la celda (1,-1)) pero como ya se alcanzó la profundidad de **Tamaño Solución** - 1 no genera hijos y se mueve al hijo del centro. Al pasar lo mismo se posiciona en el hijo derecho y vuelve a ocurrir la misma situación, por lo que el algoritmo termina posicionándose en el primer hijo del centro y en este generan sus correspondientes tres hijos siguiendo la misma lógica de los otros casos. Con este paso se generan las celdas (-1,1), (0,0) y (-1,-1). Una vez creado el árbol completo, se procede a agregar la salida como el hijo más a la izquierda del árbol, para pasar así cerrar el mapa correspondiente y terminar la inicialización.

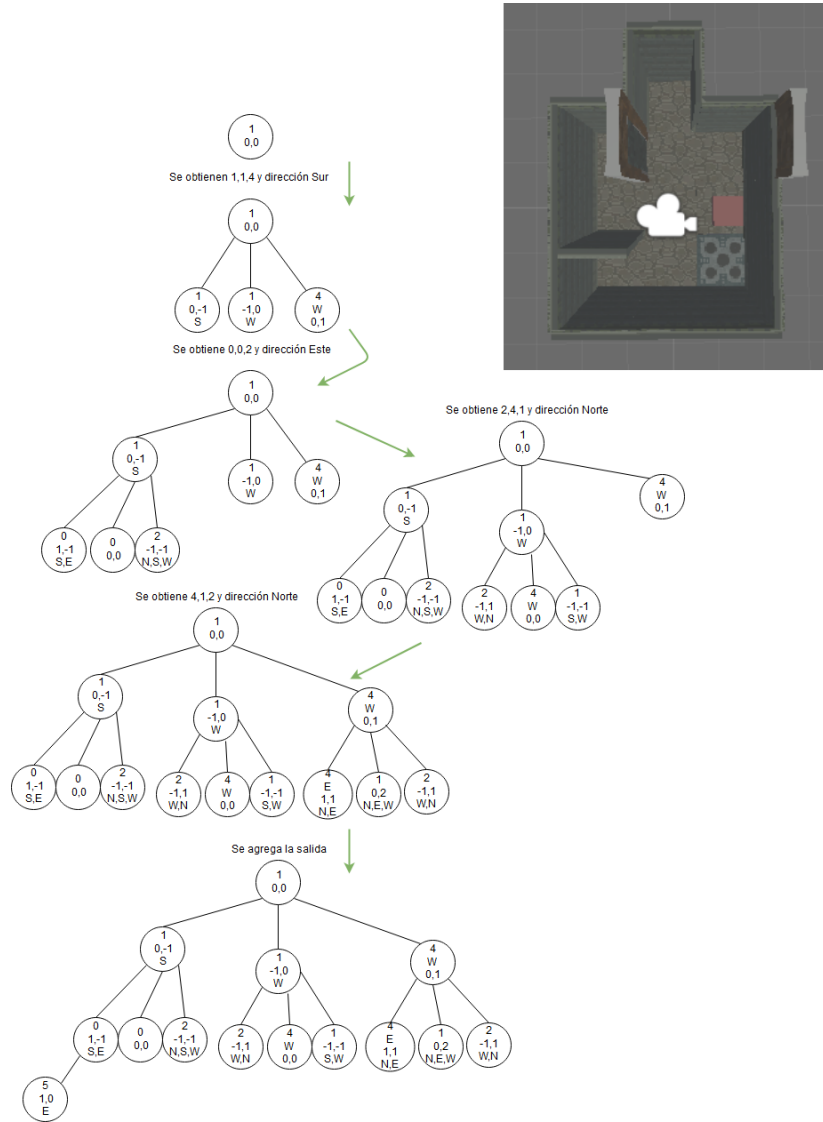


Figura 13: Ejemplo de Inicialización de una solución.
Fuente: Elaboración propia.

3.5. Selección

El operador de selección que se utilizó es el operador de selección basado en ruleta. El operador asigna a cada solución una probabilidad de ser seleccionado proporcional a su función de evaluación.

En cada paso del proceso de selección se seleccionan N nuevos padres de la población actual. La ecuación 1 muestra la expresión para calcular la probabilidad de selección de cada solución.

$$P_i = \frac{\maxScore + \minScore - score_i}{\sum_{i=1}^N score_i} \quad (1)$$

Cabe notar que si una solución es infactible, tiene probabilidad nula de ser escogida. En el caso de que se esté en la primera generación y solo se obtengan soluciones infactibles, no hay cambios en la selección y se usaran los mismos 3 padres para los pasos de cruzamiento y mutación.

3.6. Cruzamiento

El operador de cruzamiento genera nuevas soluciones (hijos) al intercambiar información presente en las soluciones actuales (padres). El operador de cruzamiento que se utilizó es el de cruzamiento en un punto (one point crossover). Este operador intercambia una rama de un padre con otra en la misma ubicación de otro padre desde un punto entre 1 y **Numero Crossover** -1, es decir, si **Numero Crossover** es 4, se intercambiarán las ramas en un punto entre los puntos 1 y 3 de ambos padres. La manera de elegir entre Left, Center o Right es aleatoria a la hora de ir moviéndose en los padres para llegar al punto de cruzamiento.

Al igual que en [Valtchanov y Brown, 2012], la ventaja de usar este tipo de cruzamiento, es que se intercambian sub-ramas completas entre soluciones, lo que genera la posibilidad de que celdas más apropiadas sean colocadas en las coordenadas usadas.

Este operador se ejecuta con una probabilidad igual al **Pc** escogido en el experimento.

En la figura 14 se puede ver un ejemplo de cruzamiento. En la figura 14 se puede observar que se hizo un cambio en el nivel 2 de las soluciones, en donde se cruzaron los hijos del centro. Como ahora hay un nodo (1,1) más a la izquierda en el árbol de la izquierda, se procesa primero ese nodo, haciendo aparecer una puerta en el mapa. Solo se mostró el lado izquierdo de las soluciones por temas de espacio.

3.7. Mutación

El operador de mutación, a diferencia del de cruzamiento, genera nuevas soluciones mediante cambios unarios a las soluciones actuales.

En esta propuesta, se utilizó como operador de mutación el de cambiar una cierta cantidad de nodos definida entre 1 y **Numero Nodos Mutación**. En nuestra propuesta se cambia el tipo de celda a cualquier otro tipo de celda, excepto la celda donde se encuentra la salida (la cual tampoco es mutable) y se reinician las direcciones de las paredes, agregando direcciones aleatorias si se cambió a una celda tipo normal con una pared o con dos paredes.

Para la elección de que celda cambiar, se utilizó un modelo ruleta, en donde las celdas que les falta una mayor cantidad para llegar a la cantidad deseable tienen mayor probabilidades de ser escogidas que las celdas que se pasaron de la cantidad deseable.

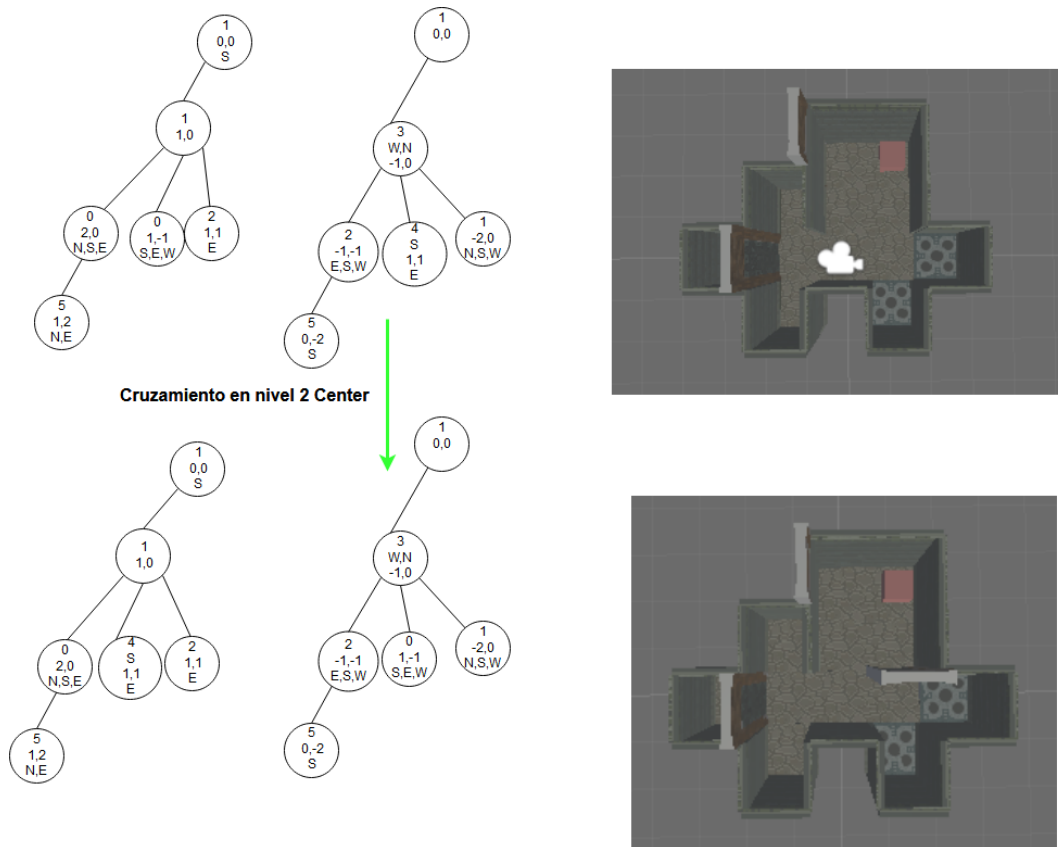


Figura 14: Ejemplo de cruzamiento
Fuente: Elaboración propia.

Este operador se ejecuta con una probabilidad igual al **Pm** escogido en el experimento, y si no se ejecutó el paso de cruzamiento, este se realiza obligatoriamente. La idea de esto es obligar cambios entre generaciones.

En la figura 15 se puede ver un ejemplo de mutación. En la figura 15 se puede observar que el operador realizó una mutación en el primer hijo de la izquierda, cambiando su tipo de celda de normal (El 1 que se ve en el nodo) a una de espinas (El 0 que ahora se ve en el nodo). Este cambio se puede observar en los mapas de la derecha, donde aparece una celda de espinas en la posición (1,0) en la que antes solo había una celda normal.

3.8. Función de Evaluación

Los GA favorecen las soluciones que tienen una aptitud alta (en el caso de maximización) y una aptitud baja (en el caso de minimización), el cual es obtenido en una función de evaluación. En nuestro acercamiento se trabaja con soluciones factibles. A continuación se describen los criterios considerados para medir la factibilidad y calidad de las soluciones.

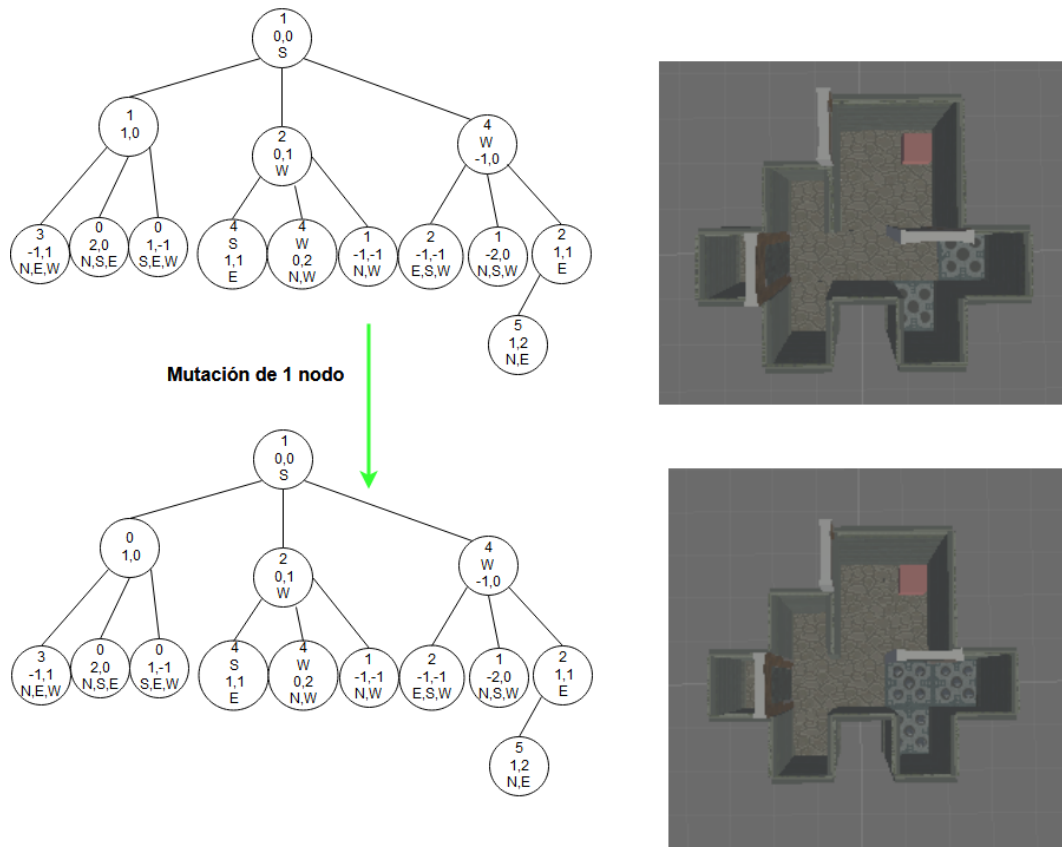


Figura 15: Ejemplo de mutación.
Fuente: Elaboración propia.

3.8.1. Factibilidad

La única restricción que tiene un mapa para ser factible, es que desde donde parte el jugador, es decir, desde las coordenadas (0,0), se pueda llegar a las coordenadas de la salida, en otras palabras, se debe poder ganar el nivel.

Para lograr esto se almacena el diseño del mapa, en donde cada coordenada tiene asociada la información de si existe una pared o no en cada dirección (Norte, Sur, Este, Oeste), luego se recorre el diseño del mapa obtenido usando una función con backtracking recursivo, marcando por donde se ha pasado, para evitar ciclos. Si la función logra llegar a las coordenadas de la salida, la solución es factible.

A continuación se muestra un pseudocódigo en el algoritmo 3 para explicar la función de factibilidad:

En la primera línea se crea un diccionario que contendrá por cada coordenada en el mapa 4 direcciones asociadas, norte, sur, este y oeste, y en LayoutMap() estas se rellenan con true o false para indicar si existe una pared o se puede pasar en esa dirección. Luego se usa este

Algoritmo 3 Función de factibilidad

```
1: Diccionario = new Dictionary<coordenadas, Lista(bool) >()
2: Diccionario = LayoutMap()
3: x = z = 0
4: Factible = BackTrackingFeasibleSolution(Diccionario, coordenadasExit,x,z, direccion,
    coordenadasUsadas)
```

diccionario junto a las coordenadas de la salida de la solución, las coordenadas x y z para recorrer el mapa, partiendo en (0,0), una dirección, partiendo sin dirección (denotada por un 4 en el algoritmo) y una lista con las coordenadas usadas, partiendo con la lista vacía.

En el algoritmo 4 se muestra un pseudocódigo de esta función:

Algoritmo 4 BackTrackingFeasibleSolution

```
1: if (x,z) == coordenadasExit then
2:     return 1
3: posicion = (x,z)
4: if posicion is not in coordenadasUsadas then
5:     add (x,z) in coordenadasUsadas
6: else return 0
7: if direccion != 4 then
8:     Diccionario[posicion][direccion] = True
9: for Por cada direccion i do
10:    if diccionario[posicion][i] == false then
11:        diccionario[posicion][i] = True
12:        if i == 0 then
13:            moverse 1 espacio hacia arriba y llamar a la función
14:        else
15:            if i == 1 then
16:                moverse 1 espacio a la derecha y llamar a la función
17:            else
18:                if i==2 then
19:                    moverse 1 espacio hacia abajo y llamar a la función
20:                else
21:                    if i == 3 then
22:                        moverse 1 espacio hacia la izquierda y llamar a la función
23: return 0
```

Lo que hace esta función es revisar si actualmente se encuentra en las coordenadas de la salida en la línea 1 y 2, en caso contrario, se guardan las coordenadas como posición y se revisa si no se ha pasado por esta posición, en caso afirmativo, se guardan las coordenadas en las línea 4 y 5, en caso contrario se termina el algoritmo ya que significa que pasamos por todas las coordenadas posibles, luego, en la línea 7 se revisa si aún quedan direcciones para

moverse y se agrega en el diccionario que se pasó por esa dirección, luego en el for de la línea 9 se revisa de norte a oeste en sentido horario si se puede avanzar en esa dirección, si por ejemplo se puede avanzar al norte, se suma 1 a z y se llama a la misma función en (0,1) y se marca que se pasó por esa dirección para no volver a pasar por ahí, si se logra llegar a las coordenadas de la salida la función termina y retorna un 1 indicando que es factible, en caso contrario, si se recorrieron todos los caminos posibles desde (0,0) y no se llegó a la salida, se retorna un 0, indicando que no es factible la solución.

3.8.2. Calidad

La función de evaluación trata de medir la *adecuación* del mapa creado a las solicitudes del usuario y su *accesibilidad*.

Score: Para medir la adecuación de la solución a los requerimientos del usuario se mide la diferencia de lo pedido respecto a lo construido. Así, si se pide un cierto porcentaje de celdas de cada tipo la función de evaluación considera la diferencia de la cantidad de celdas pedidas versus las reales. Para esto se calcula la diferencia en valor absoluto entre la cantidad de celdas permitidas de cada tipo y la cantidad actual en el mapa. Este valor se minimiza con la esperanza de que sea lo más cercano a 0. La ecuación 2 muestra cómo se realiza este cálculo. Este indicador de calidad se denomina *Score* en lo que sigue del escrito de memoria.

$$Score = |C_0 - C0| + |C_1 - C1| + |C_2 - C2| + |C_3 - C3| + |C_4 - C4| \quad (2)$$

Siendo $C0...C4$ la cantidad celdas de cada tipo actualmente en el mapa y $C_0...C_4$ la cantidad de celdas permitidas de cada tipo.

Accesibilidad: Además, se ha considerado medir la calidad de las soluciones respecto a la *accesibilidad* que tiene el mapa, es decir, del total de coordenadas del mapa, a cuantas se pueden acceder desde la coordenada inicial (0,0). La ecuación 3 muestra cómo se mide este indicador en la propuesta.

$$Accesibilidad = 1 - (Total - alcanzables)/Total \quad (3)$$

Cabe notar que si una solución es infactible, se le asigna una aptitud de 999999, de manera de no tener posibilidad de ser elegida en la selección y no se evalúa en el elitismo, de manera que el algoritmo solo considere soluciones factibles para el elitismo y selección.

3.9. Elitismo

El elitismo considera la identificación y almacenamiento de la mejor solución obtenida a través de todas las generaciones. El elitismo obliga incluir esta solución en cada población de las nuevas generaciones para poder aprovechar sus características.

3.10. Ejemplo final

En la figura 16 se muestra un mapa final construido luego de haber pasado por todo el proceso de resolución aquí propuesto:

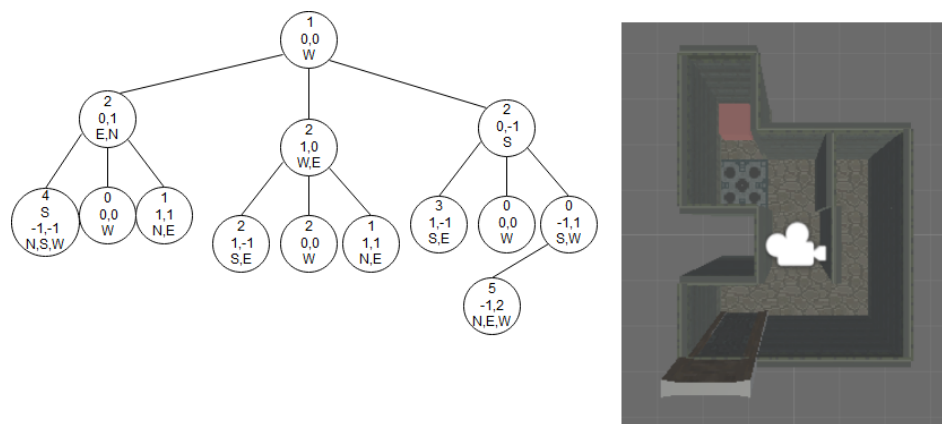


Figura 16: Ejemplo de traducción de árbol a mapa luego de haber pasado por el proceso completo.

Fuente: Elaboración propia.

En este caso los valores dentro de los nodos en la figura representan desde arriba a abajo: Tipo de celda, dirección puerta (Solo tipo 4), coordenadas y dirección de paredes. La coordenada (0,0) muestra la celda en donde se ubica la cámara y el eje de referencias es positivo hacia la derecha y arriba. A partir del árbol generado en el algoritmo, se decodifica el mapa como fue explicado en la sección 3.3. El mapa resultante es el que se puede observar a la derecha en la figura 16.

3.11. Parámetros de entrada

En esta sección se presentan los parámetros que utiliza la propuesta. Los parámetros corresponden a los que utiliza un GA junto a otros definidos para esta propuesta y son los siguientes:

- **Tamaño Solución:** Tamaño máximo del árbol que representa la solución.

- **Generaciones:** Número de generaciones que ejecutará el algoritmo.
- **Población inicial:** Número de soluciones con las que se ejecutará el algoritmo.
- **Pc:** Probabilidad de que se realice el paso de cruzamiento.
- **Pm:** Probabilidad de que se realice el paso de mutación.
- **Número Crossover:** Profundidad en donde se utilizara Cruzamiento.
- **Número Nodos Mutación:** Cantidad máxima de nodos que se mutarán.

Cabe mencionar que las probabilidades Pc y Pm se usan como cotas, es decir si por ejemplo Pc es 70 % y el número aleatorio que se genera entre 0 y 100 para verificar si se ejecuta el cruzamiento se encuentra entre 0 y 70, entonces se ejecuta la operación de cruzamiento, lo mismo para la mutación con Pm.

3.12. Variantes

De manera de evaluar la importancia de los indicadores de calidad propuestos: Score y accesibilidad, se han diseñado tres versiones del algoritmo genético que se diferencian esencialmente en la función de calidad que utilizan. Este análisis se considera interesante pues proviene de la naturaleza multi-objetivo del problema donde se busca una solución que cumpla de la mejor manera posible con los requerimientos del usuario y la accesibilidad del mapa que asegura un cierto nivel de dificultad de éste.

3.13. Fase 1

La primera versión del algoritmo considera la evaluación de la calidad de los mapas considerando únicamente el indicador *Score*. En esta versión el elitismo revisa únicamente este valor para comparar las soluciones.

3.14. Fase 2

Para la fase 2 se modificó la función de evaluación para calcular la calidad, en esta fase se agregó el indicador de alcanzabilidad. Esta solo se calcula para las soluciones que tienen un mejor puntaje, ya que su cómputo es un proceso costoso.

Cabe notar que si una solución es infactible, se le asigna puntaje muy pequeño, de manera de tener muy poca probabilidad de ser elegible en la selección y no se evalúa en el elitismo.

Así, el algoritmo solo considera soluciones factibles para el elitismo, pero para la selección considera soluciones factibles e infactibles, de manera de beneficiar la exploración de soluciones.

En este caso, en el proceso de elitismo se consideran las dos funciones de calidad. Para verificar que se encuentra una mejor solución elitista, la nueva solución debe dominar a la antigua solución elitista, es decir, debe tener al menos la misma alcanzabilidad, pero ser mejor en el indicador de score.

Por otro lado, en el proceso de selección se considera el indicador de score como un porcentaje de score respecto al mejor posible. Así, es posible incluir soluciones infactibles en el proceso de selección. Por lo tanto, las probabilidades de selección, cuando hubo una mejora se calculan utilizando la ecuación 4.

$$P_i = \frac{F_i}{\sum_{i=1}^N F_i} \quad (4)$$

En caso contrario, si se considera la mejor solución, llamada BestSolution, hasta ahora, ya que esta sería distinta a los N padres actuales. En este caso la probabilidad de selección se calcula mediante la ecuación 5.

$$P_i = \frac{F_i}{\sum_{i=1}^{N+1} F_i} \quad (5)$$

En donde F_i es el score de la solución i , considerando valores en el rango $[0;1]$.

3.15. Fase 3

En la fase 3 se modifica la función de evaluación de todas las soluciones considerando los valores de ambos indicadores. Así, el cálculo de la calidad de cada una de las soluciones se realiza de acuerdo a la ecuación 6. Como es posible observar se agregaron dos parámetros, α y β , los cuales van de 0 a 1 y se usan para ponderar el score final, el cual consiste del score antiguo y la alcanzabilidad respectivamente.

$$Score_{Final} = Score * \alpha + alcanzabilidad * \beta \quad (6)$$

Cabe notar que el score y la alcanzabilidad están en porcentaje, para que se puedan sumar normalmente.

Además, ahora la alcanzabilidad se calcula para todas las soluciones, por lo que las soluciones infactibles también tienen consideradas la alcanzabilidad.

Resumen

En este capítulo se vio todo lo que se implementó para crear la propuesta de solución al problema de generación automática de mapas. Como solución se propuso utilizar un GA (Genetic Algorithm - Algoritmo Genético) por el tipo de control que se tiene sobre los parámetros y por el tipo de mapas que se pueden construir con esta técnica. Los algoritmos genéticos son algoritmos iterativos en donde en cada iteración se realizan los procesos de evaluación, elitismo, selección, cruzamiento y mutación. El cromosoma que se utilizó para representar las soluciones es de un árbol triario, es decir con 3 hijos en cada nodo y cada nodo contiene la información de su celda correspondiente. Para decodificar el mapa, se recorre el árbol en sentido pre-orden. Para la inicialización del algoritmo, se crea una cantidad de soluciones igual al parámetro **Población inicial**. El proceso de selección utiliza el operador de selección basado en ruleta. El operador de cruzamiento es el de one point crossover, y trata sobre intercambiar 1 rama en un punto entre 2 soluciones. El operador de mutación modifica una cantidad de nodos entre 1 y **Numero Nodos Mutación**, en donde el nodo puede cambiar a cualquier tipo excepto el nodo salida. La función de evaluación consiste de la factibilidad y la calidad de las soluciones, para que una solución sea factible, se tiene que poder llegar a la salida desde las coordenadas (0,0) que es desde donde parte el jugador. La calidad consta de 2 funciones, el score y la accesibilidad, el score mide que también se cumple el requerimiento de cantidad de celdas de cada tipo pedidas mientras que la accesibilidad mide a cuantas celdas se pueden llegar desde las coordenadas (0,0). El elitismo consiste en la identificación y almacenamiento de la mejor solución a lo largo de todas las iteraciones. Se crearon 3 variaciones del algoritmo para evaluar la importancia de los indicadores de calidad, en la primera fase solo se considera el score. En la fase 2 se agregó la accesibilidad, la cual solo se calcula para las soluciones que tienen un mejor puntaje que el actual, por lo que se calculan por separado. En la fase 3 se agregaron los parámetros α y β , para ponderar el score y la accesibilidad juntos, calculando ambos indicadores para todas las soluciones, incluso las infactibles.

CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

En el presente capítulo se presentan los experimentos realizados para la verificación y validación de la solución propuesta. Los experimentos se concentran en evaluar las capacidades de la propuesta de generar mapas factibles, difíciles y de diferentes estructuras y tamaños.

Experimentos

Los experimentos consideran la evaluación de las tres versiones del algoritmo genético implementado: Fase 1, Fase 2 y Fase 3.

La principal diferencia entre la fase 1 y 2 es que la segunda considera la alcanzabilidad para encontrar mejores resultados, solo cuando se obtiene un mejor score. El cálculo de alcanzabilidad es costoso en términos de tiempo pues requiere la evaluación de existencia de caminos desde la celda de partida a todas las demás del mapa. Además, en esta versión del algoritmo se permite que soluciones infactibles sean elegidas en el proceso de selección. Por otro lado, la diferencia con la fase 3, es que esta agregó los parámetros α y β , que ponderan el score y la alcanzabilidad juntos, para obtener un nuevo tipo de score, todo en porcentaje. Por lo mismo en esta versión se calcula la alcanzabilidad para todas las soluciones, no solo las con mejor score por lo que es posible un importante incremento en términos de tiempo de ejecución.

El algoritmo fue implementado en C#, usando Unity 2018.2.6f1 y las pruebas se realizaron en un computador con procesador Intel(R) Core(TM) i5-4210M con un CPU de 2.60GHz con 8 GB de memoria RAM sobre sistema operativo Microsoft Windows 7 Ultimate SP1.

En estos experimentos los parámetros nodos mutación y número cruzamiento fueron usados como la mitad del tamaño población+1, aproximando hacia arriba. La probabilidad de mutación (P_m) siempre será menor o igual a la probabilidad de cruzamiento (P_c) ya que si se usaran los dos operadores todo el tiempo, podrían cambiar mucho los mapas, pudiendo llegar a ser destructivo. Así, se utilizaron los valores de probabilidades que mostraron los mejores resultados en experimentos previos. Se usó un tamaño de población de cuatro soluciones para todos los experimentos debido a que utilizar 3 o menos daba muy poca opciones a la hora de utilizar los cruzamientos o simplemente no tenía sentido intercambiar todo el tiempo entre los mismos padres. Por otro lado, no se usaron más de 4 ya que con cada aumento en una unidad de población, el tiempo de ejecución aumentaba de manera considerable.

Para la versión Fase 3 del algoritmo se estudian tres combinaciones de valores de α y β = (0.5, 0.5), (1, 0) y (0, 1), para así analizar los casos extremos que se podían obtener con el

algoritmo, utilizando exactamente la mitad de cada uno, utilizando solo el score y utilizando solo la alcanzabilidad como medidas de calidad.

Cada uno de los experimentos fue ejecutado 10 veces de manera de obtener indicadores adecuados para algoritmos de naturaleza estocástica.

4.1. Indicadores

Dentro de los indicadores que se midieron en cada uno de los experimentos se encuentran el tiempo total que demora el algoritmo, el tiempo de convergencia, la alcanzabilidad y el score obtenido. Cada uno de los cuales se explica a continuación:

- El tiempo total se refiere al tiempo que se demoró el algoritmo en realizar una ejecución completa, es decir desde que inicia hasta que se entrega el mapa final generado.
- La convergencia es el tiempo que se demoró el algoritmo desde su inicio hasta que obtiene la mejor solución del proceso.
- La alcanzabilidad es el porcentaje de mapa alcanzable desde donde parte el jugador, es decir a cuantas celdas se puede acceder desde la posición (0,0). Se mide porcentualmente respecto al total de celdas del mapa correspondiente.
- El score obtenido es simplemente el score final que se obtuvo al evaluar la mejor solución. Se mide porcentualmente respecto al total de celdas del mapa correspondiente.

4.2. Instancias/Casos de pruebas

En los experimentos se han considerado tres tamaños de mapa a estudiar: Tamaño 4, Tamaño 7 y Tamaño 9. Los mapas de tamaño 4 generan mapas de alrededor de 20 celdas y demoran aproximadamente entre 0.2 y 9 segundos en entregar el resultado final. Los mapas de tamaño 7 generan mapas de alrededor de 71 celdas y demoran aproximadamente 10 segundos en la fase 1 y 2 y aproximadamente 100 segundos en la fase 3. Estos tiempos son distintos entre las fases ya que se evalúan distintas cosas en cada uno, como se explicó en el capítulo 3.12 y estos tiempos consideran la evaluación de varios mapas del tamaño correspondiente. Los mapas de tamaño 9 generan mapas de alrededor de 125 celdas y demoran aproximadamente 90 segundos en la fase 1 y 2 y aproximadamente 1400 segundos en la fase 3. Como se puede observar, estos tamaños abarcan tanto tiempos casi instantáneos como 0.2 segundos hasta tiempos de espera mayores a 20 minutos y abarcan una gran cantidad de celdas. Desde mapas pequeños donde es posible encontrar la salida en un par de pasos, hasta mapas donde es necesario investigar para poder encontrar la salida.

A continuación se listan detalladamente los parámetros adicionales a definir en cada uno de los mapas:

- **Fitness Cell espinas:** Porcentaje de celdas tipo espinas permitidas.
- **Fitness Cell normal:** Porcentaje de celdas normales permitidas.
- **Fitness Cell normal con una pared:** Porcentaje de celdas normales que tienen una pared permitidas.
- **Fitness Cell normal con dos paredes:** Porcentaje de celdas normales que tienen dos paredes permitidas.
- **Fitness Cell Puerta:** Porcentaje de celdas con puerta permitidas.

En el proceso de generación de mapas se considera como estructura del mapa una cierta distribución de celdas de cada tipo. Así, se definieron porcentajes de cantidad de celda de cada tipo ya que los mapas están compuestos por varios tipos de estas. Claramente, el porcentaje de celdas de cada tipo también se relaciona con la factibilidad de las soluciones, pues no puede suceder que se obtenga un mapa en donde todo el piso está lleno de espinas o un mapa lleno de paredes.

Para las pruebas se consideraron dos tipos de experimentos por cada tamaño de mapa estudiado: configuración estándar y configuración laberinto. La configuración estándar tiene la misma ponderación para todos los tipos de celda, por lo que es más fácil que el algoritmo encuentre una solución que satisfaga cada una de la cantidad de celdas permitidas. Estos mapas poseen una estructura más balanceada en el sentido de poseer una cantidad baja de cada tipo de celda. Por otro lado, la configuración laberinto está enfocada en que los mapas tengan una gran cantidad de celdas con 2 paredes, generando así mapas con pocas áreas con espacio abierto, logrando así una estructura laberíntica dentro del mapa.

Las tablas 1 y 2 muestran los valores de los parámetros de las instancias usadas en los experimentos descritos.

Parámetro	Valor
Espinas	20
Normal	20
1 pared	20
2 paredes	20
Puerta	20

Tabla 1: Parámetros usados en la configuración estándar.
Fuente: Elaboración Propia.

Parámetro	Valor
Espinas	10
Normal	15
1 pared	10
2 paredes	55
Puerta	10

Tabla 2: Parámetros usados en la configuración laberinto.
Fuente: Elaboración Propia.

La tabla 3 muestra los valores de parámetros del algoritmo genético usados en los experimentos.

Parámetro	Valor
Pc	70
Pm	50
Generaciones	100
Población inicial	4
Número Crossover	3/5/6
Número Nodos Mutación	3/5/6

Tabla 3: Parámetros del algoritmo genético.
Los números 3/5/6 son los números utilizado para los tamaños de mapa 4, 7 y 9 respectivamente. Fuente: Elaboración Propia.

Resultados

En esta sección se presentan los resultados obtenidos en los experimentos realizados. Los resultados se presentan organizados por tamaño de instancia. Para cada caso se estudian las versiones de mapas estándar y laberinto. Se usan boxplots para mostrar de mejor manera los resultados obtenidos. Los boxplot o diagramas de cajas son un método estandarizado para representar gráficamente una serie de datos numéricos a través de sus cuartiles. Por lo tanto, un boxplot muestra a simple vista la mediana y los cuartiles de los datos al mismo tiempo de mostrar datos atípicos (outliers), representados por círculos blancos y los "bigotes" (whiskers) representados por líneas punteadas que se extienden verticalmente de los boxplot que representan data que esta fuera del cuartil superior e inferior.

En cada uno de los gráficos que se presentan a continuación la caja naranja representa los resultados obtenidos usando el algoritmo genético con la fase 1, la caja roja representa los resultados del algoritmo genético con la fase 2 y las cajas azul, verde y amarilla representan el algoritmo genético de la fase 3 con sus respectivos α y β , en donde el azul es con (.5 .5), el verde es con (1. 0.) y el amarillo es con (0. 1.). El experimento de la fase 1 solo considera la función de calidad del score y no considera soluciones infactibles para la evaluación y selección, el experimento de la fase 2 considera la alcanzabilidad solo para las soluciones que tienen un score mejor que la mejor solución actual y considera las soluciones infactibles en la selección y por último, la fase 3 agrego el α y β para ponderar el score y la alcanzabilidad juntas y se calcula para todas las soluciones, por lo que ahora se consideran las soluciones infactibles para la evaluación y selección.

4.3. Mapas de tamaño pequeño

En esta sección se presentan los resultados obtenidos para los mapas de tamaño pequeño, los cuales corresponden a los mapas de tamaño 4 y utilizan alrededor de 20 celdas.

La figura 17 muestra los resultados obtenidos al ejecutar las tres versiones del algoritmo en el mapa estándar y la figura 18 muestra un ejemplo de un resultado de un mapa de tamaño 4 estándar.

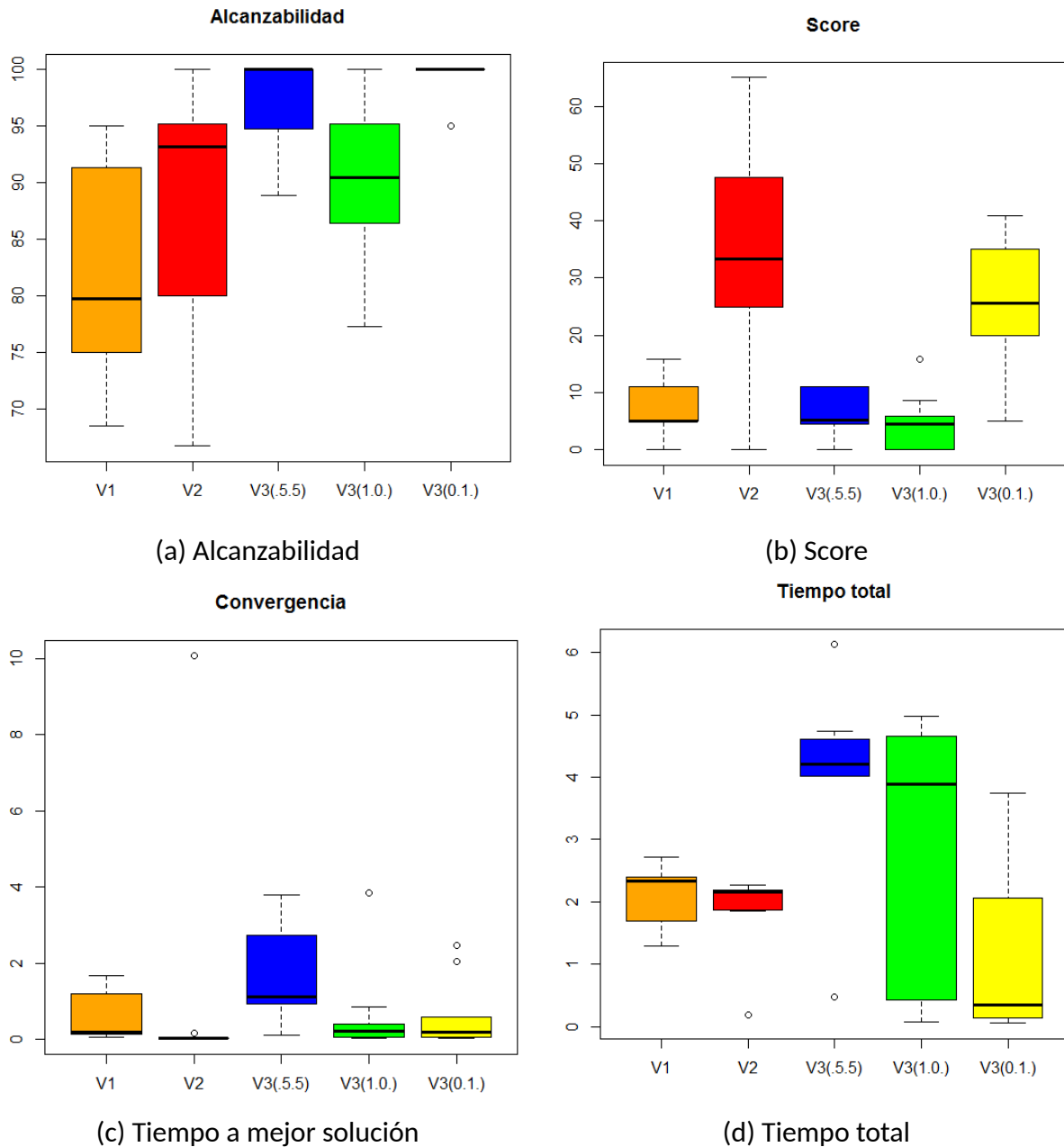


Figura 17: Comparación entre las 3 fases con configuración estándar con mapas de tamaño 4.

Fuente: Elaboración propia.

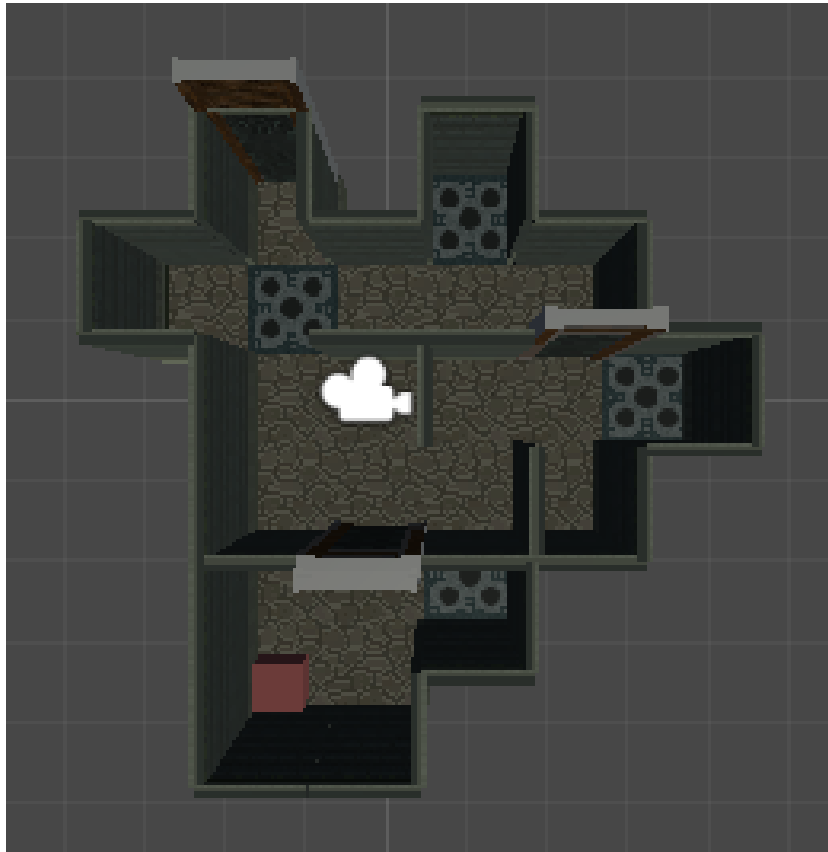


Figura 18: Ejemplo del resultado de un mapa con configuración estándar con mapas de tamaño 4.

Fuente: Elaboración propia.

Como es posible observar, la alcanzabilidad crece a medida que se agregan mejoras al algoritmo y en la fase 3, considerando las configuraciones de α y β que consideran la alcanzabilidad y el score se logran muy buenos resultados a este respecto, notar que el con α y β (0. 1.) tiene una alcanzabilidad casi perfecta, lo cual era esperado y esto se debe a que al solo considerar este aspecto y ser pocas celdas, el algoritmo llega fácilmente a un 100 % de alcanzabilidad.

Por otro lado, al poner atención al score es posible notar que los mejores score fueron obtenidos por la versión del algoritmo que incorpora la fase 3 que no considera la alcanzabilidad, es decir la V3(1. 0.) ya que como es una función de minimización, mientras más cerca del 0, mejor es la solución. Esto indica que al ser un mapa pequeño, la alcanzabilidad no aporta mucho a los resultados finales, ya que en estos mapas, al tener poco espacio, tener o no una pared en cierta celda puede cambiar drásticamente la alcanzabilidad del mapa. Esto conlleva a que fácilmente se obtengan peores scores al no cumplir la cantidad de paredes pedidas por la configuración. Esto se sustenta al ver el boxplot de la fase 3 (0. 1.) que tiene mejor alcanzabilidad y tienen por tanto los peores scores junto con la fase 2 la cual se explica que al considerar el 100 % de este factor, pero solo en las mejores soluciones, obtiene peores

score que por ejemplo el experimento de la fase 3 (.5 .5) que solo considera la mitad de la alcanzabilidad.

Al observar los tiempos de ejecución es posible notar que la versión del algoritmo que incorpora la fase 2 es la más eficiente pues posee tiempos bajos tanto de convergencia como totales. Si bien los cálculos de la fase 2 son más costosos, al ser un mapa pequeño, esto impacta más en los tiempos totales, en donde se puede ver que la fase 1 es superior en algunos casos, pero en la convergencia la fase 2 es superior en todo sentido, al llegar a un mapa con bajo score y alta alcanzabilidad rápidamente. Por otro lado, la versión que incorpora la fase 3 con parámetros (0.5, 0.5) resulta ser la versión más consumidora en términos de tiempos de convergencia y totales al solo utilizar la mitad de ambos indicadores de calidad y debido a que en la fase 3 la alcanzabilidad se calcula para todas las soluciones, lo cual conlleva un gran consumo de tiempo.

Si solo se calculara la alcanzabilidad, se llegaría rápido a una solución óptima y terminaría antes, como se puede ver en el tiempo total y de convergencia del V3(0. 1.).

Por otro lado, al tratar de hacer las 2 cosas al mismo tiempo en el (0.5 0.5) el algoritmo no logra terminar tan rápido ya que si el score es bueno, probablemente tiene poca alcanzabilidad y el puntaje empeora y vice versa, logrando así no llegar nunca al óptimo, lo que conlleva a que se demore al tratar de equilibrar ambos mundos.

Además, la fase 2 es la más rápida por que hace justamente lo contrario, solo utiliza la alcanzabilidad cuando hay un puntaje mejor que el actual, por lo que no hace cálculos extras y solo apunta a mejorar lo mejor de lo mejor, convergiendo muy rápido.

Observando los tiempos totales de ejecución, se puede observar que todos son muy bajos, menos de 10 segundos, por lo que para la generación de mapas pequeños, los tres algoritmos son candidatos para ser usados en un videojuego por su rápido y buen desempeño.

Como se puede observar en el mapa de la figura 18, se obtuvo un mapa bien balanceado, un par de puertas bien colocadas (la superior es una puerta trampa ya que no se puede abrir), las paredes y espinas están bien colocadas, dando una sensación de un calabozo y la salida no está inmediatamente a la vista, por lo que en resumen, es un buen mapa generado que sirve para el primer nivel de un juego.

La figura 19 muestra los resultados obtenidos al ejecutar las tres versiones del algoritmo en el mapa laberinto y la figura 20 muestra un ejemplo de un resultado de un mapa de tamaño 4 con configuración laberinto.

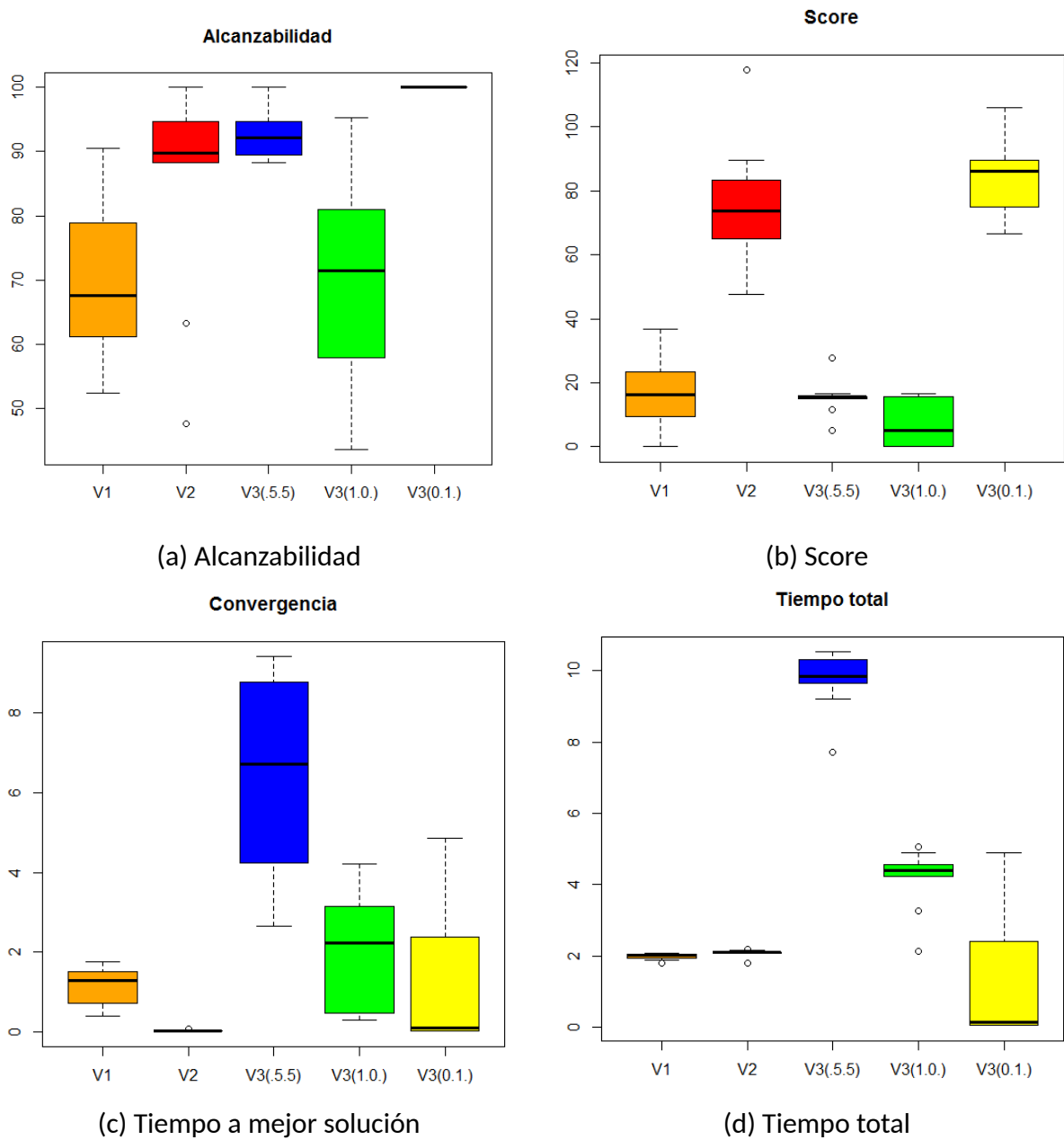


Figura 19: Comparación entre las 3 fases con configuración laberinto con mapas de tamaño 4.

Fuente: Elaboración propia.

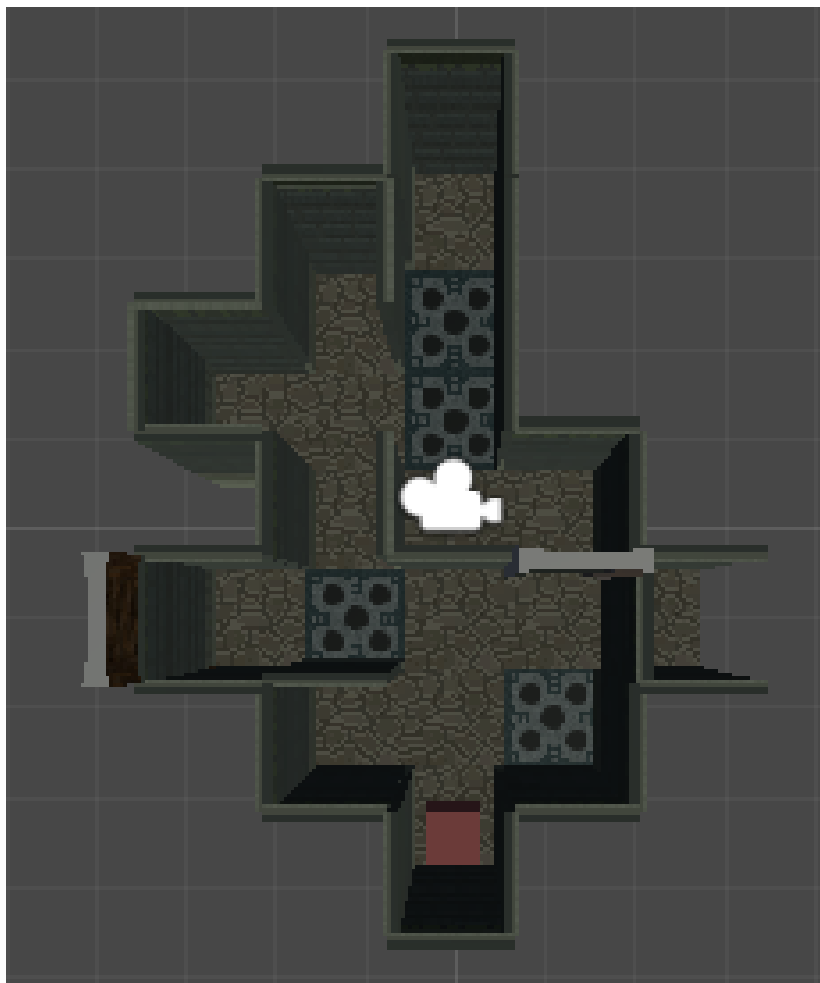


Figura 20: Ejemplo del resultado de un mapa con configuración laberinto con mapas de tamaño 4.

Fuente: Elaboración propia.

Al igual que en la versión estándar, la alcanzabilidad mejora a medida que se agregan mejoras al algoritmo que consideran este parámetro de calidad, en donde se puede observar que al solo considerar esta función en la fase 3 (0. 1.) se obtiene siempre una alcanzabilidad perfecta. Por otro lado con esta configuración, el experimento v3 (1. 0.) obtuvo la peor alcanzabilidad, mientras que en la versión estándar, la v1 era la peor, esto se debe a que al ser un experimento enfocado en laberintos, es más probable que al no considerar la alcanzabilidad en mapas pequeños, estos cierren gran parte del mapa al agregar una pared en cierta zona, lo que conlleva a que el v1 y v3 (1. 0.) al no utilizar esta función de calidad, los resultados obtenidos sean peores que los anteriores y como el v3 (1. 0.) utiliza funciones infactibles, se vuelve más complicado para el algoritmo obtener una solución factible que tenga alta alcanzabilidad.

Por otro lado, en el score se puede observar que a pesar de que se repite la situación de la configuración estándar, ahora todos los puntajes son al menos 10 puntos peores. Esto, debido a que al ser un mapa enfocado en tener varias paredes, deja que se puedan colocar pocas celdas de otro tipo. Así, resulta muy fácil que el algoritmo exceda la cantidad ideal de celdas normales, con espinas o puertas por ejemplo, lo cual es aún más cierto cuando se considera la alcanzabilidad. Esto debido a que esta función forzaría al algoritmo a sacar paredes para poder llegar a todas las celdas, empeorando aún más el score, como es el caso de la fase 2 y la fase 3 (0. 1.).

Con respecto a los tiempos de ejecución, se puede observar que ocurre algo similar a los tiempos de la configuración estándar, con la diferencia que aquí los tiempos son casi el doble para la fase 3 (0.5 0.5). Esto se debe a que en este experimento, al utilizar la mitad de ambos factores de calidad, estos llegan más lento a los valores deseados, ya que en la versión laberinto, si se mejora el score, agregando más paredes, probablemente empeore la alcanzabilidad y viceversa, lo que conlleva a que se realicen más generaciones antes de llegar a los valores deseados. Por otro lado, algunos tiempos se 'equilibraron' como en la fase 1 y 2. Esto debido a que al tener muchas paredes y pocas celdas, los mapas siempre se terminan pareciendo entre ellos, ocasionando que el algoritmo intente mejorarlo de la misma manera en todos los casos.

Los tiempos, al igual que la versión estándar, siguen siendo muy bajos, por lo que para la generación de mapas pequeños de tipo laberinto, cualquiera de las tres versiones del algoritmo sirve por su buen y rápido desempeño.

Como se puede observar en el mapa de la figura 20, se obtuvo un mapa que al ser pequeño, no se distingue que sea con configuración de laberinto, pero se puede apreciar un cierto nivel de dificultad, al tener 2 celdas de espinas juntas al lado de la celda de inicio, nuevamente la salida no está inmediatamente a la vista, otra vez existe una puerta trampa y una puerta bien colocada, por lo que en resumen, es un buen mapa generado que sirve para el primer nivel de un juego a pesar de que por el tamaño no se aprecie que es un laberinto.

4.4. Mapas de tamaño mediano

En esta sección se presentan los resultados obtenidos para los mapas de tamaño mediano. La figura 21 muestra los resultados obtenidos al ejecutar las tres versiones del algoritmo en el mapa estándar y la figura 22 muestra un ejemplo de un resultado de un mapa de tamaño 7 estándar.

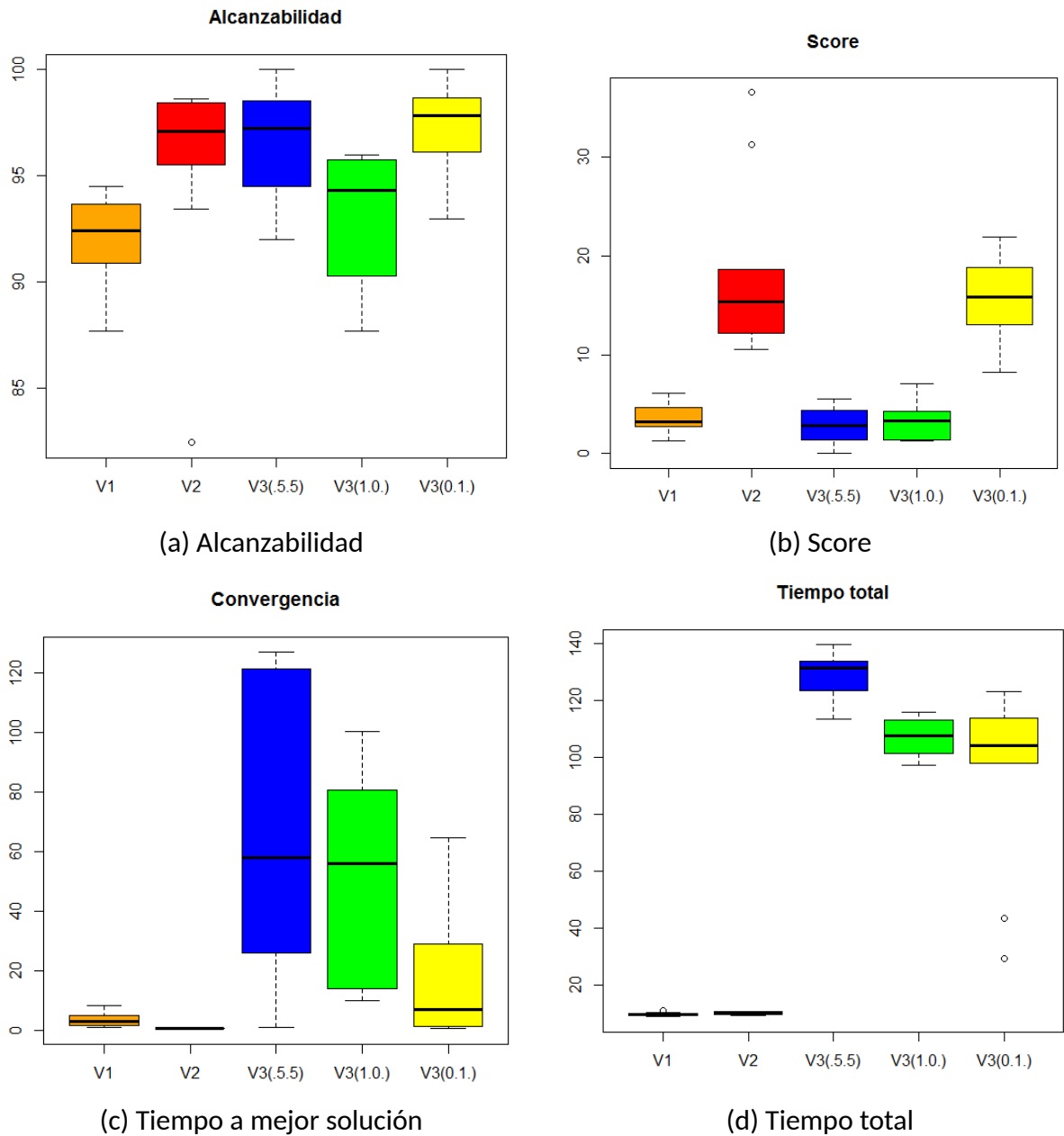


Figura 21: Comparación entre las 3 fases con configuración estándar con mapas de tamaño 7.

Fuente: Elaboración propia.

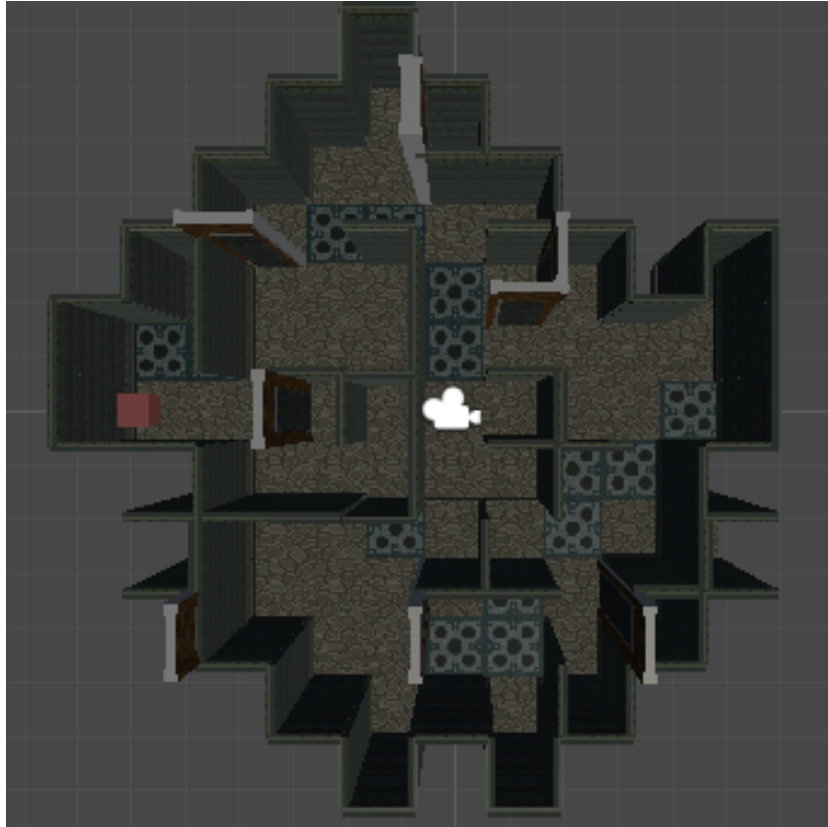


Figura 22: Ejemplo del resultado de un mapa con configuración estándar con mapas de tamaño 7.

Fuente: Elaboración propia.

Se puede observar en la alcanzabilidad que esta no aumenta mucho después de la fase 2, lo cual se debe a que al ser mapas más grandes y homogéneos en su cantidad de celdas de cada tipo, aumentar la alcanzabilidad se vuelve más difícil ya que quitar o agregar una pared en una celda puede que ayude o puede que quede igual.

Por otro lado, los score por lo general mejoraron en todos los casos con respecto a los de tamaño 4, esto es debido a que al utilizar más celdas, el porcentaje de celdas permitidas de cada tipo aumenta, relajando así las restricciones de obtener el número exacto de cada celda. Se puede observar que ahora los score de V3(.5 .5) y V3(1. 0.) son casi los mismos, lo cual se debe a que al ser mapas más grandes y la alcanzabilidad no es tan volátil como en los de tamaño 4, esta logra ser un real aporte al score final.

En el tema de los tiempos, se puede observar que estos aumentaron drásticamente en la fase 3, haciendo parecer que los tiempos de la fase 1 y 2 fuesen casi 0. Esto es debido a lo que se había mencionado anteriormente referido al cálculo de alcanzabilidad. Este costo se vuelve más importante a medida que crece el tamaño del mapa.

Se puede observar en la figura 22 que ahora los mapas se parecen más a los de un juego de esta temática, habiendo más espacios para explorar y perderse en sus pasillos antes de encontrar la salida, la cual nuevamente está escondida en el borde del mapa, existe una buena cantidad de espinas que están en el camino para llegar a la salida, por lo que tiene cierto grado de dificultad.

La figura 23 muestra los resultados obtenidos al ejecutar las tres versiones del algoritmo en el mapa laberinto y la figura 24 muestra un ejemplo de un resultado de un mapa de tamaño 7 laberinto.

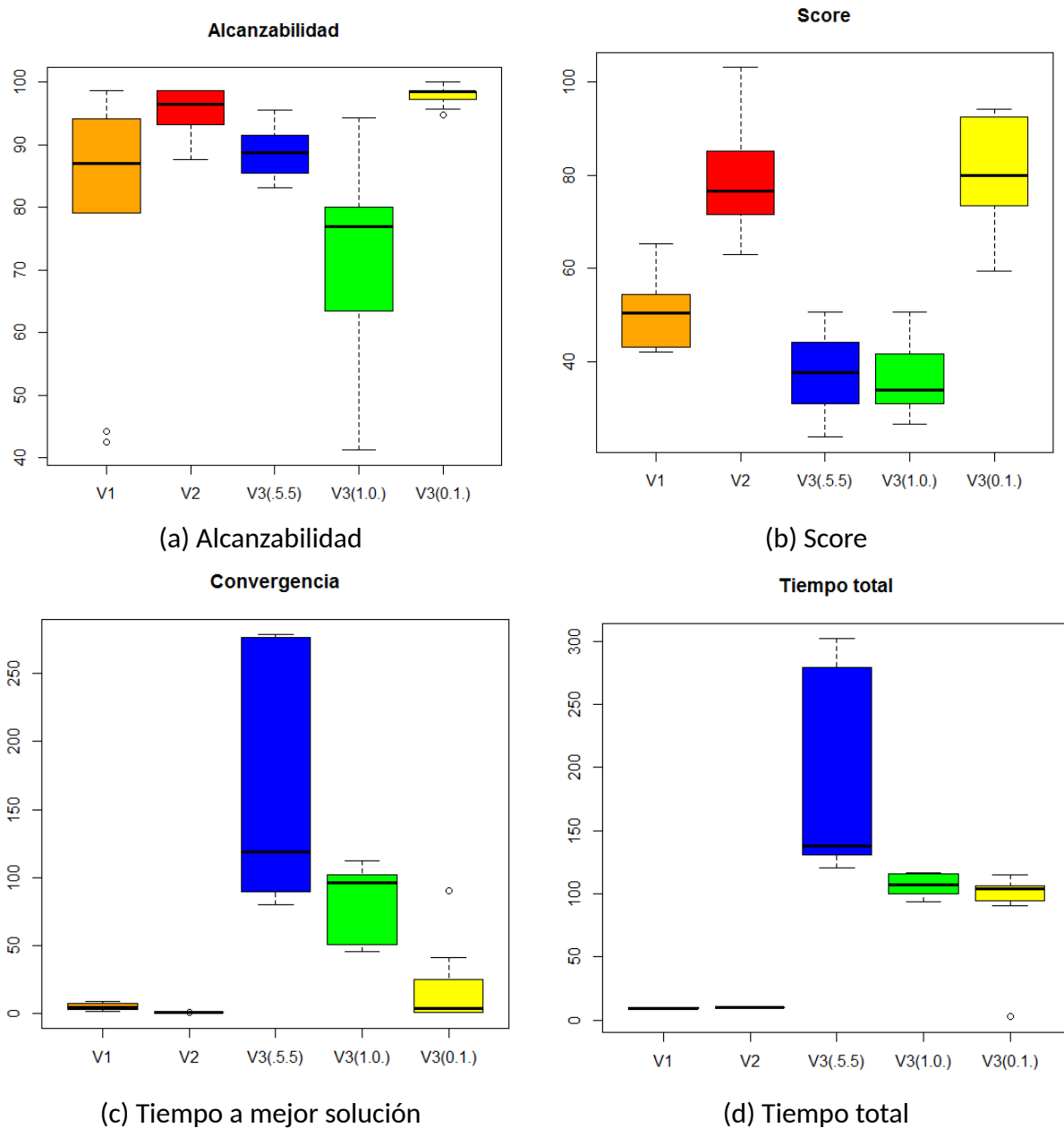


Figura 23: Comparación entre las 3 fases con configuración laberinto con mapas de tamaño 7.

Fuente: Elaboración propia.



Figura 24: Ejemplo del resultado de un mapa con configuración laberinto con mapas de tamaño 7.

Fuente: Elaboración propia.

La alcanzabilidad en este experimento, a diferencia del estándar, si mejora, lo cual se debe a que al generarse un laberinto, el agregar o quitar una pared puede lograr cerrar o abrir grandes áreas, lo que conlleva a que la alcanzabilidad aumente, sobre todo si solo se considera la alcanzabilidad como es el caso de la V3(0. 1.)

Por su parte, el score, en todas sus versiones, aumentó significativamente versus la configuración estándar, esto es debido a que la configuración laberinto es más restrictiva en cuanto a la cantidad de celdas de cada tipo que se pueden agregar, el pedir muchas con dos paredes y pocas del resto. Así, cuando es necesario agregar más celdas, es más fácil que el algoritmo exceda o le falten celdas para la cantidad óptima deseada, lo que conlleva a peores resultados.

Por otro lado, se puede observar en los tiempos tanto de ejecución como de convergencia que el usar tanto la alcanzabilidad como el score al mismo tiempo entrega tiempos bastante elevados. Mientras que para las versiones V3(1. 0.) y V3(0. 1.) los tiempos son

casi los mismos que a la versión estándar, el problema escala en la V3(.5 .5). Esto debido nuevamente a que el algoritmo tiene que calcular varias veces la alcanzabilidad y el score, las cuales combinadas, son costosas en todas las soluciones, produciendo esperas mayores a 4 minutos y al ser costosas y no llegar rápidamente a una solución óptima, el algoritmo esta más tiempo procesando que en las otras versiones.

Se puede ver en la figura 24 que ahora el mapa se parece más a un laberinto, produciendo caminos sin salidas o puertas que llevan a callejones sin salida, incluso la salida ahora se encuentra encerrada en una habitación que está a la derecha del final de un pasillo largo.

Al comparar este mapa con el de su versión estándar, se puede observar que la de laberinto se crean varias habitaciones en el mapa, generando más instancias para colocar objetos o enemigos en ellas, creando así un factor de sorpresa mayor para el jugador, el cual si bien se puede crear en la estándar, no es tan fácil al no tener tantas zonas cerradas. Por otro lado, al comparar estos mapas con los de tamaño 4, estos al tener una pequeña cantidad de celdas no logran obtener mucha forma, por lo que si bien sirven para un primer nivel del juego, quedan cortos comparados con todo lo que se puede lograr hacer en los dos tipos de mapas de tamaño 7.

4.5. Mapas de tamaño grande

En esta sección se presentan los resultados obtenidos para los mapas de tamaño grande. La figura 25 muestra los resultados obtenidos al ejecutar las tres versiones del algoritmo en el mapa estándar y la figura 26 muestra un ejemplo de un resultado de un mapa de tamaño 9 estándar.

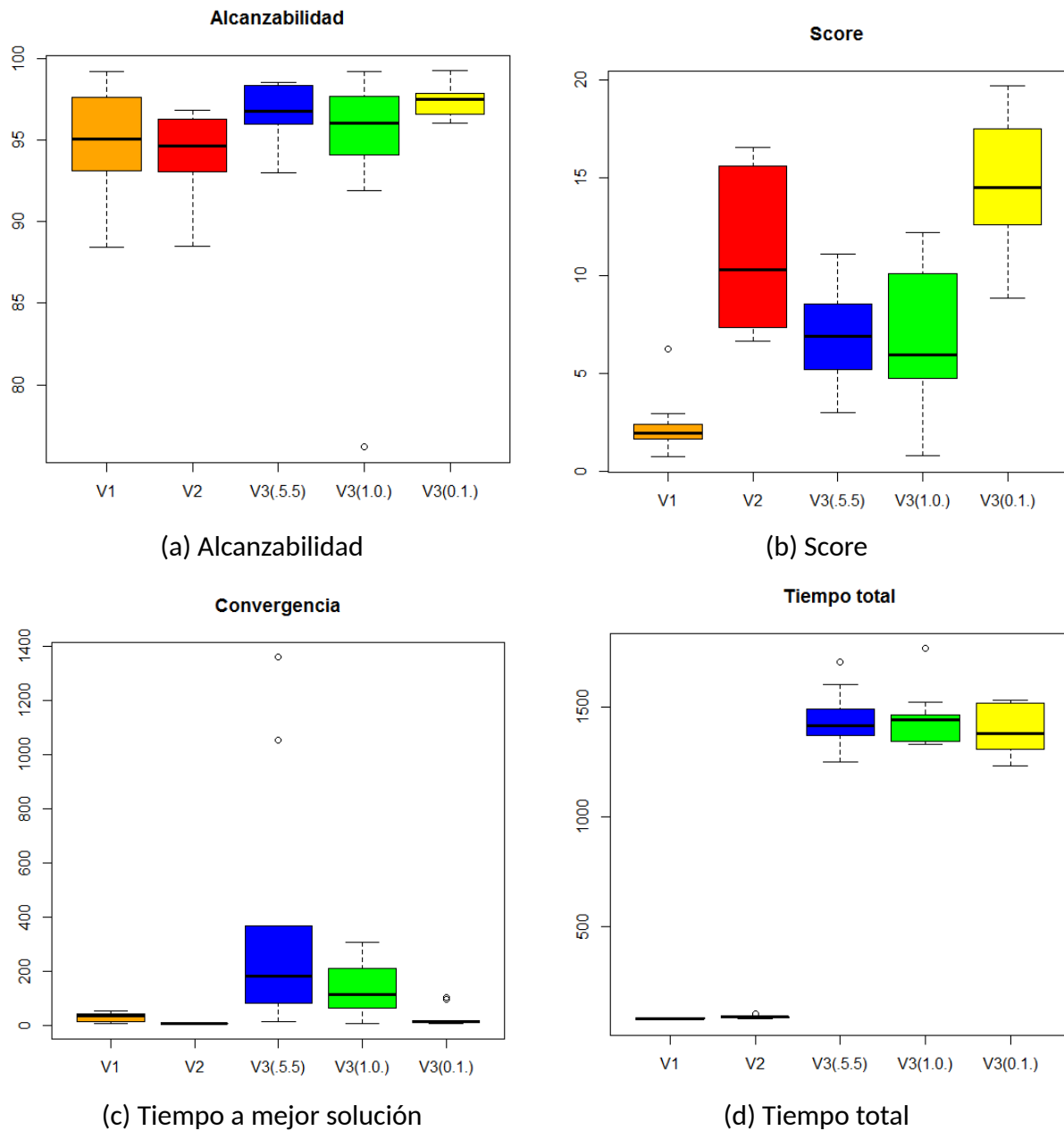


Figura 25: Comparación entre las 3 fases con configuración estándar con mapas de tamaño 9.

Fuente: Elaboración propia.

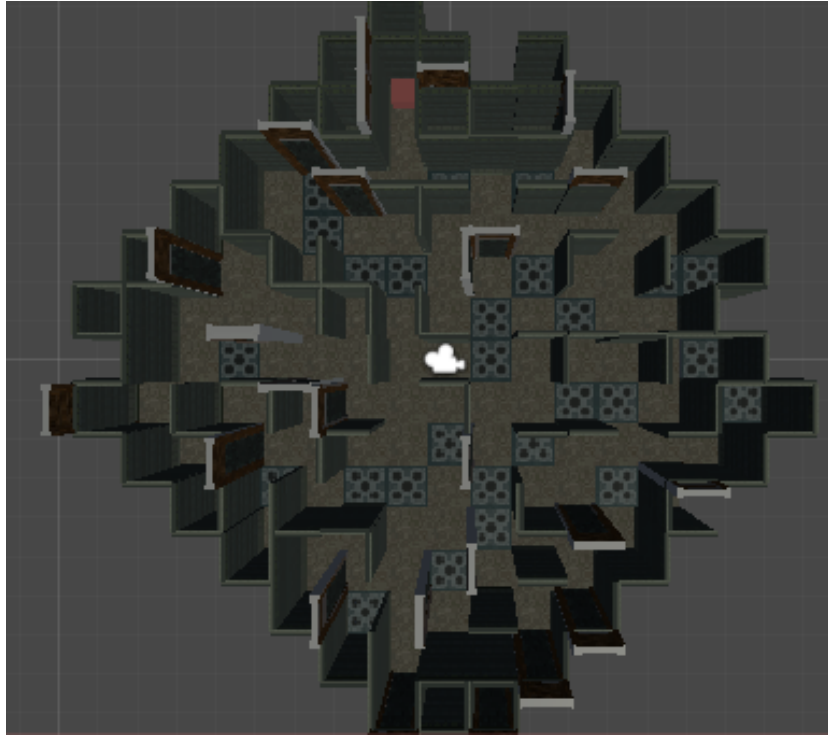


Figura 26: Ejemplo del resultado de un mapa con configuración estándar con mapas de tamaño 9.

Fuente: Elaboración propia.

La alcanzabilidad en los mapas de tamaño 9 es muy alta en todas las fases, mayor al 95 % en la mayoría de los casos. Esto es debido a que son tan grandes los mapas de tamaño 9, que es difícil bloquear completamente un conjunto de celdas, por lo que el agregar la alcanzabilidad en las siguientes fases no ayuda en gran medida como sí pasa en los mapas más pequeños. En donde se puede observar que la versión amarilla sigue siendo la mejor en este ámbito.

Por otro lado, el score mejoró, en general, en cuanto a los resultados de los mapas de tamaño 7, ya que al tener que usar la alcanzabilidad en mapas grandes, estos son más fáciles de acceder a una celda, lo que conlleva a que este criterio sea bien alto, lo que a su vez ayuda a mejorar el score. Las excepciones son las versiones V3 (0.5 0.5) y V3(1. 0.). Estos dos casos empeoraron debido a que utilizan la mitad de la accesibilidad o simplemente no la utilizan, logrando peores resultados, que los otros, incluso que la fase 1. Esto es debido a que la forma de calcular de la fase 1 y de la fase 2 y 3 es ligeramente distinta, ya que la primera no consideraba soluciones infactibles, mientras que las demás sí, lo que significa que estas consideran soluciones con peor alcanzabilidad, lo que explica porque la fase 1 obtiene mejores scores.

Respecto de los tiempos, hay que notar que si bien a simple vista parecen todos muy cercanos a 0, en realidad, en la escala se ve que los tiempos son muy superiores a los que

se habían visto antes, por lo que no se logran apreciar los tiempos de la fase 1 y 2, que si bien son superiores a los de tamaño 4 y 7, es en la fase 3 donde realmente hay un alza muy significativa. Las versiones de la fase 3 llegan a tiempos de espera de casi 25 minutos para cualquiera de las tres versiones. Por otro lado, como era de esperarse, el experimento V3(0. 1.) tiene el mejor tiempo de convergencia de los experimentos de la fase 3 ya que la alcanzabilidad en los mapas grandes es mayor a medida que el mapa crece en tamaño. Por otro lado, la fase 2 tiene la convergencia más rápida al utilizar la alcanzabilidad y el score por separado.

En la figura 26 se puede observar que los mapas de este tamaño son enormes, generando paredes y espinas por todos lados, haciéndolos ideales para mapas difíciles o como último nivel de un juego. El único problema en este mapa es que al tener tantos espacios, las puertas pierden un poco el sentido a diferencia de los mapas más pequeños a este, en donde las puertas por lo general estaban bien colocadas y cumplían su función.

La figura 27 muestra los resultados obtenidos al ejecutar las tres versiones del algoritmo en el mapa laberinto y la figura 28 muestra un ejemplo de un resultado de un mapa de tamaño 9 laberinto.

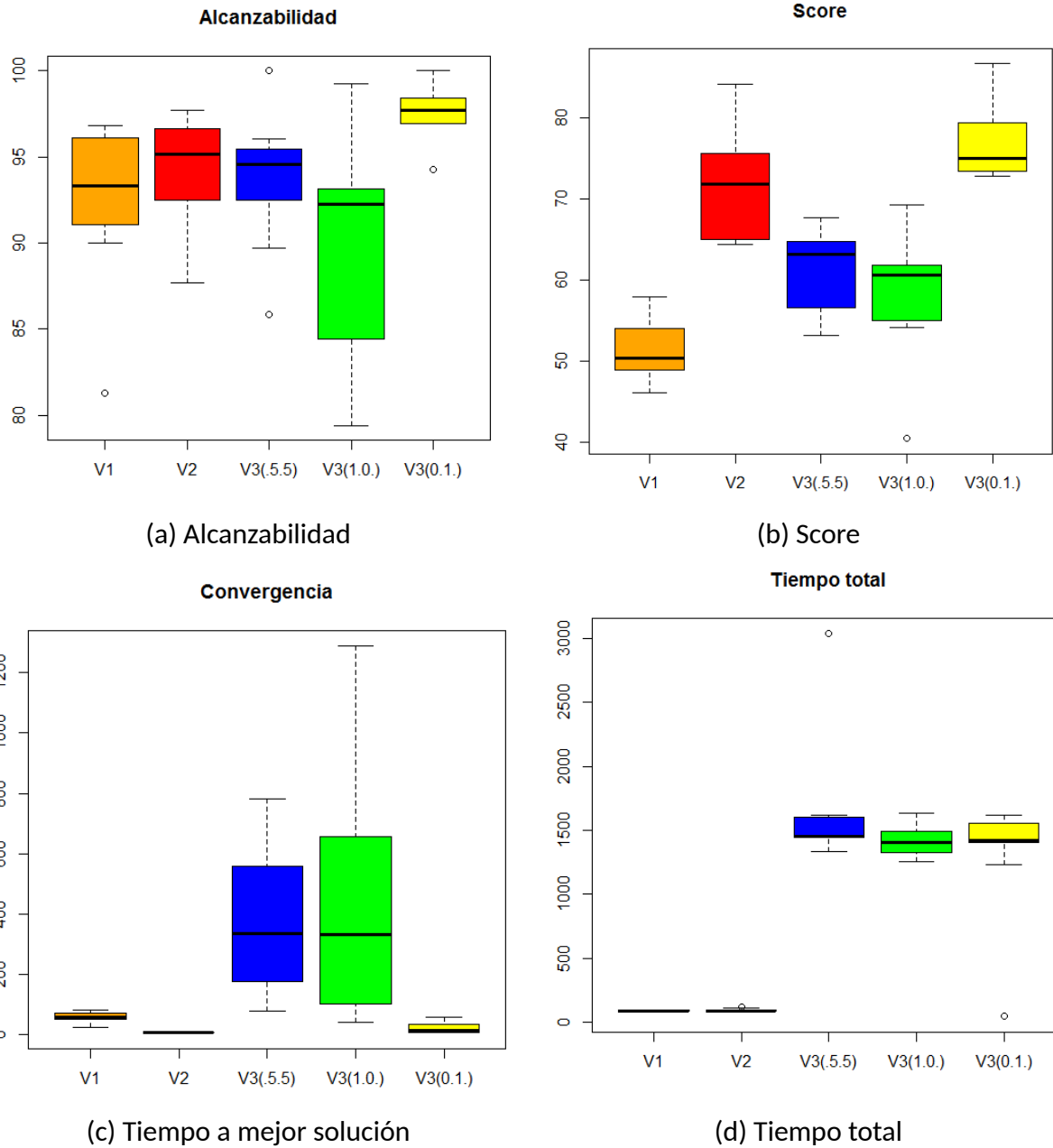


Figura 27: Comparación entre las 3 fases con configuración laberinto con mapas de tamaño 9.

Fuente: Elaboración propia.



Figura 28: Ejemplo del resultado de un mapa con configuración laberinto con mapas de tamaño 9.

Fuente: Elaboración propia.

La alcanzabilidad en los mapas de laberinto, por lo usual es un poco inferior a los de configuración estándar, lo cual es esperable, sobre todo en los mapas de V3(1. 0.) que no utilizan la alcanzabilidad, esto se cumplió en la mayoría de los casos y este experimento no fue la excepción. La versión del algoritmo genético que solo utiliza la alcanzabilidad siempre obtuvo la mejor alcanzabilidad, lo cual también es de esperarse, sobre todo en cuanto mas grande es el mapa, ya que más fácil es llegar a una celda en específico.

Por otro lado, los score también suelen ser inferior a los estándar, ya que es más difícil que el algoritmo cumple todas las condiciones, y este experimento no fue la excepción. Al requerir muchas celdas con 2 paredes y pocas del resto, sin importar cuanto se mejorara el algoritmo, no se obtuvieron mejores resultados que en la fase 1, probablemente debido a que de la fase 2 en adelante se empezaron a considerar soluciones infactibles.

Por último, los tiempos de convergencia y totales también son muy elevados en la fase 3, e incluso una vez se obtuvo un outlier de más de 50 minutos de espera. Por otro lado,

los tiempos de espera totales en ambas configuraciones es similar, mientras que la de convergencia de este experimento es mayor, debido a que el cambiar una celda pared en un laberinto hace que los puntajes puedan cambiar fácilmente, haciendo más difícil al algoritmo el llegar a la mejor solución posible.

En la figura 28 se puede ver que el mapa tiene más paredes conectadas, pero aun así no da la impresión de ser un laberinto, lo cual se debe al gran tamaño que tiene. Por otro lado, la salida está bien oculta, en donde hay que pasar por varias espinas para llegar, lo que claramente agrega dificultad al mapa, en cambio, su contra parte estándar, se enfoca en tener un poco de todo, sin importar si tienen mucho sentido o no, esta configuración le da un mejor uso a las puertas, lo cual no se logró con la estándar. Todo esto conlleva a que al igual que los mapas pequeños o medianos que tenían sus propios propósitos, este tipo de mapa sea ideal para los últimos niveles de un juego por su gran tamaño y buena ubicación de celdas, siendo más apto para esto esta configuración que la anterior.

Se puede ver con todos estos experimentos que la generación automática de contenido en videojuegos es factible, y puede dar mejores resultados o más interesantes que contenido creado por personas. Por otro lado, se puede crear todo tipo de contenido, enemigos, objetos, mapas, npc, trampas, misiones, etc. siempre y cuando se tengan los conocimientos de que debe tener cada uno de estos elementos para poder ser interesante y así poder aplicar las técnicas de generación automática de manera eficiente y efectiva, porque de lo contrario el contenido generado sería de peor calidad que si lo hiciera una persona. Además es una buena idea utilizar generación automática de contenido, ya que ahorra recursos a la hora de crear un videojuego, logrando así asignar estos recursos a mejorar otros aspectos del juego.

De los 3 tamaños y 2 configuraciones que se utilizaron para estos experimentos, comparando los resultados obtenidos, para los mapas pequeños los cuales utilizan alrededor de 20 celdas, la configuración estándar es más apropiada para lo que quiere crear, tiene mejor alcanzabilidad y score por lo general en las 3 fases del algoritmo y el resultado final es más apropiado para un mapa de tipo dungeon y es ideal para los primeros niveles, en cuanto a los mapas medianos que utilizan alrededor de 71 celdas, ambas versiones, estándar y laberinto, obtuvieron resultados satisfactorios en cuanto a los mapas generados, logrando bien su objetivo en ambos casos. Si bien los resultados de la versión laberinto, en cuanto a datos son por lo general peores que la estándar, el mapa generado por este es lo suficientemente interesante para ser considerado para jugarse en este tipo de videojuegos y son ideales para los niveles intermedios de este. Por último, en los mapas grandes que utilizan alrededor de 125 celdas, al ser tan grandes, la configuración estándar, a pesar de tener mejores resultados en los datos como la alcanzabilidad o el score, deja algo que desear al perder un poco el sentido la colocación de puertas, mientras que la versión laberinto arregla esto al colocar varias paredes, generando mapas mucho más interesantes para ser colocados como últimos niveles de un juego, pero la fase 3 al tener tiempos muy elevados, queda descartada en esta configuración y tamaño, por lo que la fase 1 o 2 serían las utilizadas en un videojuego.

Resumen

En este capítulo se vieron los experimentos realizados para validar la propuesta descrita en el capítulo anterior, para esto se consideraron 3 tamaños de mapas y 2 configuraciones de parámetros de celdas para las 3 fases del algoritmo. Los tamaños son pequeños, con alrededor de 20 celdas, tamaño mediano, con alrededor de 71 celdas y tamaño grande, con alrededor de 125 celdas. Las configuraciones son la estándar, donde todas las celdas tiene un 20 % del total de celdas y la configuración laberinto, en donde la celda con 2 paredes tienen un 55 % del total de celdas. Para la fase 3 se realizaron 3 versiones según sus α y β , las cuales son V3(.5 .5), V3(1. 0.) y V3(0. 1.). Con todo esto decidido, se utilizaron 4 indicadores para poder comparar los distintos experimentos, los cuales son: El tiempo total en realizar una ejecución completa, el tiempo en encontrar la mejor solución, denominado convergencia, la alcanzabilidad y por último el score obtenido. Con todo esto decidido, se procedió a realizar los experimentos, en donde se pueden ver los resultados obtenidos en las siguientes figuras: 17, 19, 21, 23, 25 y 27 y un ejemplo de los mapas generados para cada configuración y tamaño de mapa en las siguientes figuras: 18, 20, 22, 24, 26 y 28.

CAPÍTULO 5

CONCLUSIONES

En este trabajo de memoria se diseñó y se implementó un Procedural Content Generation - Generación de Contenido Procedural (PCG) utilizando un Algoritmo Genético para resolver el problema de la generación automática de mapas. Este algoritmo consiste en una población de soluciones que van evolucionando a través de generaciones las cuales constan con las etapas de evaluación, elitismo, selección, cruzamiento y mutación, con las cuales se logra encontrar soluciones cada vez mejores e ir seleccionando la más apta al final de cada generación.

Entre los aportes que se hicieron a esta propuesta de solución se encuentra la función objetivo a evaluar, la cual se fue modificando en las 3 fases del algoritmo, partiendo por algo simple y terminando en una ponderación entre 2 funciones, en donde se puede concluir de los resultados obtenidos, que dependiendo de si se quiere minimizar el score y los tiempos de ejecución y convergencia o maximizar la alcanzabilidad, es mejor una versión u otra, por ejemplo, los score de la fase 3 son los mejores para mapas pequeños y medianos mientras que los tiempos de convergencia de la fase 2 son superiores al resto en todos los tamaños y la alcanzabilidad del último experimento de la fase 3 es superior en todos los tamaños. Lo bueno de estas funciones es que al no ser muy complejas, si en el futuro se quiere agregar otro tipo de celda o quitar alguna o favorecer otras, simplemente se agrega, se quita o se modifica la función según sea el caso sin un mayor coste, incluso se podría agregar una tercera función, agrando un gamma para la fase 3 o simplemente evaluándola por separado como en la fase 2.

También se implementó de manera diferente la forma en que se crean las celdas, en esta propuesta no existen salas predefinidas, todo se crea dentro del algoritmo y las celdas se van colocando alrededor de la anterior, por lo que lo único que se mantiene es que el jugador parte aproximadamente al centro del mapa, mientras que el resto del mapa siempre es distinto.

Este algoritmo mostró buen rendimiento en soluciones pequeñas y medianas, para las fases 1 y 2, teniendo tiempos de respuesta muy pequeños y aceptables para un videojuego, mientras que en la fase 3 no son óptimas, pero pueden ser usables en ciertos casos, por lo que son aceptables, pero es en las soluciones grandes en donde deja mucho que desear, obteniendo tiempos de respuesta de varios minutos, sobre todo en la fase 3, en donde no son aceptables por sus grandes tiempos. Esto se debe a que los árboles que se usaron para representar las soluciones crecen en gran medida con el tamaño del mapa y recorrer los árboles enteros se vuelve costoso y en la fase 3 además se calcula la alcanzabilidad para todas las soluciones, lo cual también se vuelve más costoso mientras más grande es el árbol, lo que hace que los tiempos crezcan casi exponencialmente.

Esto se podría mejorar generando árboles no completos, por ejemplo, que cada rama solo

tuviera 1 hijo directo en vez de 3 o que fuera completo hasta cierto nivel y luego incompletos, lo que generaría una menor carga a la hora de recorrer y trabajar sobre los árboles, ya que son las operaciones alrededor de éstos el cuello de botella del algoritmo. Otra cosa que se podría mejorar son las veces necesarias que se lee el árbol completo de manera de realizar menos operaciones costosas como por ejemplo la forma en que se evalúan las soluciones, es decir, como se calcula cada score o verificar si es una solución factible, que no se requiera leer el árbol completo para ambas operaciones o incluso la forma en que se calcula la alcanzabilidad, teniendo memoria de que se modificó la última vez, para así no calcular partes que ya se saben que son alcanzables y solo calcular las modificaciones.

En trabajos futuros se podría ver cómo mejorar los aspectos recién mencionados realizando un conjunto de operaciones al mismo tiempo para reducir la lectura de los árboles o reducir la densidad de estos sin perder información, por ejemplo no colocando nodos que ya han sido colocados anteriormente en él. Otro trabajo a futuro posible es el hacer que el mapa no se genere alrededor del jugador, de manera de que éste no sepa donde parte al iniciar un nivel, aumentando la dificultad del juego o se podría mejorar la función objetivo para que el algoritmo favorezca pasillos más largos y el jugador no pueda escapar de situaciones de peligro tan fácilmente o agregando nuevas funciones para medir otros temas, como lo que se hizo con la alcanzabilidad.

REFERENCIAS BIBLIOGRÁFICAS

- [H-a, 2009] (2009). Hunt-and-kill algorithm. <http://wiki.roblox.com/index.php?title=Hunt-and-Kill>. Accessed: 2018-06-25.
- [Ont, 2009] (2009). Ontogenetic algorithms for procedural content generation. <http://pcg.wikidot.com/pcg-algorithm:ontogenetic>. Accessed: 2019-03-11.
- [Tel, 2009] (2009). Teleological algorithms for procedural content generation. <http://pcg.wikidot.com/pcg-algorithm:teleological>. Accessed: 2019-03-11.
- [alg, 2014] (2014). Algorithms for procedural content generation. <http://pcg.wikidot.com/category-pcg-algorithms>. Accessed: 2018-06-25.
- [Amato, 2017] Amato, A. (2017). Procedural content generation in the game industry. En *Game Dynamics*, pp. 15–25. Springer.
- [Bhojan y Wong, 2016] Bhojan, A. y Wong, H. W. (2016). TīTāl-asynchronous multiplayer shooter with procedurally generated maps. *Entertainment Computing*, 16:81–93.
- [Booth, 2009] Booth, M. (2009). The ai systems of left 4 dead.
- [Buck, 2003] Buck, J. (2003). Random dungeon design. <http://makingbrowsergames.com/starterkits/adventure/RandomDungeonGenerator.pdf>.
- [Cabrera et al., 2015] Cabrera, R. L., Collazo, M. N., Porras, C. C., y Leiva, A. J. F. (2015). Procedural content generation for real-time strategy games. *IJIMAI*, 3(2):40–48.
- [Hastings et al., 2009] Hastings, E. J., Guha, R. K., y Stanley, K. O. (2009). Evolving content in the galactic arms race video game. En *2009 IEEE Symposium on Computational Intelligence and Games*, pp. 241–248. IEEE.
- [Johnson et al., 2010] Johnson, L., Yannakakis, G. N., y Togelius, J. (2010). Cellular automata for real-time generation of infinite cave levels. En *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, p. 10. ACM.
- [Merrell et al., 2010] Merrell, P., Schkufza, E., y Koltun, V. (2010). Computer-generated residential building layouts. En *ACM Transactions on Graphics (TOG)*, volumen 29, p. 181. ACM.
- [Pedreño Moya, 2014] Pedreño Moya, V. (2014). Generació procedural de mapes per a un jrpg.
- [Pell, 1992] Pell, B. (1992). Metagame in symmetric chess-like games.
- [Roden y Parberry, 2004] Roden, T. y Parberry, I. (2004). From artistry to automation: A structured methodology for procedural content creation. En *International Conference on Entertainment Computing*, pp. 151–156. Springer.

- [Shaker *et al.*, 2012] Shaker, N., Nicolau, M., Yannakakis, G. N., Togelius, J., y O'Neill, M. (2012). Evolving levels for super mario bros using grammatical evolution. En *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 304–311. IEEE.
- [Shaker *et al.*, 2016] Shaker, N., Togelius, J., y Nelson, M. J. (2016). *Procedural content generation in games*. Springer.
- [Togelius *et al.*, 2012] Togelius, J., Justinussen, T., y Hartzen, A. (2012). Compositional procedural content generation. En *Proceedings of the The third workshop on Procedural Content Generation in Games*, p. 16. ACM.
- [Valtchanov y Brown, 2012] Valtchanov, V. y Brown, J. A. (2012). Evolving dungeon crawler levels with relative placement. En *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, pp. 27–35. ACM.
- [Van der Linden *et al.*, 2013] Van der Linden, R., Lopes, R., y Bidarra, R. (2013). Designing procedurally generated levels. En *Proceedings of the the second workshop on Artificial Intelligence in the Game Design Process*.
- [van der Linden *et al.*, 2014] van der Linden, R., Lopes, R., y Bidarra, R. (2014). Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89.