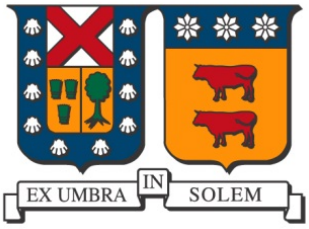
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



2-Step Evolutionary Algorithm for the generation of dungeons with lock door missions using horizontal symmetry

Felipe Antonio Dumont Ortiz

Tesis desarrollada como requisito parcial para la obtención del grado
Magister en Ciencias de la Ingeniería Informática

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE

TITULO DE TESIS

2-Step Evolutionary Algorithm for the generation of dungeons with lock door missions using horizontal symmetry

AUTOR
Felipe Antonio Dumont Ortiz

Tesis desarrollada como requisito parcial para la obtención del grado
Magister en Ciencias de la Ingeniería Informática

Dra. María Cristina Riff Rojas	_____ Profesor Guía
Dra Elizabeth Montero	_____ Correferente
Dr Ignacio Araya	_____ Correferente Externo
Dr Claudio Torres	_____ Presidente Comisión de Examen

Valpraraíso, Chile
Noviembre - 2024

DEDICATORIA

A mi familia y a la familia de mi polola y mi polola Gabriella, por su apoyo incondicional y constante durante todo este proceso. Su aliento y comprensión han sido fundamentales para alcanzar esta meta. Gracias por estar siempre ahí y creer en mí.

AGRADECIMIENTOS

Quiero expresar mi más profundo agradecimiento a Leonardo Pereira, por su invaluable colaboración y aportes al desarrollo de este trabajo, su conocimiento y disposición para ayudar han sido cruciales para la culminación de este proyecto y la comprensión del algoritmo Constrained Evolutionary Algorithm para la generación de mazmorras con barreras y llaves. A mi profesora guía, María Cristina Riff, le extiendo mi más sincero agradecimiento por su dedicación y apoyo constante. Su orientación me ha permitido comprender mejor el proceso de desarrollo de un paper, así como los elementos clave que deben ser resaltados y recalados al presentarlo. Su experiencia y consejos han sido fundamentales para la realización exitosa de esta tesis.

RESUMEN

Resumen— Esta tesis examina la generación procedural de contenido en videojuegos, enfocándose en la creación de niveles y mazmorras. Primero, se analizan las técnicas generales de generación procedural, seguidas de un estudio más detallado sobre la generación de niveles. El trabajo principal se centra en un Algoritmo Evolutivo de 2 Pasos (2-Step EA) para generar mazmorras con misiones de llaves y barreras. La primera fase del algoritmo configura el diseño básico de la mazmorra, y la segunda fase asigna llaves y barreras estratégicamente. Los resultados muestran que el algoritmo propuesto es efectivo en producir mazmorras de alta calidad y eficientes computacionalmente. Este enfoque ofrece una metodología sólida para la generación procedural de contenido dinámico.

Palabras Clave—Algoritmos Evolutivos, Videojuegos, Entretenimiento y Medios.

ABSTRACT

***Abstract*—** This thesis examines procedural content generation in video games, focusing on level and dungeon creation. Initially, general procedural generation techniques are analyzed, followed by a more detailed study on level generation. The main work centers on a 2-Step Evolutionary Algorithm (2-Step EA) for generating dungeons with locked door missions. The first phase of the algorithm configures the basic dungeon layout, while the second phase strategically assigns keys and barriers. The results demonstrate that the proposed algorithm effectively produces high-quality and computationally efficient dungeons. This approach offers a robust methodology for dynamic and entertaining procedural content generation in video games.

***Keywords*—**Evolutionary algorithms, Games, Entertainment and media

GLOSARIO

PCG: Procedural Content Generation, es decir, la Generación Procedural de Contenidos, que se refiere a la creación automática de contenido a través de algoritmos.

PDG: Procedural Dungeon Generation, o Generación Procedural de Mazmorras, que se centra en la creación automática de niveles de mazmorras.

CEA: Constrained Evolutionary Algorithms, o Algoritmos Evolutivos Restringidos, que se utilizan para la optimización de problemas complejos mediante técnicas inspiradas en la evolución biológica, considerando restricciones específicas del problema.

GA: Genetic Algorithm, o Algoritmo Genético, una técnica de optimización heurística basada en los principios de la evolución natural y selección genética.

EA: Evolutionary Algorithms, o Algoritmos Evolutivos, una categoría de algoritmos utilizados en la optimización de problemas complejos mediante técnicas inspiradas en la evolución biológica.

Semilla: Utilizada en la generación procedural para asegurar que el contenido generado pueda ser reproducido exactamente igual en futuras ejecuciones.

Replayability: Rejugabilidad, la capacidad de un juego para seguir siendo interesante y divertido después de múltiples partidas.

Simetría: Propiedad de una mazmorra generada donde ciertas partes de la estructura son reflejadas o replicadas, creando un equilibrio visual y de diseño.

Mazmorra: Entorno subterráneo generado proceduralmente, comúnmente encontrado en juegos de rol y aventura.

Llave y Barrera: Mecánica de juego donde el jugador debe encontrar llaves para desbloquear puertas o barreras que bloquean el progreso.

Dimensionalidad: Número de variables independientes que definen una solución en el espacio de búsqueda.

Espacio de Búsqueda: Conjunto de todas las posibles soluciones en un problema de optimización.

Grafo: Estructura de datos utilizada para representar la disposición de habitaciones y pasillos en una mazmorra.

Automata Celular Técnica utilizada en la generación procedural de mazmorras que emplea reglas locales aplicadas a una cuadrícula de celdas.

Gramáticas Generativas: Método que utiliza reglas gramaticales para generar contenido, como estructuras de mazmorras.

Modelos de Aprendizaje Automático: Algoritmos que aprenden patrones a partir de datos y se utilizan en la generación procedural para crear contenido adaptativo.

Crossover (Cruce:) Operación en algoritmos genéticos donde dos individuos se combinan para producir descendencia.

Mutación: Operación en algoritmos genéticos donde se introducen cambios aleatorios en algunos individuos para mantener la diversidad genética.

Selección: Proceso en algoritmos genéticos donde se eligen los individuos más aptos para reproducirse.

Evaluación: Proceso en algoritmos genéticos donde se mide la calidad de las soluciones candidatas mediante una función de aptitud.

Inicialización: Etapa en algoritmos genéticos donde se crea la población inicial de individuos de manera aleatoria.

Reemplazo: Proceso en algoritmos genéticos donde la nueva generación de individuos reemplaza a la anterior.

ÍNDICE DE CONTENIDOS

RESUMEN iv

ABSTRACT iv

GLOSARIO v

ÍNDICE DE FIGURAS xi

ÍNDICE DE TABLAS xiii

ÍNDICE DE ALGORITMOS xiii

INTRODUCCIÓN 1

CAPÍTULO 1: Creación de Mazmorras en Videojuegos 3
1.1 Descripción del Proceso Manual . . . 3
1.1.1 Herramientas y Métodos . . . 3
1.1.2 Proceso de Diseño . . . 4
1.1.3 Uso de Tiles . . . 4
1.2 Actores Involucrados . . . 6
1.2.1 Objetivos de Diseño . . . 7
1.3 Dificultades en la Creación de Mazmorras . . . 8
1.4 Generación Procedural de Contenido (PCG) en Juegos . . . 9
1.5 Aplicaciones de la PCG en Videojuegos . . . 9
1.5.1 Juegos de Mazmorras . . . 10
1.5.2 Juegos de Mundo Abierto . . . 12
1.6 Historia de la PCG . . . 13
1.7 Beneficios de la PCG . . . 14
1.8 Desafíos Actuales y su Impacto . . . 15
1.9 Competencia Actual y Objetivos de la Solución . . . 15
1.10 Generación Procedural de Mazmorras (PDG) . . . 16
1.10.1 Métodos de Generación . . . 16
1.10.2 Ejemplo en Juegos . . . 17
1.11 Resumen . . . 17

CAPÍTULO 2: Formulación del Problema 19
2.1 Habitaciones inicial y final . . . 19
2.2 Restricciones de configuración . . . 19
2.3 Restricciones de conectividad . . . 20
2.4 Relaciones entre llaves y barreras . . . 21
2.5 Diferenciabilidad de disposiciones . . . 21
2.6 Características Escogidas por el Diseñador . . . 22
2.7 Impacto de las Características en el Problema . . . 25

2.8 Abstracción del Problema . . . 25
2.9 Visualización de la Disposición y el Grafo . . . 26
2.10 Disposición de Mazmorra . . . 26
2.11 Resumen . . . 26

CAPÍTULO 3: Estado del Arte **29**
3.1 Generación Procedural de Mazmorras . . . 29
3.2 Generación procedural de niveles y Mazmorras . . . 30
3.2.1 Técnicas Generativas . . . 30
3.2.2 Técnicas Evolutivas . . . 32
3.2.3 Aspectos relevantes en la generación de Mazmorras . . . 41
3.2.3.1 Mecánicas de Llave y Barrera . . . 41
3.2.3.2 Aplicación de Simetría . . . 41
3.2.4 Otras Técnicas . . . 43
3.3 Representaciones en la Generación Procedural de Mazmorras . . . 44
3.4 Resumen . . . 45

CAPÍTULO 4: Algoritmos Evolutivos **47**
4.1 Introducción . . . 47
4.2 Algoritmos Genéticos (GA) . . . 47
4.3 Algoritmos Evolutivos (EA) . . . 48
4.4 Espacio de Búsqueda . . . 48
4.4.1 Dimensionalidad del Espacio de Búsqueda . . . 48
4.5 Aplicaciones en la Generación de Mazmorras . . . 49
4.5.1 División en Dos Fases del Espacio de Búsqueda . . . 49
4.6 Resumen . . . 52

CAPÍTULO 5: Detalles Técnicos del Algoritmo Propuesto **53**
5.1 Representación . . . 53
5.1.1 Ventajas de la Representación Basada en Grafos . . . 54
5.2 Función de Evaluación . . . 56
5.2.1 Aplicación de la Función de evaluación en cada fase . . . 56
5.3 Estructura del Algoritmo . . . 57
5.3.1 Fase 1: Generación estructural . . . 58
5.3.1.1 Ejemplo de Generación . . . 59
5.3.2 Población Inicial . . . 61
5.3.2.1 Operadores Genéticos de la fase 1 . . . 62
5.3.2.1.1 Crossover . . . 62
5.3.2.1.2 Mutación de Simetría . . . 64
5.3.2.1.3 Mutación de adición de habitación . . . 65
5.3.2.1.4 Mutación de eliminación de habitación . . . 66
5.3.3 Fase 2: Alocación de llaves y Barreras . . . 71
5.3.3.1 Distribución Inicial de Llaves y Barreras . . . 73
5.3.3.2 Validación de Distribución . . . 74

5.3.4 Transformación . . . 75
5.3.4.1 Mutación de eliminación de llave-barrera . . . 75
5.3.4.2 Mutación de adición de llave-barrera . . . 76
5.4 Conclusiones . . . 78

CAPÍTULO 6: Evaluación y Comparación **79**
6.1 Métricas de Evaluación . . . 79
6.1.1 Resultados de la Evaluación . . . 80
6.2 Algoritmo Evolutivo Restringido (CEA) . . . 82
6.2.1 Funcionamiento del CEA . . . 83
6.2.2 Ventajas del CEA sin DFS . . . 83
6.2.3 Implementación del Algoritmo Original y Comparación entre Versiones 84
6.3 Configuración Experimental . . . 85
6.4 Resultados de la Comparación . . . 85
6.4.1 Calidad de las Mazmorras Generadas . . . 85
6.4.2 Eficiencia Computacional . . . 86
6.4.3 Comparación de mapas de calor de los niveles Generados . . . 86
6.5 Proceso de Ajuste . . . 86
6.5.1 Ajuste de la Generación de la Forma . . . 87
6.5.2 Ajuste de la Evolución de Barreras y Llaves . . . 88
6.5.3 Importancia de las Probabilidades de Mutación . . . 88
6.6 Resultados . . . 89
6.6.1 Análisis del Coeficiente Lineal . . . 90
6.7 Análisis con el Test de Wilcoxon . . . 94
6.8 Conclusiones . . . 95

CAPÍTULO 7: Análisis de mapas generados **97**
7.1 Mapas Generados para Cada Problema . . . 97
7.2 Análisis de Linealidad en Mapas Generados . . . 100
7.2.1 Sentido de 1.0 como coeficiente lineal . . . 101
7.3 Conclusión del capítulo . . . 101

CAPÍTULO 8: Conclusiones **103**
8.1 Alcances y Limitaciones . . . 103
8.1.1 Complejidad y Adaptabilidad: . . . 103
8.1.2 Naturaleza de los Nodos y Habitaciones: . . . 104
8.2 Resultados y Contribuciones . . . 104
8.3 Validez de los Objetivos . . . 104
8.4 Impacto y Aplicaciones . . . 105
8.5 Conclusiones sobre el uso del Algoritmo Evolutivo y la Representación Actual 105
8.6 Recomendaciones y Futuras Líneas de Investigación . . . 106
8.7 Aplicaciones Prácticas en el Desarrollo de Videojuegos . . . 107
8.8 Impacto en la Experiencia del Jugador . . . 107
8.9 Enfoque Interdisciplinario y Colaboración . . . 107

REFERENCIAS BIBLIOGRÁFICAS **A**

ÍNDICE DE FIGURAS

1 Juego *Mario Maker* en el modo de edición de niveles. 5
2 **diseñador de juegos** y su interacción con el mundo conceptual para crear las reglas del juego y cómo los usuarios interactúan el este. 7
3 **diseñador de niveles** el que utiliza los conceptos creados por el diseñador de juegos, en las reglas conceptuales del juego para poder crear niveles y ver la progresión del juego en estos mismos. 7
4 Generación de mundo en *Dwarf Fortress*. 10
5 Nivel del juego *The Binding of Isaac*, mostrando la combinación de diseño manual y generación procedural. 11
6 Ejemplo de una mazmorra generada proceduralmente en *Diablo 3*. 12
7 Ejemplo de un mundo generado en *Minecraft*. 13
8 Ejemplo de una disposición de mazmorra en 2D. 27
9 Representación en forma de grafo de la disposición de mazmorra. Los nodos representan habitaciones y las aristas representan las conexiones entre habitaciones. 27
10 Separación de nivel en dos elementos, mission y espacio por Dormants . . . 31
11 Reglas de gramática de forma para generar las misiones 32
12 Reglas de CA: A la izquierda, sin posibilidad de agregar habitaciones; a la derecha, la habitación activa (X) con las direcciones de búsqueda. 32
13 Caverna generada por Ashlock et al. 33
14 Pasos para la generación de niveles de Spelunky: evolución de la gráfica, habitaciones principales y habitaciones de libre selección para llegar al mapa final. 34
15 Actuar de las personas generadas en comparación con los agentes 34
16 Pasos para la generación de niveles de mazmorras evolucionando segmentos desde la representación al mapa final 36
17 Representación de nivel en formato grafo 37
18 Representación de nivel conectado como red 38
19 Visualización del Evolutionary Dungeon Designer (EDD), y opciones de similitud en base a las métricas seleccionadas 39
20 Representación de los niveles con llaves y barreras por genes 40
21 Representación de los niveles con llaves y barreras divididos por sectores, nivel factible o infactible 40
22 Uso de simetría completa, simetría con respecto a los ejes horizontales, verticales y con respecto al punto central. 42
23 Espacio generado en base a la asignación de la línea roja, que guía la generación del nivel en base a estructura 44
24 Espacio de búsqueda, enfocado en la forma del mapa 50
25 Espacio de búsqueda, enfocado en la selección de un subespacio de búsqueda para la segunda fase. 51
26 Espacio de búsqueda, enfocado en la ubicación de llaves y barreras en el mapa. 51

27 Representación de un laberinto, indicando las habitaciones a la izquierda (L), abajo (D) y derecha (R) de cada nodos, partiendo desde el nodo P. Configuración [7, 0, 0, 3.0, 0] 60
28 Crossover, los padres se encuentran a la izquierda generando las mazmorras hijas mostradas a la derecha 63
29 Ejemplo de movimiento simétrico, desde el nodo padre P, utilizando la simetría entre izquierda y derecha de los nodos hijo 65
30 Mutación de adición de habitación, a la izquierda se presenta el laberinto inicial, al cual se le agrega un nodo marcado en azúl 67
31 Mutación de eliminación de habitación, a la izquierda se presenta el laberinto inicial, al cual se le remueve un nodo superior demarcado de color azúl . . . 68
32 Ejemplo de un laberinto imposible de pasar dada la distribución de llaves y barreras 74
33 Ejemplo de la mutación de eliminación de llaves, el cual elimina la barrera 1 y la llave 1 76
34 Ejemplo de la mutación de adición de llave-barrera, se agrega una nueva combinación llave y barrera dentro del laberinto 78
35 Ejemplo de una mazmorra generada utilizando el algoritmo 2-STEP EA, mostrando la distribución de habitaciones, llaves y barreras para un mapa con la siguiente configuración: [15,3,2,2.0,2] 81
36 Metricas de comparación del Algoritmo Evolutivo Restringido (CEA) sin DFS y con DFS 84
37 Boxplot del Fitness resultante en el problema [20,4,4,X,4], con un coeficiente Linear (X) variable entre 1.0, 2.0, y 3.0, comparando el algoritmo Constrained Evolutionary Algorithm (CEA) con el 2-Step EA. 92
38 Mapa de calor para visualizar la ubicación de las habitaciones dentro de un laberinto por CEA, donde el eje y representa la posición en Y y el eje X representa la posición de la habitación en X, considerando la primera habitación como la habitación (0,0). Las generaciones fueron realizadas para el problema [20,4,4,X,4], con un coeficiente Linear (X) variable. 93
39 Mapa de calor para visualizar la ubicación de las habitaciones dentro de un laberinto por 2-StepEA, donde el eje y representa la posición en Y y el eje X representa la posición de la habitación en X, considerando la primera habitación como la habitación (0,0). Las generaciones fueron realizadas para el problema [20,4,4,X,4], con un coeficiente Linear (X) variable. 94
40 Mapas generados por el algoritmo 2-StepEA para los problemas (15,3,2,2.0,2.0) y (20,4,4,1.0,4.0). 98
41 Mapas generados por el algoritmo 2-StepEA para los problemas (20,4,4,2.0,4.0) y (25,8,8,1.0,8.0). 99
42 Mapas generados por el algoritmo 2-StepEA para los problemas (30,4,4,2.0,4.0) y (30,6,6,1.5,6.0) 99
43 Mapas generados por el algoritmo 2-StepEA para el problema (15,2,2, Lc, 2) con coeficientes lineales 1.0 y 2.5. 100

44 Mapas generados por el algoritmo 2-StepEA para el problema (15,2,2,Lc,2) con coeficientes lineales 2.0 y 2.5. 101
45 Mapa generados por el algoritmo 2-StepEA para el problema (15,2,2,3.0,2) . 102

ÍNDICE DE TABLAS

1 Pruebas de configuraciones de generación de mazmorras especificando los requisitos de habitaciones, llaves, barreras y llaves necesarias. 23
2 Resumen comparativo de técnicas evolutivas. 46
3 Resumen comparativo de técnicas. 46
4 Requerimientos de habitaciones, llaves, barreras y coeficiente lineal para la generación de un mapa. 59
5 Comparación de la calidad de las mazmorras generadas. 86
6 Ejemplo de comparación de la eficiencia computacional de los algoritmos. . . 86
7 Configuraciones utilizados para sintonizar el algoritmo. 88
8 Parámetros ajustados para el Algoritmo Evolutivo de Dos Pasos. 89
9 Métricas principales de la mazmorra, considerando el número de habitaciones, llaves y barreras, entre el 2-Step EA y el Algoritmo Evolutivo Restringido (CEA). 91
10 Coeficiente Lineal de la mazmorra, aptitud obtenida y tiempo en segundos para que el algoritmo genere la mazmorra, entre el 2-Step EA y el Algoritmo Evolutivo Restringido (CEA). 91
11 Wilcoxon test results. Bold values indicate statistical significance in the obtained results. 95
12 Configuraciones de usuario. 97

List of Algorithms

1 2-Step Evolutionary Algorithm 58
2 Generate Initial Map 62
3 Crossover Algorithm 64
4 Add Node Mutation Algorithm 66
5 Remove Node Mutation Algorithm 68
6 Generación Estructural 70
7 Evolve BK Algorithm 72
8 Distribución Inicia 74
9 MutationRemoveKey 76
10 MutationAddKey 77

INTRODUCCIÓN

En la última década, la industria de los videojuegos ha experimentado un crecimiento explosivo, superando los 50 mil millones de dólares en ingresos en 2020 y ampliándose aún más a más de 60 mil millones de dólares en 2021 [ESA, 2022]. Las previsiones sugerían cifras más altas para 2025 y más allá [newzoo, 2023], pero hubo una caída de los ingresos en el mercado de los juegos en los años 2022 con malas previsiones para 2023, con una recuperación prevista para 2025. Uno de los aspectos más desafiantes del desarrollo de videojuegos es la creación de contenido dinámico y variado que mantenga el interés del jugador a lo largo del tiempo. La generación procedural de contenido se ha consolidado como una técnica esencial para lograr este objetivo.

La generación procedural se refiere a la creación de datos de manera automática mediante algoritmos, en lugar de o como apoyo al diseño manual. En el contexto de los videojuegos, esto permite la creación de mundos, niveles y mazmorras únicas en cada partida, normalmente asociados a una semilla, la cual se puede utilizar para regenerar el mismo mundo a partir de esta, proporcionando una experiencia de juego fresca y repetible. Este enfoque no solo ahorra tiempo y recursos en el desarrollo, sino que también aumenta la rejugabilidad y el atractivo de los juegos.

Los niveles y mazmorras generados proceduralmente deben cumplir con ciertos criterios de diseño, tales como coherencia espacial, balance de las mecánicas que se le agregan a los niveles. La creación de estos elementos presenta numerosos desafíos, desde la estructura básica del diseño hasta la disposición de enemigos, obstáculos y recompensas. En particular, la generación de mazmorras plantea el problema de crear un entorno que sea interesante de explorar, equilibrado en términos de dificultad y que ofrezca desafíos significativos al jugador.

El problema específico abordado en esta tesis es la generación de mazmorras que no solo sean navegables y coherentes, sino que también presenten misiones de barreras, donde los jugadores deben encontrar llaves para progresar. Este tipo de mazmorras añade un nivel adicional de estrategia y planificación al juego, haciendo que la experiencia sea más inmersiva y desafiante, cumpliendo con las caracteristicas de diseño.

Para abordar este problema, se propone el uso de un Algoritmo Evolutivo de 2 Pasos (2-Step EA) para la creación de niveles con llaves y barreras. Este enfoque se divide en dos fases: la primera fase se dedica a la configuración del diseño de la forma de la mazmorra, considerando factores como el número de habitaciones y su disposición general. La segunda fase se enfoca en la colocación estratégica de llaves y puertas cerradas dentro del diseño generado,

asegurando que los jugadores deban superar desafíos específicos para progresar, y cerciorando que los niveles sean posibles de terminar y requieran de alcanzar todas las llaves para terminar el nivel.

La representación de los niveles generados se basa en una estructura de grafo, donde cada nodo representa una habitación y los enlaces representan conexiones entre ellas. Las llaves y barreras se distribuyen de manera que los jugadores deben explorar y resolver problemas de llaves y barreras para avanzar. Este diseño no solo mejora la jugabilidad, sino que también asegura una distribución equilibrada de los desafíos dentro de la mazmorra.

La metodología empleada se basa en algoritmos evolutivos, utilizando una población inicial de diseños de mazmorras que se mejora iterativamente a través de procesos de selección, mutación y cruce. La calidad de cada diseño se evalúa mediante una función de evaluación que considera la coherencia del diseño, la distribución de llaves y puertas, y la navegabilidad general de la mazmorra. Las dos fases del algoritmo aseguran tanto la estructura básica como los elementos de llaves y barreras.

Esta tesis se organiza de la siguiente manera: la **Creación de maznorras en videojuegos** donde se presenta el contexto general, la **Formulación del Problema** detalla el problema específico de la generación de mazmorras y los criterios de diseño considerados, el **Estado del Arte** revisa trabajos relacionados y discute diferentes enfoques en la generación procedural de contenido, seguido por **Algoritmos Evolutivos** donde se presenta una breve introducción a los algoritmos evolutivos, para luego avanzar hacia la **Detalles técnicos del algoritmo propuesto**, donde se describe la representación de los niveles, la distribución de llaves y barreras, y explica en detalle el **Algoritmo Evolutivo de 2 Pasos** y su implementación. Posteriormente, en **Evaluación y comparación** se presentan y analizan los resultados obtenidos con el algoritmo propuesto. Finalmente, en **Conclusiones y Trabajo Futuro** se resumen los hallazgos principales y se sugieren posibles direcciones para investigaciones futuras.

CAPÍTULO 1
Creación de Mazmorras en Videojuegos

La generación procedural de mazmorras es un campo crucial dentro del desarrollo de videojuegos. En particular, este trabajo se enfoca en mejorar la generación de mazmorras en videojuegos de rol (RPG), aventura y roguelike, donde las mazmorras actúan como entornos clave para la exploración y el progreso del jugador. La técnica de generación utilizada en este trabajo se basa en algoritmos evolutivos, con un enfoque particular en la simetría horizontal para la disposición de las mazmorras, pero para comprender de mejor manera la generación procedural de mazmorras, tenemos que introducir la generación manual y los métodos y herramientas existentes, para poder contextualizarnos con el problema abordado.

1.1. Descripción del Proceso Manual

La creación manual de niveles en videojuegos sigue siendo una práctica común y esencial para muchos títulos, especialmente aquellos que buscan ofrecer una experiencia de juego cuidadosamente diseñada y curada. Aunque la generación procedural ha ganado popularidad por su capacidad para generar contenido de manera automática y variada, la creación manual sigue siendo insustituible en ciertos contextos donde se busca una mayor coherencia narrativa, una dirección artística específica o una jugabilidad meticulosamente ajustada.

1.1.1. Herramientas y Métodos

En el proceso de creación manual de niveles, los desarrolladores utilizan una variedad de herramientas y métodos para diseñar entornos de juego atractivos y bien estructurados. Estas herramientas suelen incluir editores de niveles integrados en los motores de juego o software independiente especializado en diseño de niveles.

Los editores de niveles ofrecen una interfaz gráfica intuitiva que permite a los diseñadores colocar y manipular objetos, terrenos, personajes y otros elementos del juego en un entorno visualizado en tiempo real. Estos editores suelen proporcionar una variedad de herramientas de manipulación, como selección, transformación, pintura y pinceles de terreno, que permiten a los diseñadores dar forma al mundo del juego según sus especificaciones.

Además de las herramientas visuales, los diseñadores también pueden utilizar scripts y lenguajes de programación integrados en el motor de juego para implementar lógica de juego personalizada y funcionalidades específicas del nivel. Estos scripts pueden utilizarse para controlar el comportamiento de los enemigos, activar eventos de juego, gestionar la progresión del jugador y crear desafíos únicos dentro del nivel.

1.1.2. Proceso de Diseño

El proceso de diseño manual de niveles generalmente sigue varias etapas clave:

- **Conceptualización**: En esta etapa inicial, los diseñadores esbozan las ideas generales del nivel, incluyendo su temática, estética, y los objetivos de juego que se quieren lograr. Se crean bocetos y mapas conceptuales que guían las siguientes fases de desarrollo.
- **Bloqueo de Nivel (Blockout)**: Se crea una versión básica del nivel utilizando formas geométricas simples para definir la estructura general y el flujo del juego. Este paso permite a los diseñadores ajustar el diseño sin preocuparse por los detalles finos.
- **Detalles y Decoración**: Una vez que la estructura básica está establecida, se añaden detalles visuales y decorativos para dar vida al entorno del juego. Se incluyen texturas, modelos 3D detallados, iluminación y otros elementos estéticos que mejoran la inmersión.
- **Implementación de la Lógica de Juego**: En esta fase, se incorporan scripts y lógica de programación para definir la interacción del jugador con el entorno. Esto incluye la colocación de enemigos, trampas, puzzles y otros desafíos.
- **Pruebas y Ajustes**: Se realizan pruebas exhaustivas para identificar problemas y áreas de mejora en el nivel. Los diseñadores ajustan el diseño y la jugabilidad basándose en el feedback recibido, iterando sobre el nivel hasta alcanzar la calidad deseada.
- **Optimización**: Finalmente, se optimiza el nivel para asegurar un rendimiento fluido en diversas plataformas, ajustando elementos como la carga de texturas, la cantidad de objetos en pantalla y la eficiencia de los scripts.

1.1.3. Uso de Tiles

El proceso de creación de niveles requiere flexibilidad a la hora de colocar elementos y la reutilización de estos dentro de cada nivel. Por esta razón, el uso de tiles (mosaicos) es fundamental. Los tiles permiten a los diseñadores de niveles utilizar imágenes completas como guía para la creación de niveles, estableciendo reglas y funciones específicas para cada tile. Estas reglas de posicionamiento y función facilitan la creación de niveles, apoyada por programas y herramientas de desarrollo.

En el desarrollo de juegos de plataformas en 2D, los diseñadores utilizan conjuntos de tiles que representan diferentes partes del entorno del juego, como pisos, paredes y obstáculos. Estos tiles pueden ser manipulados manualmente dentro del editor de niveles para crear niveles detallados y específicos. Además, los mismos elementos utilizados para pintar los

niveles se pueden emplear en la generación procedural. En *Mario Maker* (Figura 1), estas herramientas permiten a los jugadores diseñar sus propios niveles, demostrando la versatilidad y utilidad de los tiles tanto en la creación manual como en la generación procedural.

Figura 1: Juego *Mario Maker* en el modo de edición de niveles.

El diseño manual en juegos de plataformas 2D implica la colocación precisa de tiles para crear plataformas, obstáculos, trampas y otros elementos del entorno del juego. Los diseñadores experimentan con diferentes disposiciones de tiles para encontrar la configuración óptima que proporcione un flujo de juego suave y desafiante para el jugador.

Además de la disposición de tiles, los diseñadores también pueden definir propiedades físicas y comportamientos para cada tile, como fricción, elasticidad, colisión e interacción con otros objetos del juego. Estas propiedades determinan cómo se mueve y se comporta el jugador dentro del entorno del juego, contribuyendo a la experiencia de juego general.

La creación de elementos con restricciones y funcionalidades agregadas permite el uso automatizado de tiles, apoyando el desarrollo e implementación de niveles dentro de los juegos. Esto facilita la creación rápida y eficiente de niveles complejos, mejorando la rejugabilidad y la variedad del contenido.

1.2. Actores Involucrados

La creación de mazmorras en videojuegos involucra a varios actores clave, cada uno de los cuales desempeña un papel fundamental en el desarrollo y la calidad del juego. Los principales actores involucrados son:

- **Diseñadores de Juegos**: Se encargan de planificar y conceptualizar los niveles, estableciendo los desafíos, la narrativa y la jugabilidad. Su objetivo es crear una experiencia atractiva y coherente para los jugadores y establecer medidas para crear los juegos basándose en su experiencia o estudios relacionados. Ellos establecen las reglas que dominarán el mundo del juego y los objetivos de los elementos dentro del juego.
- **Diseñadores de Niveles**: Son responsables de crear la estructura y el flujo de cada mazmorra o nivel en el juego. Trabajan en conjunto con los diseñadores de juegos para asegurar que los niveles reflejan las mecánicas y narrativa planeadas. Definen la disposición de los elementos jugables como obstáculos, enemigos, y objetos, asegurando que cada nivel sea desafiante y divertido, además de mantener el balance entre dificultad y progresión del jugador.
- **Programadores**: Implementan los algoritmos de generación procedural y otras herramientas técnicas necesarias para crear y gestionar las mazmorras. Son responsables de la funcionalidad y la optimización del código, dando vida al juego y a cada uno de sus elementos internos.
- **Artistas**: Diseñan los elementos visuales de las mazmorras, incluyendo texturas, modelos y efectos visuales. Su trabajo es crucial para mantener la cohesión temática y estética del juego, además de encargarse de la creación de las representaciones de niveles y las definiciones de los tiles que se utilizarán para la creación de los niveles.
- **Músicos y Diseñadores de Sonido**: Crean la música y los efectos de sonido que acompañan a las mazmorras, aumentando la inmersión y mejorando la experiencia general del jugador.
- **Otros Profesionales**: Incluyen testers, escritores y productores que contribuyen a diferentes aspectos del desarrollo, asegurando la calidad y la coherencia del producto final.
- **Jugadores**: Se benefician de mazmorras bien diseñadas que ofrecen una experiencia de juego rica y satisfactoria. Su feedback es vital para ajustar y mejorar el diseño del juego.

Para ilustrar el proceso de interacción entre los distintos actores, se presentarán a los actores de **diseñador de juegos** el cual interactúa con el juego en un nivel conceptual, definiendo las reglas y mecánicas que regirán el mundo del juego (ver Fig. 2) . A partir de estas mecánicas, el

diseñador de niveles, toma esas reglas y las aplica en un contexto más específico, creando los espacios donde los jugadores experimentarán las interacciones que el diseñador de juegos planeó (ver Fig. 3). Estas dos funciones están interconectadas, ya que los niveles diseñados dependen de la base de las mecánicas previamente definidas, y juntos crean una experiencia cohesiva y entretenida.

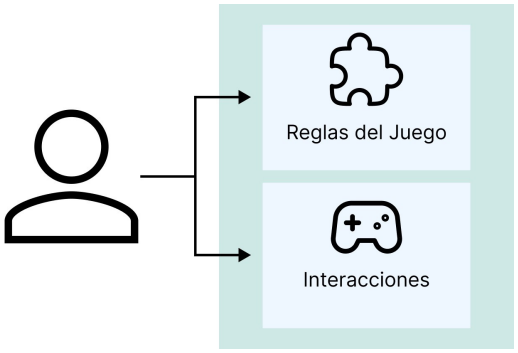


Figura 2: **diseñador de juegos** y su interacción con el mundo conceptual para crear las reglas del juego y cómo los usuarios interactúan el este.

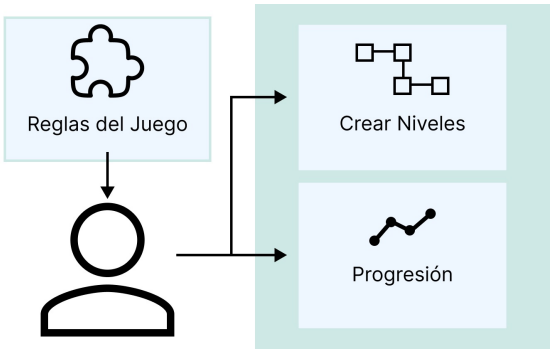


Figura 3: **diseñador de niveles** el que utiliza los conceptos creados por el diseñador de juegos, en las reglas conceptuales del juego para poder crear niveles y ver la progresión del juego en estos mismos.

1.2.1. Objetivos de Diseño

Los objetivos de diseño que pueden ser establecidos por un diseñador para la creación de niveles son diversos y pueden incluir:

- **Coherencia Narrativa**: Asegurar que el nivel se integre perfectamente con la historia y el universo del juego, proporcionando contexto y significado a las acciones del jugador.
- **Dirección Artística**: Mantener una estética visual y un estilo artístico consistentes que refuercen la atmósfera y la temática del juego.

- **Jugabilidad Ajustada**: Crear desafíos bien balanceados y progresiones de dificultad que mantengan al jugador comprometido y motivado.
- **Exploración y Descubrimiento**: Diseñar espacios que fomenten la curiosidad y recompensen la exploración, ofreciendo secretos y contenido opcional para enriquecer la experiencia del jugador.
- **Flujo de Juego**: Asegurar un flujo de juego suave y natural que guíe al jugador a través del nivel de manera intuitiva, evitando frustraciones y puntos muertos.

La estructuración de los niveles y mazmorras tiene un sentido y finalidad que proviene del diseñador, por lo que es muy importante que la comunicación entre programadores, artistas y diseñadores sea directa, para que la intención de la creación de un nivel tenga sentido en el contenido que se está creando. Por esta misma necesidad, la creación de herramientas que apoyen al diseño y evaluación rápida por parte del diseñador es crucial, para crear una coherencia estructural en las decisiones de diseño.

1.3. Dificultades en la Creación de Mazmorras

Las dificultades que enfrentan los desarrolladores y jugadores en la creación de mazmorras incluyen:

- **Tiempo y Recursos**: El diseño manual de mazmorras consume mucho tiempo y recursos, lo que puede limitar la cantidad y la calidad de las mazmorras disponibles en un juego. Los diseñadores deben equilibrar la creación de contenido con las restricciones del proyecto y ajustar sus ideales con los objetivos de diseño del juego.
- **Coherencia y Jugabilidad**: Las mazmorras generadas automáticamente a menudo carecen de coherencia temática y narrativa, y no siempre ofrecen un desafío adecuado. Esto puede resultar en una experiencia de juego menos satisfactoria para los jugadores. Por lo tanto, el uso de herramientas que apoyen la generación de niveles en base a los requerimientos del diseñador puede ayudar a generar espacios que mantengan esta coherencia.
- **Balance de Dificultad**: Encontrar el equilibrio adecuado entre la dificultad y la accesibilidad es un desafío constante. Las mazmorras deben ser desafiantes pero no frustrantes, lo que requiere ajustes constantes y pruebas, por lo que la capacidad de testeo automatico en la generación o por sistemas de apoyo ayudaría bastante la creación del balance y dificultad dentro de los niveles.
- **Rejugabilidad**: Aunque no todos los juegos buscan pertenecer al género roguelike o roguelite, para aquellos que deseen incluir estos conceptos, mantener la rejugabilidad mientras se asegura que las mazmorras sean frescas e interesantes en cada partida

es una tarea complicada. Las variaciones procedurales deben ser lo suficientemente significativas para evitar la monotonía.

1.4. Generación Procedural de Contenido (PCG) en Juegos

La Generación Procedural de Contenido (PCG, por sus siglas en inglés) se refiere al uso de algoritmos para crear datos de manera automática, en lugar de depender exclusivamente del diseño manual. En el ámbito de los videojuegos, la PCG se ha convertido en una herramienta vital para desarrollar mundos, niveles, personajes y otros elementos del juego de manera eficiente y variada. Esta técnica ha sido adoptada por numerosos desarrolladores de videojuegos para mejorar la rejugabilidad, reducir los costos de producción y ofrecer experiencias únicas a los jugadores.

La PCG no solo se utiliza durante el juego para ofrecer variabilidad en tiempo real, sino también offline, lo que facilita la creación y optimización de contenido durante el desarrollo del juego [Togelius *et al.*, 2011]. Este enfoque offline permite a los desarrolladores iterar rápidamente sobre grandes volúmenes de contenido, probar diferentes configuraciones y mejorar la calidad del juego antes de su lanzamiento. Un estudio detallado de las diferencias entre generación online y offline puede encontrarse en el trabajo de [Smith y Mateas, 2011], que explora cómo la generación offline puede influir significativamente en la fase de diseño y prueba de un videojuego.

1.5. Aplicaciones de la PCG en Videojuegos

La PCG se utiliza para crear diversos elementos del juego, desde niveles y mazmorras hasta personajes, enemigos y narrativas, proporcionando variabilidad y personalización en cada partida. Algunos de estos aspectos del diseño de videojuegos son:

- **Niveles y Mazmorras**: La generación procedural de niveles y mazmorras es una de las aplicaciones más comunes de la PCG. Juegos como *Diablo* (1996) y *Spelunky* (2008) utilizan algoritmos para crear mazmorras y niveles únicos cada vez que se juega, asegurando que los jugadores siempre tengan una experiencia fresca y desafiante. La generación de mazmorras en *The Legend of Zelda* ilustra cómo estos algoritmos no solo crean variabilidad, sino que también mantienen la coherencia y lógica interna del diseño de niveles, garantizando que sean jugables [Lavender y Thompson, 2017].
- **Terrenos y Mundos**: Los juegos de mundo abierto, como *Minecraft* (2009) y *No Man's Sky* (2016), utilizan la PCG para generar vastos mundos explorables. Esto permite a los jugadores explorar paisajes interminables y descubrir nuevos entornos y recursos, lo que incrementa significativamente la escala y la profundidad del juego. [Summerville y Mateas, 2015]

discuten los métodos de muestreo y generación de terrenos, destacando la importancia de la cohesión visual y la jugabilidad en mundos generados proceduralmente.

- **Personajes y Enemigos**: La PCG también se usa para crear personajes y enemigos con atributos, habilidades y comportamientos únicos. Esto se observa en juegos como *Borderlands* (2009), donde los enemigos y las armas se generan proceduralmente, ofreciendo una variedad interminable de combinaciones y desafíos. [Shaker *et al.*, 2016] examinan cómo la generación de personajes y enemigos puede mejorar la diversidad del juego y mantener el interés del jugador mediante la variabilidad en los encuentros y el equipamiento.

- **Misiones y Narrativas**: Algunos juegos utilizan la PCG para generar misiones y narrativas dinámicas. Juegos como *Dwarf Fortress* (2006) y *The Elder Scrolls V: Skyrim* (2011) pueden crear misiones secundarias y eventos aleatorios que enriquecen la historia y el mundo del juego, proporcionando un contenido más diverso y atractivo. Dormans [Dormans, 2010] explora cómo la generación procedural de misiones y narrativas puede añadir profundidad y longevidad al contenido de un juego, manteniendo a los jugadores inmersos con nuevas historias y desafíos en cada sesión de juego.

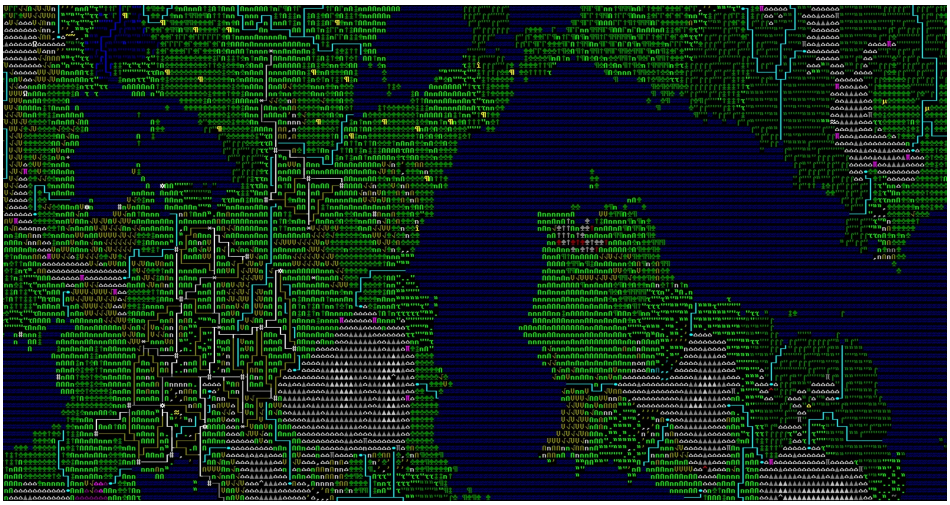


Figura 4: Generación de mundo en *Dwarf Fortress*.

1.5.1. Juegos de Mazmorras

Para niveles más complejos y dinámicos, como los de *The Binding of Isaac* (Figura 5), los desarrolladores adoptan un enfoque mixto que combina la creación manual de habitaciones y la generación procedural de la disposición y la conexión de estas habitaciones. En este enfoque híbrido, los diseñadores crean una serie de habitaciones o módulos que representan escenarios temáticos y desafíos de juego únicos. Esta técnica también se utiliza para la generación procedural, utilizando la variabilidad de los niveles generados manualmente para

luego generar un mapa único seleccionando las posibles habitaciones dentro de un grupo factible para determinar el mapa actual.

Estas habitaciones se diseñan manualmente para garantizar una jugabilidad equilibrada y una coherencia visual dentro del juego. Cada habitación puede contener enemigos, trampas, recompensas y otros elementos interactivos que contribuyen a la experiencia de juego general.

Una vez que se han creado las habitaciones individuales, se utilizan algoritmos de generación procedural para conectar estas habitaciones y crear un laberinto dinámico que se adapte a cada partida. Estos algoritmos pueden ajustar la disposición, el tamaño y la dificultad de las habitaciones en función de varios factores, como el progreso del jugador, la dificultad del juego y la estructura narrativa del juego.

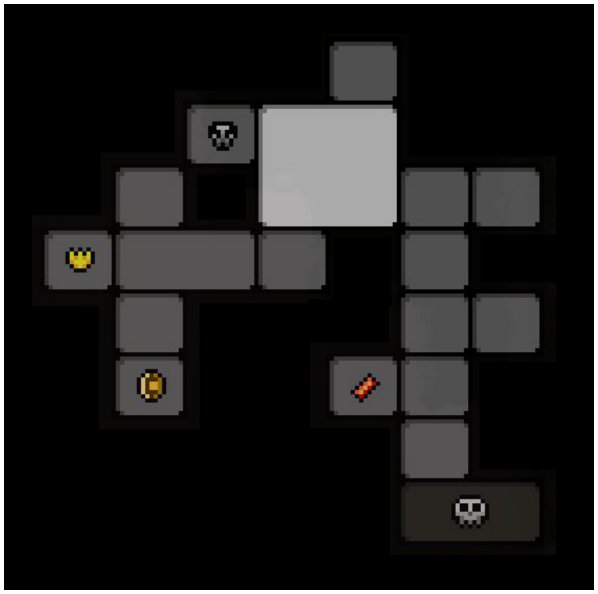


Figura 5: Nivel del juego *The Binding of Isaac*, mostrando la combinación de diseño manual y generación procedural.

La creación de niveles en mazmorras, especialmente en juegos de rol y roguelikes, puede ser manual o procedural. En el diseño manual, los niveles son creados con gran detalle, asegurando una narrativa y un flujo de juego específicos. Por otro lado, la generación procedural de mazmorras utiliza algoritmos para crear niveles únicos en cada partida, lo que aumenta la rejugabilidad y el desafío del juego.

En juegos como *Diablo 6*, la generación procedural se emplea para crear laberintos complejos y variados cada vez que el jugador ingresa a una nueva mazmorra. Estos algoritmos aseguran que cada mazmorra mantenga una lógica interna y un balance adecuado de dificultad, proporcionando una experiencia fresca y emocionante en cada incursión [Amato, 2017].

Figura 6: Ejemplo de una mazmorra generada proceduralmente en *Diablo 3*.

Los juegos de rol y roguelikes también hacen un uso extensivo de la generación procedural de mazmorras. En juegos como *The Binding of Isaac* y *Spelunky*, los niveles subterráneos se generan aleatoriamente, proporcionando una experiencia de juego única en cada partida. La generación procedural permite que estos juegos mantienen su frescura y desafío incluso después de numerosas repeticiones, ya que el diseño de la mazmorra cambia constantemente [Shaker *et al.*, 2016].

En resumen, la creación de mazmorras en videojuegos puede abordarse tanto desde un enfoque manual, que permite un control total sobre el diseño y la narrativa, como desde un enfoque procedural, que ofrece variabilidad y rejugabilidad. La combinación de ambas técnicas en algunos juegos permite aprovechar lo mejor de ambos mundos, proporcionando experiencias de juego equilibradas y emocionantes tanto para desarrolladores como para jugadores.

1.5.2. Juegos de Mundo Abierto

Juegos de mundo abierto como *Minecraft* 7 y *No Man's Sky* utilizan la PCG para generar vastos mundos explorables. En *Minecraft*, los algoritmos generan terrenos, cuevas y biomas, permitiendo una experiencia de juego casi ilimitada donde cada partida es única. *No Man's Sky* lleva este concepto aún más lejos, generando proceduralmente un universo entero con planetas únicos, flora y fauna distintas, ofreciendo una escala sin precedentes y demostrando cómo la PCG puede transformar la escala y la ambición de los videojuegos modernos.

Figura 7: Ejemplo de un mundo generado en *Minecraft*.

1.6. Historia de la PCG

El uso de la PCG en los videojuegos se remonta a la década de 1980 con el lanzamiento de *Rogue* (1980), un juego que introdujo la generación aleatoria de niveles de mazmorras. Este enfoque permitía a los jugadores enfrentarse a desafíos siempre diferentes en cada partida, lo que aumentaba la rejugabilidad y mantenía el interés del jugador a lo largo del tiempo. A partir de entonces, la PCG se ha expandido a múltiples géneros de videojuegos, desde los juegos de rol hasta los simuladores espaciales.

En los años 90, juegos como *Diablo* (1996) y *Daggerfall* (1996) popularizaron aún más el uso de la PCG, demostrando su potencial para crear experiencias de juego vastas y complejas. Con la llegada del nuevo milenio, títulos como *Minecraft* (2009) y *No Man's Sky* (2016) llevaron la PCG a nuevas alturas, generando mundos enteros que los jugadores pueden explorar y modificar casi infinitamente.

Estos juegos mostraron la evolución de la PCG desde la simple generación de niveles o mazmorras hasta la creación de vastos y complejos mundos abiertos. Por ejemplo, *Minecraft* utiliza algoritmos de PCG para crear terrenos, cuevas y biomas, lo que permite una experiencia de juego casi ilimitada donde cada partida es única. *No Man's Sky* llevó este concepto aún más lejos al generar proceduralmente un universo entero con planetas únicos, flora y fauna distintas, ofreciendo una escala sin precedentes y demostrando cómo la PCG puede transformar la escala y la ambición de los videojuegos modernos.

1.7. Beneficios de la PCG

La Procedural Content Generation (PCG) ofrece una serie de beneficios significativos en el desarrollo de videojuegos:

- **Rejugabilidad**: La capacidad de generar contenido nuevo en cada partida mantiene el interés del jugador a lo largo del tiempo. Según Liapis et al. [Liapis *et al.*, 2015], ofrecer experiencias frescas y únicas en cada sesión de juego es esencial para mantener la lealtad y el compromiso del jugador. Esto no solo aumenta la longevidad del juego, sino que también fomenta la exploración y la experimentación por parte del jugador, ya que cada partida presenta desafíos y escenarios diferentes.
- **Eficiencia en el Desarrollo**: La PCG reduce la necesidad de diseñar manualmente cada elemento del juego, lo que ahorra tiempo y recursos. [Smith y Mateas, 2011] se discute cómo la generación procedural puede acelerar significativamente el proceso de desarrollo al automatizar la creación de grandes volúmenes de contenido. Esto permite a los desarrolladores concentrarse en aspectos más complejos del juego, como la jugabilidad, la historia y la optimización, en lugar de pasar tiempo en la creación manual de contenido repetitivo.

Para enfrentar estas dificultades, los desarrolladores de videojuegos recurren cada vez más a herramientas de generación procedural (PCG, por sus siglas en inglés), que ofrecen varias ventajas en la creación de mazmorras y otros contenidos del juego. Algunas de las formas en que estas herramientas apoyan el desarrollo incluyen:

- **Optimización de Recursos**: Las herramientas de generación procedural permiten a los desarrolladores crear grandes cantidades de contenido con menos recursos, reduciendo la necesidad de diseñar manualmente cada nivel.
- **Prototipado Rápido**: PCG permite la creación rápida de prototipos de niveles, lo que facilita la iteración y el ajuste de diseños antes de la producción final.
- **Variedad y Rejugabilidad**: Los algoritmos procedurales pueden generar mazmorras únicas cada vez que se juega, lo que aumenta la rejugabilidad y mantiene el interés de los jugadores.
- **Personalización y Adaptación**: Las herramientas modernas permiten a los diseñadores ajustar parámetros y reglas para personalizar la generación de niveles, asegurando que se mantengan alineados con la visión del juego.
- **Soporte en el Diseño**: Al combinar elementos generados proceduralmente con diseño manual, los desarrolladores pueden asegurar tanto la calidad como la variedad. Herramientas como Unity, Unreal Engine y motores específicos de PCG facilitan este proceso.

1.8. Desafíos Actuales y su Impacto

Aunque las herramientas de generación procedural ofrecen muchos beneficios, también presentan desafíos que los desarrolladores deben enfrentar:

- **Control y Predictibilidad**: Mantener el control sobre el diseño y asegurarse de que los niveles generados cumplan con los estándares de calidad puede ser difícil.
- **Combinación de Elementos**: Integrar elementos diseñados manualmente con los generados proceduralmente puede resultar en inconsistencias si no se hace correctamente.
- **Curva de Aprendizaje**: Las herramientas de generación procedural pueden tener una curva de aprendizaje pronunciada, requiriendo tiempo y esfuerzo para ser utilizadas de manera efectiva, sobre todo si estas no son claras y no son fáciles de integrar en los engines actuales utilizados.

Gran cantidad de desarrolladores buscan automatizar procesos como la generación de niveles para optimizar recursos y mejorar la calidad del contenido. Con el continuo avance de las tecnologías de PCG, se espera que estas herramientas jueguen un papel aún más importante en el desarrollo de videojuegos, permitiendo a los desarrolladores crear experiencias más ricas y variadas para los jugadores. Sin embargo, si no se solucionan estas dificultades, los desarrolladores continuarán enfrentando altos costos y tiempos de desarrollo prolongados, mientras que los jugadores podrían desmotivarse debido a la baja calidad de las mazmorras generadas. Esto podría impactar negativamente las ventas y la reputación de los videojuegos, afectando la sostenibilidad de los proyectos y la confianza del público en los desarrolladores. Además algunas de estas herramientas a medida que avanza el tiempo son implementadas directamente en los Game Engines.

1.9. Competencia Actual y Objetivos de la Solución

Existen varias soluciones para la generación procedural de mazmorras, como el uso de algoritmos básicos y herramientas comerciales. Sin embargo, estas soluciones a menudo no proporcionan la flexibilidad y la calidad necesarias para crear mazmorras complejas y bien balanceadas. Nuestro enfoque con el algoritmo evolutivo en dos pasos y la aplicación de simetría horizontal tiene el potencial de superar estas limitaciones, ofreciendo una solución más robusta y adaptable que puede generar mazmorras más coherentes y desafiantes.

El objetivo principal de esta tesis es desarrollar un algoritmo evolutivo de dos pasos para la generación de mazmorras con misiones de puertas bloqueadas, utilizando la simetría horizontal para mejorar la coherencia y jugabilidad de las mazmorras generadas. Los alcances incluyen:

- Generar disposiciones de mazmorras que cumplan con un conjunto específico de parámetros y restricciones que representan las decisiones del diseñador.
- Asegurar que las mazmorras generadas proporcionen un desafío adecuado y una experiencia de juego satisfactoria.
- Ofrecer una herramienta eficiente para los desarrolladores de videojuegos, facilitando la creación de contenido variado y de alta calidad.

1.10. Generación Procedural de Mazmorras (PDG)

La Generación Procedural de Mazmorras (PDG) es una aplicación específica de la Generación Procedural de Contenido (PCG) en los videojuegos que se centra en la creación automática de niveles de mazmorras. Las mazmorras son entornos subterráneos comúnmente encontrados en juegos de rol, aventuras y roguelikes, donde los jugadores exploran pasajes oscuros, enfrentan monstruos y resuelven acertijos para avanzar en la historia del juego.

La PDG utiliza algoritmos para generar mazmorras de manera automática, lo que proporciona una amplia variedad de niveles únicos en cada partida. Estos algoritmos pueden crear laberintos complejos, llenos de habitaciones interconectadas, pasadizos secretos, trampas y tesoros ocultos, todo diseñado para desafiar y entretener a los jugadores.

1.10.1. Métodos de Generación

Existen varios enfoques para la generación procedural de mazmorras, cada uno con sus propias técnicas y desafíos. Algunos de los métodos más comunes incluyen:

- **Cellular Automata**: Este método utiliza reglas simples para simular el crecimiento de estructuras laberínticas. Los algoritmos de Cellular Automata pueden generar mazmorras con pasajes retorcidos y áreas abiertas de manera eficiente. Esto permite la creación de entornos subterráneos con una sensación natural y orgánica, ideales para la exploración del jugador. Teniendo excelentes resultados en la velocidad de generación.
- **Recursive Division**: En este enfoque, se divide repetidamente una mazmorra en secciones más pequeñas, creando pasajes y cámaras de forma recursiva. Este método es eficaz para generar mazmorras con diseños orgánicos y múltiples niveles de profundidad. Al dividir la mazmorra en secciones más pequeñas, se pueden crear áreas detalladas y únicas, lo que añade variedad y complejidad a la experiencia del juego.
- **Algoritmos basados en Grafos**: Estos algoritmos representan la mazmorra como un grafo de nodos y aristas, donde los nodos son habitaciones y las aristas son pasajes. Los

algoritmos de búsqueda y recorrido de grafos se utilizan para conectar las habitaciones de manera coherente y crear una mazmorra jugable. Este enfoque permite una mayor flexibilidad en la generación de mazmorras, ya que se pueden ajustar parámetros como la densidad de las conexiones o la distribución de las habitaciones para adaptarse a diferentes estilos de juego.

1.10.2. Ejemplo en Juegos

La Generación Procedural de Mazmorras (PDG) se ha utilizado en una amplia gama de juegos para crear mazmorras emocionantes y desafiantes. Ejemplos destacados incluyen:

- **The Binding of Isaac**: Este roguelike utiliza la PDG para generar mazmorras únicas en cada partida. Las mazmorras están llenas de habitaciones generadas proceduralmente, enemigos, trampas y recompensas, lo que garantiza una experiencia de juego fresca y emocionante en cada sesión. La combinación de elementos generados al azar y una jugabilidad táctica intensa hace que cada incursión en las mazmorras sea una experiencia única y desafiante para los jugadores.
- **Darkest Dungeon**: En este juego de rol táctico, la PDG se utiliza para crear mazmorras oscuras y peligrosas que los jugadores deben explorar en busca de tesoros y gloria. Las mazmorras están llenas de trampas mortales, enemigos despiadados y secretos ocultos, lo que ofrece un desafío constante para los valientes aventureros. La PDG contribuye a la atmósfera opresiva y la sensación de peligro inminente que define la experiencia de juego en Darkest Dungeon.

La PDG ha demostrado ser una herramienta invaluable para los desarrolladores de videojuegos que desean crear experiencias de juego dinámicas y emocionantes. Al generar mazmorras de manera automática, los desarrolladores pueden ofrecer a los jugadores una variedad infinita de desafíos y aventuras, garantizando que cada partida sea única y memorable. Este enfoque también permite a los desarrolladores mantener el interés de la comunidad a lo largo del tiempo, ya que cada nueva partida presenta nuevas sorpresas y desafíos para descubrir.

1.11. Resumen

Las herramientas de generación procedural de contenido (PCG) son esenciales en el desarrollo moderno de videojuegos, particularmente en géneros como RPG, aventura y roguelike. Permiten crear entornos dinámicos y variados, mejorando la experiencia del jugador y optimizando los recursos de desarrollo.

En contraste con la creación manual de niveles, que es intensiva en tiempo y recursos pero necesaria para un control narrativo y artístico específico, las herramientas de PCG ofrecen una serie de ventajas clave:

- Eficiencia en el desarrollo mediante la automatización.
- Incremento de la variabilidad y rejugabilidad de los juegos.
- Escalabilidad en la creación de contenido.

La generación procedural de mazmorras (PDG) se enfoca en diseñar entornos desafiantes utilizando algoritmos evolutivos y técnicas específicas como la simetría horizontal. Las herramientas de PDG permiten:

- Prototipado rápido y eficiente.
- Mejora del flujo de trabajo con edición en tiempo real.
- Apoyo a la creatividad del diseñador, liberándose de tareas repetitivas.

Estas herramientas no solo reducen la carga de trabajo de los desarrolladores, sino que también mejoran la calidad del producto final, permitiendo una adaptación ágil a cambios y feedback. En conclusión, PCG y PDG son cruciales para un desarrollo de videojuegos más eficiente, creativo y de alta calidad.

En el siguiente capítulo, hablaremos de la formulación del problema, donde se define de manera detallada el problema a resolver. Se describirán las habitaciones inicial y final, las restricciones de configuración y conectividad, y las relaciones entre llaves y barreras. También se discutirá la diferenciabilidad de disposiciones y las características escogidas por el diseñador, así como su impacto en el problema. Finalmente, se abordará la abstracción del problema y la visualización de la disposición y el grafo.

CAPÍTULO 2
Formulación del Problema

La generación de mazmorras es el proceso de crear de manera procedural disposiciones para mazmorras en videojuegos. En una disposición de mazmorras en 2D, las habitaciones están interconectadas con habitaciones adyacentes, formando una red de pasillos y cámaras. El problema de la generación de mazmorras implica encontrar una función f que mapee un conjunto de parámetros de entrada P, incluyendo el número deseado de habitaciones R, el número de llaves K, el número de barreras B, y la semilla S, a una disposición de mazmorras generada $g \in G$, cumpliendo con las siguientes restricciones y requisitos:

2.1. Habitaciones inicial y final

La inclusión de una habitación inicial y una habitación final en la disposición de mazmorras generada es esencial para proporcionar un punto de inicio claro y una meta para los jugadores. Matemáticamente, este requisito se puede representar de la siguiente manera:

Sea g la disposición de mazmorras generada. Definimos $Inicial(g)$ como el conjunto de habitaciones iniciales en g y $Final(g)$ como el conjunto de habitaciones finales en g. El tamaño de $Inicial(g)$ y $Final(g)$ debe ser exactamente 1:

$$|Inicial(g)| = 1 \tag{1}$$

$$|Final(g)| = 1 \tag{2}$$

La ecuación 1 asegura que haya exactamente una habitación inicial en la disposición de mazmorras generada, mientras que la ecuación 2 garantiza la presencia de una única habitación final.

2.2. Restricciones de configuración

Las restricciones de tamaño aseguran que la disposición de mazmorras generada tenga el número deseado de habitaciones, llaves y barreras. Estas restricciones se pueden representar de la siguiente manera:

$$|Habitaciones(g)| = R \tag{3}$$

De manera similar, esta ecuación establece que el número de habitaciones en la disposición de mazmorras generada, representado como $Habitaciones(g)$, debe ser exactamente igual al número deseado de habitaciones, denotado como R.

$$|Llaves(g)| = K \tag{4}$$

De manera similar, esta ecuación establece que el número de llaves en la disposición de mazmorras generada, representado como $Llaves(g)$, debe ser exactamente igual al número deseado de llaves, denotado como K.

$$|Barreras(g)| = B \tag{5}$$

Esta ecuación especifica que el número de barreras en la disposición de mazmorras generada, representado como $Barreras(g)$, debe ser exactamente igual al número deseado de barreras, denotado como B.

2.3. Restricciones de conectividad

Las restricciones de conectividad aseguran que cada habitación en la mazmorra generada esté adecuadamente conectada a otras habitaciones adyacentes, creando una disposición completamente conectada en un entorno en 2D:

$$\forall r \in Habitaciones(g), 1 \leq |Conexiones(r)| \leq 4 \tag{6}$$

Esta ecuación establece que para cada habitación r en la disposición de mazmorras generada g, el número de conexiones desde r, denotado como $|Conexiones(r)|$, debe estar entre 1 y 4, inclusive. En un entorno 2D, estas conexiones deben establecerse con habitaciones

adyacentes, lo que significa que cada habitación puede tener conexiones con sus habitaciones vecinas en las direcciones cardinales (arriba, abajo, izquierda, derecha). Al adherirse a estas restricciones de conectividad, la disposición de mazmorras generadas mantiene una estructura coherente e interconectada.

2.4. Relaciones entre llaves y barreras

Las mazmorras generadas deben incorporar diferentes tipos de relaciones entre llaves y barreras, incluyendo relaciones de 1 a 1 (1 llave desbloquea 1 barrera). Estas relaciones deben distribuirse estratégicamente en toda la disposición de la mazmorra.

Además de estas relaciones, es importante asegurar que las llaves no se coloquen en las habitaciones inicial y final. Esta restricción evita el acceso inmediato a las llaves necesarias para desbloquear las barreras. Por lo tanto, se introduce la siguiente restricción:

$$\forall k \in Llaves(g), k \notin Inicial(g) \land k \notin Final(g) \tag{7}$$

Esta ecuación establece que para cada llave k en la disposición de mazmorras generada g, k no debe estar ubicada en la habitación inicial (punto de inicio) o la habitación final (meta).

Las relaciones entre llaves y barreras se pueden representar utilizando las siguientes ecuaciones:

$$\sum_{b \in Barreras(g)} x_{k,b} = 1 \quad \forall k \in Llaves(g) \tag{8}$$

(Cada llave desbloquea exactamente una barrera)

$$\sum_{k \in Llaves(g)} x_{k,b} = 1 \quad \forall b \in Barreras(g) \tag{9}$$

(Cada barrera es desbloqueada por exactamente una llave)

2.5. Diferenciabilidad de disposiciones

El requisito de diferenciabilidad de disposiciones establece que las disposiciones de mazmorras generadas deben exhibir configuraciones y estéticas distintas para proporcionar a los

jugadores una sensación de exploración y descubrimiento. Considerar una semilla S para el proceso de generación asegura la reproducibilidad de la misma disposición. Matemáticamente, la diferenciabilidad de disposiciones se puede representar de la siguiente manera:

$$D(g_1, g_2) \neq 0 \quad \forall g_1, g_2 \in G(S), g_1 \neq g_2 \tag{10}$$

donde $D(g_1, g_2)$ representa una métrica o función de distancia que mide la diferencia entre dos disposiciones de mazmorras g_1 y g_2, y $G(S)$ denota el conjunto de mazmorras generadas usando la semilla S. La ecuación 10 establece que para todos los pares de disposiciones de mazmorras distintas g_1 y g_2 en el conjunto $G(S)$, la diferencia entre ellas debe ser distinta de cero, indicando sus configuraciones distintas.

2.6. Características Escogidas por el Diseñador

Para abordar el problema de generación de mazmorras de manera efectiva, es esencial considerar las características específicas escogidas por el diseñador del juego. Estas características incluyen el número de habitaciones, llaves, barreras, el coeficiente lineal, y las llaves necesarias. Cada una de estas características afecta significativamente el problema de generación de habitaciones:

- **Número de Habitaciones (Rooms)**: El número total de habitaciones en la mazmorra, representado como R. Este parámetro define el tamaño y la complejidad de la mazmorra. Un mayor número de habitaciones aumenta la complejidad del problema, ya que hay más nodos a considerar en el grafo de la mazmorra.
- **Número de Llaves (Keys)**: La cantidad de llaves necesarias para desbloquear barreras dentro de la mazmorra, representada como K. Más llaves requieren una distribución cuidadosa para mantener el equilibrio del juego y asegurar que el jugador explore adecuadamente la mazmorra.
- **Número de Barreras (Barriers)**: La cantidad de barreras que deben ser desbloqueadas por llaves, representada como B. Más barreras implican más puntos de control dentro de la mazmorra, afectando la secuencia y el flujo del juego.
- **Coeficiente Lineal (Linear Coef)**: Un coeficiente que define la linealidad de la disposición de las habitaciones. Un coeficiente más alto sugiere que cada habitación tendrá más ramificaciones. Por ejemplo, una habitación con un coeficiente lineal de 3 intentará tener 3 habitaciones adyacentes sin considerar la habitación de origen. El coeficiente lineal se maneja entre 1.0 a 3.0, asegurando que al menos una habitación

esté conectada con otra, y como máximo, cada habitación se conectará con otras 4, incluyendo la habitación de origen.

- **Llaves Necesarias (Needed Keys)**: El número de llaves necesarias para que el jugador pueda avanzar a través de todas las barreras hasta llegar a la habitación final. Este parámetro asegura que el jugador debe encontrar un número mínimo de llaves para poder progresar en la mazmorra, afectando directamente el flujo y la dificultad del juego, este parámetro nos ayuda a determinar de manera más directa si las llaves y barreras puestas, son necesarias para finalizar un nivel.

La Tabla 1 muestra ejemplos de problemas de generación de mazmorras con los parámetros específicos. Estos problemas fueron seleccionados del paper de Leonard sobre algoritmos evolutivos para la generación de contenido (CEA), destacando diferentes combinaciones de tamaño y complejidad para evaluar la eficacia del algoritmo propuesto.

Habitaciones	Llaves	Barreras	Coeficiente Lineal	Llaves necesarias
15	3	2	2.0	2.0
20	4	4	1.0	4.0
20	4	4	2.0	4.0
25	8	8	1.0	8.0
30	4	4	2.0	4.0
30	6	6	1.5	6.0
100	20	20	1.5	20.0
500	100	100	1.5	100.0

Tabla 1: Pruebas de configuraciones de generación de mazmorras especificando los requisitos de habitaciones, llaves, barreras y llaves necesarias.

Estos problemas específicos representan una variedad de configuraciones que ponen a prueba diferentes aspectos del algoritmo de generación de mazmorras:

- **Problemas Pequeños (15-30 habitaciones)**: Estos problemas sirven para evaluar la eficiencia del algoritmo en escenarios de menor escala, donde la complejidad es manejable y los resultados se pueden verificar rápidamente. Son útiles para iterar rápidamente en el desarrollo del algoritmo y asegurarse de que funcione correctamente en escenarios básicos.
- **Problemas Medianos (100 habitaciones)**: Con un número mayor de habitaciones, llaves y barreras, estos problemas permiten evaluar cómo el algoritmo maneja una mayor complejidad. Esto incluye la capacidad de mantener la coherencia en el diseño y la jugabilidad, así como el rendimiento del algoritmo en términos de tiempo de ejecución y recursos necesarios.

- **Problemas Grandes (500 habitaciones)**: Estos representan desafíos significativos debido a la escala y la complejidad involucradas. Son cruciales para probar la escalabilidad del algoritmo y su capacidad para generar mazmorras grandes y complejas que sigan siendo jugables y equilibradas. Evaluar el rendimiento en estos escenarios asegura que el algoritmo puede manejar incluso los casos más extremos y exigentes. Estos problemas nos ayudan a comprender las limitaciones de los algoritmos al ser expuestos a una dificultad aún mayor.

Cada combinación de parámetros en la tabla permite una evaluación detallada de cómo el algoritmo responde a diferentes requisitos y restricciones. Al abarcar desde problemas pequeños hasta grandes, se obtiene una visión completa del desempeño y las capacidades del algoritmo en una amplia gama de situaciones, lo que es esencial para validar su robustez y aplicabilidad en la generación procedural de contenido para videojuegos.

Cantidad de Habitaciones en Juegos de Mazmorras

En los juegos de dungeon, la cantidad de habitaciones puede variar significativamente según el diseño del juego y la generación procedural de niveles. A continuación, se detalla la cantidad exacta de habitaciones en varios juegos populares de este género:

- **The Binding of Isaac**: En promedio, un piso normal en *The Binding of Isaac* puede tener entre 5 y 10 habitaciones, sin embargo, esta cifra puede fluctuar ampliamente debido a la generación procedural de niveles y la presencia de salas secretas.
- **Diablo** Los niveles en esta serie pueden contener desde unas pocas decenas hasta cientos de habitaciones, en promedio, se puede esperar un mínimo de 5 a 10 habitaciones por nivel, aunque esta cifra puede aumentar considerablemente en niveles más avanzados o en mazmorras especiales.
- **Enter the Gungeon**, los pisos suelen tener entre 5 y 10 habitaciones en promedio en niveles iniciales y entre 10 30 en niveles más avanzados, sin embargo, debido al ritmo más rápido del juego, tiende a tener habitaciones más pequeñas en comparación con otros juegos de mazmorras.
- **The Legend of Zelda** En los juegos de esta serie, donde las mazmorras son creadas manualmente, la cantidad de habitaciones varía ampliamente según la entrega específica. Por ejemplo, en *The Legend of Zelda: Ocarina of Time*, los templos pueden tener entre 10 y 20 habitaciones en promedio.

En resumen, la cantidad exacta de habitaciones puede variar según el juego específico, su diseño y su generación procedural de niveles. Estos números proporcionan una idea general de lo que se puede esperar en términos de exploración y jugabilidad en cada juego.

2.7. Impacto de las Características en el Problema

La influencia de estas características en el problema de generación de mazmorras se puede resumir de la siguiente manera:

- **Número de Habitaciones (Rooms)**: Un mayor número de habitaciones aumenta la complejidad del problema, ya que hay más nodos a considerar en el grafo de la mazmorra. Esto requiere algoritmos más sofisticados para asegurar que todas las habitaciones estén adecuadamente conectadas y que el diseño general sea coherente y jugable.
- **Número de Llaves (Keys)**: Más llaves requieren una distribución cuidadosa para mantener el equilibrio del juego y asegurar que el jugador explore adecuadamente la mazmorra. La colocación estratégica de llaves puede dirigir al jugador a través de rutas específicas, mejorando la experiencia de juego y aumentando el valor de rejugabilidad.
- **Número de Barreras (Barriers)**: Más barreras implican más puntos de control dentro de la mazmorra, afectando la secuencia y el flujo del juego. Las barreras bien ubicadas pueden crear desafíos interesantes y evitar que los jugadores avancen demasiado rápido, manteniendo el interés y la tensión del juego.
- **Coeficiente Lineal (Linear Coef)**: Un coeficiente lineal más alto sugiere una disposición de mazmorras más conexa, donde las habitaciones están conectadas de manera más directa. Un coeficiente más bajo sugiere una disposiciónmenos ramificada, lo que puede aumentar la exploración y la sorpresa al navegar por la mazmorra. Este parámetro ayuda a los diseñadores a ajustar la navegación y la experiencia del usuario dentro del nivel.
- **Llaves Necesarias (Needed Keys)**: Este parámetro asegura que el jugador debe encontrar un número mínimo de llaves para poder progresar en la mazmorra, afectando directamente el flujo y la dificultad del juego. Un balance adecuado en el número de llaves necesarias puede hacer que el juego sea desafiante pero justo, mejorando la satisfacción del jugador al completar la mazmorra.

Estas características seleccionadas y sus impactos destacan la importancia de un diseño cuidadoso en la generación de mazmorras, asegurando que el contenido generado no solo sea coherente y jugable, sino también atractivo y desafiante para los jugadores.

2.8. Abstracción del Problema

Para resolver el posicionamiento de las habitaciones y la mecánica de llaves y barreras 1-1, es fundamental abstraer el problema de tal manera que se puedan aplicar algoritmos y técnicas de generación procedural. Esta abstracción implica:

- **Posicionamiento de Habitaciones**: Determinar la ubicación de cada habitación en una cuadrícula 2D, asegurando que cada habitación esté conectada de manera adecuada a sus adyacentes, respetando el coeficiente lineal. Esta etapa implica el uso de algoritmos que aseguren que todas las habitaciones estén accesibles y que la mazmorra tenga una estructura lógica y navegable.
- **Distribución de Llaves y Barreras**: Colocar llaves y barreras en habitaciones específicas siguiendo las restricciones establecidas (e.g., llaves no en habitaciones inicial o final) y asegurar que cada llave desbloquee exactamente una barrera. Este proceso garantiza que el jugador tenga que explorar la mazmorra para encontrar las llaves necesarias y pueda progresar de manera controlada.

Este enfoque asegura que la disposición de la mazmorra no solo cumpla con los parámetros y restricciones, sino que también proporcione una experiencia de juego coherente y desafiante.

2.9. Visualización de la Disposición y el Grafo

A continuación se muestra una representación visual de una disposición de mazmorra y su correspondiente representación en forma de grafo. Estas representaciones ayudan a visualizar cómo se configuran las habitaciones y las conexiones entre ellas, facilitando el análisis y la validación del diseño generado.

Cada representación tiene un nodo inicial S y un nodo Final E.

2.10. Disposición de Mazmorra

La visualización de la disposición y del grafo permite identificar rápidamente posibles problemas en el diseño, como habitaciones no conectadas o ciclos no deseados, y ajustar los parámetros del algoritmo de generación para corregirlos. Además, estas representaciones facilitan la comunicación del diseño a otros miembros del equipo de desarrollo, asegurando que todos comprendan la estructura y la lógica de la mazmorra generada. La Figura 8 y la Figura 9 representan el mismo nivel, en formato de mapa y grafo respectivamente.

2.11. Resumen

En esta sección se aborda el problema de la generación de mazmorras considerando características clave definidas por el diseñador, como el número de habitaciones, llaves, barreras,

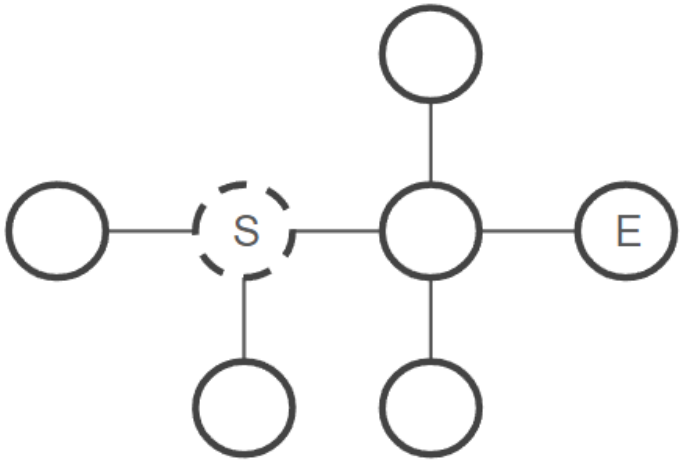


Figura 8: Ejemplo de una disposición de mazmorra en 2D.

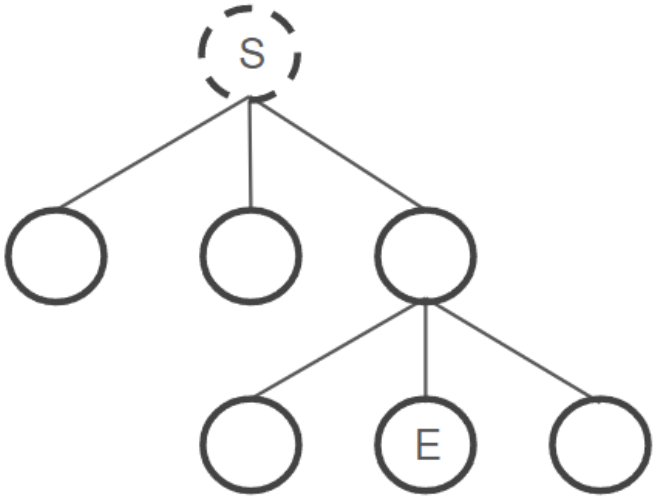


Figura 9: Representación en forma de grafo de la disposición de mazmorra. Los nodos representan habitaciones y las aristas representan las conexiones entre habitaciones.

el coeficiente lineal y las llaves necesarias. Se explica cómo estas características impactan el diseño y la jugabilidad de la mazmorra, y se presenta una tabla con ejemplos específicos tomados del paper de Leonard sobre algoritmos evolutivos para la generación de contenido.

La abstracción del problema se realiza mediante la determinación de la ubicación de las habitaciones en una cuadrícula 2D y la distribución de llaves y barreras según las restricciones del juego. Se asegura que la disposición de la mazmorra cumpla con los parámetros establecidos y proporcione una experiencia de juego coherente y desafiante.

Finalmente, se presentan visualizaciones de una disposición de mazmorra y su representación en forma de grafo. Estas visualizaciones facilitan la identificación de problemas en el diseño y la comunicación del mismo a los miembros del equipo de desarrollo.

En el próximo capítulo, revisaremos el estado del arte, explorando la literatura existente sobre generación procedural de contenido (PCG). Se examinarán diferentes técnicas generativas y evolutivas utilizadas en la creación de niveles de videojuegos, incluyendo las mecánicas de llave y barrera y la aplicación de simetría. Además, se discutirán las representaciones en la generación procedural de mazmorras y su relación con el trabajo de algoritmos evolutivos condicionados (CEA).

CAPÍTULO 3
Estado del Arte

La Generación Procedural de Mazmorras (PDG) es una subárea de la generación procedural de contenido (PCG) que se enfoca en la creación automática de niveles dentro de mazmorras. Estas estructuras juegan un rol crucial en muchos videojuegos, particularmente en géneros como los juegos de rol y los roguelikes-lite. En esta sección, se analizan las técnicas y enfoques más recientes aplicados en PDG, presentando distintas maneras en que se aborda la generación procedural de mazmorras y sus representaciones.

3.1. Generación Procedural de Mazmorras

La generación procedural de contenido (PCG, por sus siglas en inglés) es un campo extenso y multidimensional dentro del desarrollo de videojuegos. Se enfoca en la creación automatizada de contenido diverso y complejo, lo que habilita experiencias de juego únicas y personalizadas. A continuación, se presentan las principales áreas de trabajo en PCG, cada una con sus técnicas y aplicaciones específicas:

- **Generación de Niveles y Mazmorras**: La generación de niveles y mazmorras es una de las aplicaciones más conocidas de la PCG. Este proceso permite la creación de entornos de juego intrincados y desafiantes sin necesidad de diseñadores humanos para cada nivel individual, generando automáticamente niveles y habitaciones de mazmorras. La generación de niveles es un área aún más general, ya que incluye una variedad de géneros y representaciones, como juegos de plataformas (por ejemplo, *Super Mario* o *Spelunky*), donde las interacciones del jugador influyen en la forma en que se interpretan e interactúan con los niveles.
- **Generación de Terrenos y Mundos**: La generación de terrenos y mundos abiertos ha evolucionado significativamente, permitiendo la creación de vastos paisajes explorables en juegos como *Minecraft* y *No Man's Sky*. En la actualidad, muchos motores de videojuegos, como Unreal Engine, incorporan herramientas avanzadas para la generación procedural de terrenos, lo que facilita su integración en el proceso de desarrollo y permite la creación automática de entornos naturales y urbanos.
- **Generación de Personajes y Enemigos**: La generación de personajes, tanto aliados como enemigos, es crucial para añadir profundidad y dinamismo a la experiencia de juego. Este proceso involucra la creación de personajes con habilidades, características y comportamientos específicos que enriquecen la narrativa y los desafíos del juego. Los enemigos pueden variar en dificultad y estilo de combate, manteniendo el interés del

jugador a través de una variedad constante de desafíos. Asimismo, la generación de personajes aliados con distintas habilidades y roles fomenta la estrategia y la colaboración en juegos multijugador o de equipo.

- **Generación de Misiones y Narrativas**: La generación de misiones tiene como objetivo proporcionar al jugador una serie de tareas y objetivos que mantengan su interés y ofrezcan desafíos progresivos. Las misiones pueden variar en complejidad, desde simples tareas de recolección hasta secuencias elaboradas que requieren resolución de acertijos o combate estratégico. En cuanto a la narrativa, la PCG busca crear historias envolventes que sumergen al jugador en un universo coherente. Una narrativa bien desarrollada da contexto a las misiones, motivando al jugador a avanzar y explorar más a fondo el mundo del juego, y ofreciendo una experiencia emocionalmente rica y memorable.

3.2. Generación procedural de niveles y Mazmorras

En el contexto de la presente tesis, las técnicas existentes se pueden dividir en generativas y evolutivas.

3.2.1. Técnicas Generativas

Los algoritmos basados en gramáticas han sido ampliamente utilizados en la generación de contenido procedural (PDG). Dormans [Dormans, 2010] propuso dividir el espacio del nivel en dos partes: la representación de misiones y el espacio que éstas constituyen (ver Fig. 10). Utilizando gramáticas generativas, cada secuencia de interacción en el nivel puede ser considerada como parte del espacio de misiones. Por ejemplo, en un calabozo, una misión puede consistir en un obstáculo seguido de un tesoro. Dormans define cada elemento y luego forma correctamente la gramática de misiones. Por otro lado, para el espacio del nivel, se genera el nivel basándose en reglas de forma que incluyen la generación de las misiones (ver Fig. 11). Dormans utiliza una representación de grafos tanto para las misiones como para el nivel, empleando operadores generativos de gramática. El enfoque de gramática está más asociado a lo que se refiere a la narrativa de los niveles, que es un concepto que a veces se separa de la generación. Las dificultades de esta técnica son que requiere que la gramática constructiva tenga o defina de manera correcta el cómo se generan los niveles.

Basándose en el trabajo de Dormans, Lavender y Thompson (2017), adaptaron un generador para niveles de tipo mazmorras. Utilizando un conjunto predefinido de reglas gramaticales, agregaron nuevas reglas para la generación y dividieron la generación de mazmorras en dos fases, dando contexto para las acciones del jugador y trabajando en la generación del mapa y misiones (narrativa del nivel). En la primera fase, generaron el “grafo de misión” que estructura las tareas que el jugador debe completar. En la segunda fase, usaron un enfoque

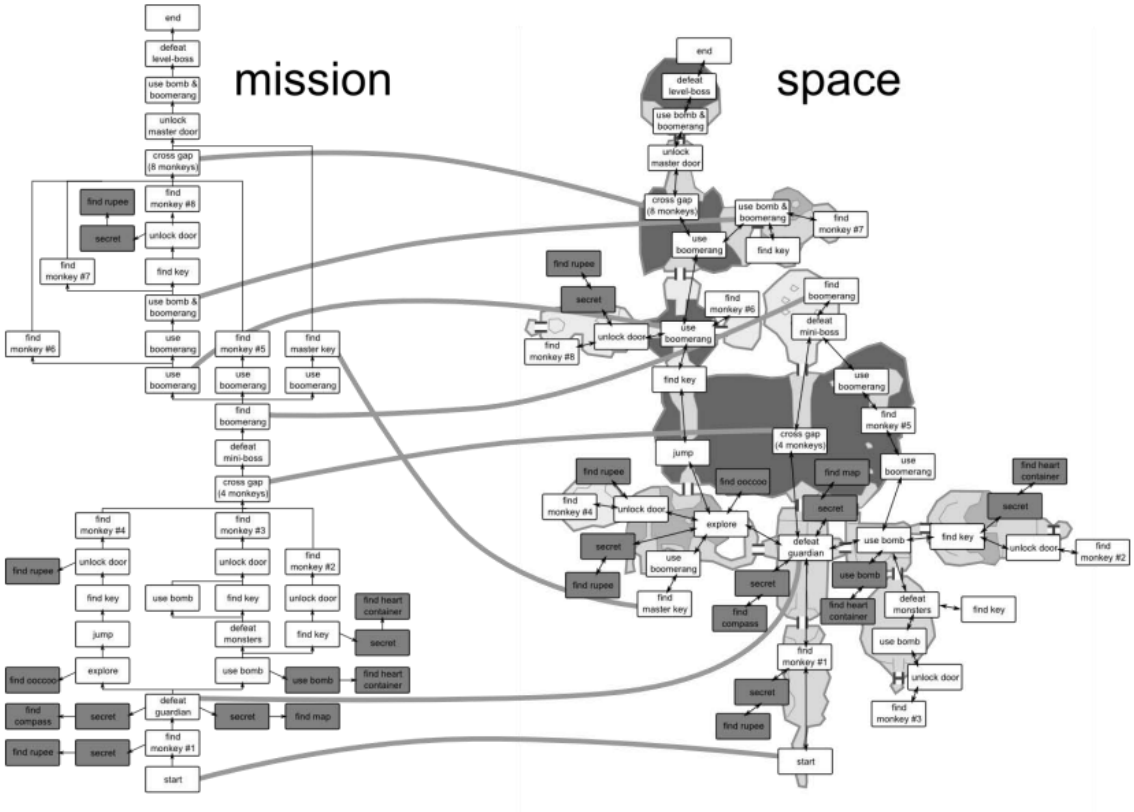


Figura 10: Separación de nivel en dos elementos, mission y espacio por Dormants

basado en restricciones para construir la geometría que representa fielmente la estructura de la misión original [Lavender y Thompson, 2017], aplicando estas técnicas en el juego de código abierto The Legend of Zelda: The Mystery of Solarus.

Otra técnica generativa se presenta en [Green *et al.*, 2019] que utiliza un algoritmo de generación de mazmorras en dos pasos para generar niveles para el juego MiniDungeons2, con tres enfoques para crear layouts y colocar elementos del juego: basado en restricciones, autómatas celulares y agentes. Los operadores incluyen reglas de colocación específicas, cambios de estado basados en vecinos y para luego probar el enfoque con movimiento de agentes según heurísticas. Este enfoque incluye diversidad en diseños, control sobre la complejidad e interactividad. La complejidad computacional, el ajuste de parámetros y la posible sobrepoblación de elementos son algunas de las limitaciones. Además, se presentan desafíos como la dependencia del diseño inicial, la interacción limitada entre elementosy la variabilidad en la jugabilidad. Este algoritmo genera mazmorras con entradas, salidas, tesoros, pociones, portales, trampas y monstruos, afectando tanto la estructura física como la experiencia de juego.

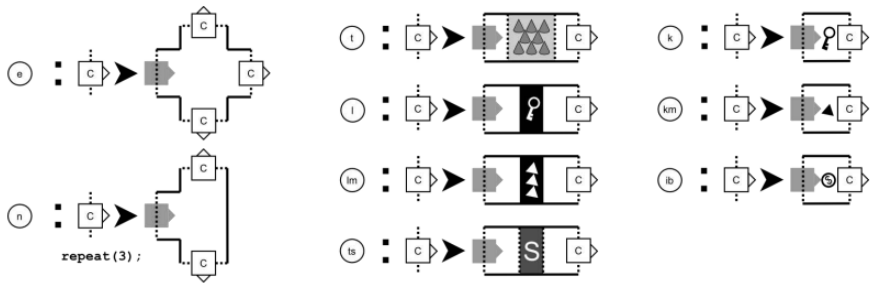


Figura 11: Reglas de gramática de forma para generar las misiones

Otra técnica que utiliza gramática se presenta en el trabajo de [Gellel y Sweetser, 2020], los autores emplean un enfoque híbrido que se inicia con la descripción de la gramática libre de contexto del nivel. Este primer paso se enfoca en definir las reglas y el esquema básico que determinarán la disposición y la lógica general de la mazmorra. Posteriormente, utilizan un proceso inspirado en autómatas celulares para generar el espacio físico detallado (ver Fig. 12), llenando el esquema inicial con contenido específico, como habitaciones, pasillos y obstáculos. Este enfoque se destaca por su coherencia estructural y detalles, mientras que las desventajas incluyen la complejidad de la implementación y la necesidad de ajustes manuales. El algoritmo se centra en la dependencia de reglas gramaticales bien definidas y la dificultad para generar niveles completamente originales sin patrones preexistentes. El algoritmo genera mazmorras de juego complejas y variadas que combinan una estructura lógica clara con un alto nivel de detalle visual y funcional.

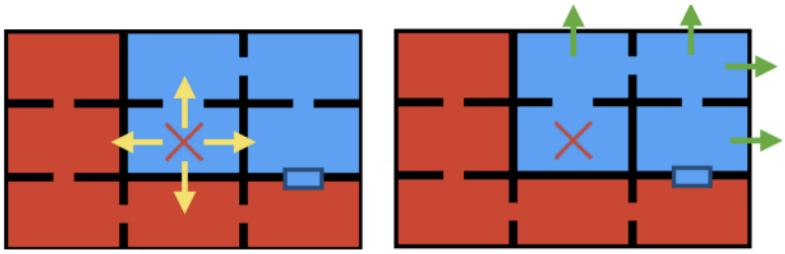


Figura 12: Reglas de CA: A la izquierda, sin posibilidad de agregar habitaciones; a la derecha, la habitación activa (X) con las direcciones de búsqueda.

3.2.2. Técnicas Evolutivas

El uso de algoritmos de búsqueda es otra técnica prominente en PDG. Togelius y Shaker (2011) exploraron el uso de estos algoritmos para la generación de contenido, donde se busca en un espacio de posibles diseños de mazmorras para encontrar aquellos que cumplen con criterios específicos de diseño [Togelius *et al.*, 2011]. Este enfoque permite una exploración exhaustiva de posibles configuraciones de mazmorras, optimizando ciertos parámetros de diseño.

Entre las técnicas evolutivas tenemos algunos con enfoques en la creación de agentes para niveles de tipo mazmorra, como el que presenta [Cerny y Dechterenko, 2015] donde aplicaron un algoritmo evolutivo codicioso para la creación de agentes en juegos tipo Roguelike, optimizando sus decisiones para una exploración efectiva y un combate estratégico. Utilizan una representación basada en una lista de acciones priorizadas y emplean operadores genéticos como cruzamiento de un punto y dos puntos, mutaciones de cambio pequeño y cruzamiento de promedio ponderado. Las ventajas incluyen la adaptabilidad de los agentes y la eficiencia en el uso de recursos, mientras que las desventajas abarcan la variabilidad en la evaluación del rendimiento y la complejidad de la implementación. El algoritmo genera agentes capaces de explorar mazmorras y combatir estratégicamente, mejorando su rendimiento hasta ser capaces de derrotar al jefe final con una tasa de éxito entre el 5-10 %.

Las técnicas que generan niveles, utilizan distintos enfoques para la creación de niveles por lo que es interesante analizar qué tipos de representaciones tienen y que algoritmos utilizan, para comprender de mejor manera el cómo se generan los niveles de mazmorras. Entre ellos se tiene a [Ashlock, 2015] que emplearon algoritmos evolutivos para definir reglas de autómatas celulares que crean estructuras similares a cuevas (ver Fig. 13). Utilizaron una matriz de 6x6 para representar las reglas, codificando como un vector de 36 números reales. Los operadores evolutivos incluyeron cruces de dos puntos y mutación puntual. En el enfoque se destacan la versatilidad y robustez en la generación de sistemas complejos y escalables. Sin embargo, presenta desventajas como la complejidad en la optimización.

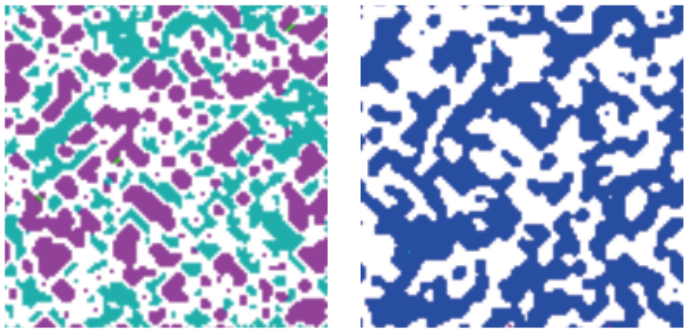


Figura 13: Caverna generada por Ashlock et al.

En [Baghdadi *et al.*, 2015] se combinaron algoritmos genéticos con agentes excavadores para diseñar niveles similares a Spelunky. La representación de los niveles se hace mediante un vector de enteros en una matriz de 4x4, con un gráfico donde cada nodo es una habitación. Los operadores incluyen mutación, cruzamiento y selección, optimizando la jugabilidad y calidad. Este enfoque genera niveles de manera automática, adaptables y variables, aunque también presenta desventajas como la complejidad computacional. Tiene una variación limitada en configuraciones similares y no considera las preferencias del jugador. El algoritmo genera niveles completos y jugables, con caminos, enemigos y objetos distribuidos según los

requisitos de diseño (ver Fig.14).

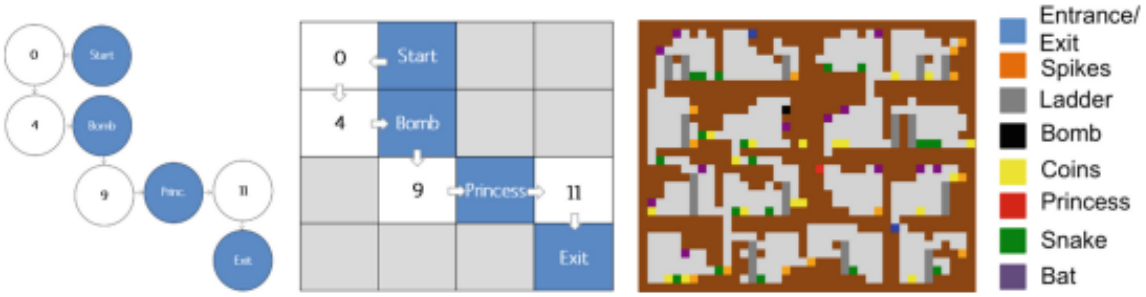


Figura 14: Pasos para la generación de niveles de Spelunky: evolución de la gráfica, habitaciones principales y habitaciones de libre selección para llegar al mapa final.

El uso de una técnica evolutiva de dos poblaciones: una factible y otra infactible (Fi-2pop) es común para la generación de niveles, como presenta Liapis y Yannakakis (2014). Ellos desarrollaron un algoritmo genético de dos poblaciones para generar niveles de *MiniDungeons*, representados como un arreglo de enteros que describen el contenido de cada casilla del nivel [Liapis *et al.*, 2015]. Este algoritmo hace la distinción entre individuos factibles e infactibles, optimizando la calidad del nivel y minimizando la distancia a la factibilidad, respectivamente. Las mutaciones permiten intercambiar características de las casillas, asegurando que se mantenga la cantidad de elementos esenciales. Los niveles generados son evaluados mediante *personas procedurales* que simulan estilos de juego específicos para garantizar jugabilidad y calidad (ver Fig. 15). Las ventajas incluyen una rápida generación de niveles jugables, mientras que las desventajas abarcan la limitada variabilidad en los niveles. Las limitaciones del algoritmo incluyen la influencia de las características de las personas en el diseño del nivel y posibles sesgos en la evaluación.

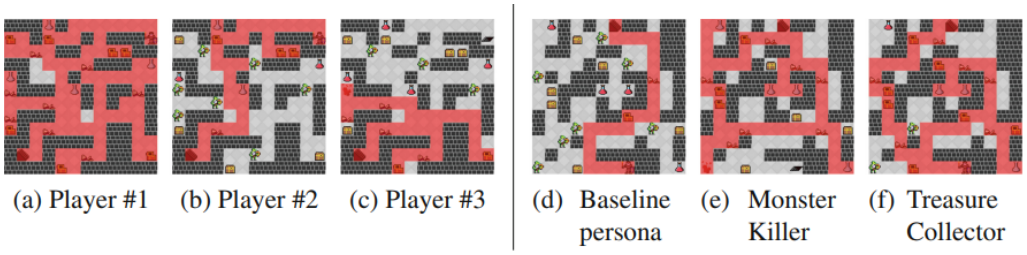


Figura 15: Actuar de las personas generadas en comparación con los agentes

El algoritmo utilizado en [Baldwin *et al.*, 2017] emplea una representación de salas de mazmorras como una matriz bidimensional de m × n, donde cada celda puede ser de seis tipos: piso, pared, enemigo, tesoro, entrada y puerta. Los operadores del algoritmo genético incluyen selección, cruce de dos puntos y mutación con una alta tasa del 90 %, donde las mutaciones pueden rotar la sala 180 grados o cambiar un solo tipo de celda. Entre las ventajas

se destacan el aumento de la diversidad de la población factible y la inclusión de patrones de diseño útiles de la población no factible. Las desventajas incluyen la posible generación de salas que no cumplen con las restricciones de jugabilidad y la terminación del algoritmo tras un número fijo de generaciones, lo que puede afectar la consistencia de la respuesta en distintos conjuntos de parámetros. Las limitaciones incluyen la dependencia de parámetros definidos por el usuario y la posible dominación de ciertos patrones (como los corredores) sobre otros. Finalmente, el algoritmo genera configuraciones de salas optimizadas para patrones específicos y niveles de dificultad, respetando restricciones de jugabilidad.

Entre los algoritmos presentados hay algunos que realizan un enfoque de múltiples pasos para generar más detalles en los niveles generados, como es en el caso del algoritmo presentado por Liapis et al. (2017) en el paper [Liapis, 2017] el cual utiliza una representación basada en un arreglo bidimensional de enteros para los bosquejos de mazmorras, que se optimiza mediante un Algoritmo Genético de Dos Poblaciones (Fi-2pop GA) evolucionando segmentos para obtener el mapa final(ver Fig. 16). Los operadores utilizados incluyen la mutación de un solo padre con una probabilidad y la recombinación mediante cruce de dos puntos. Las ventajas del algoritmo incluyen la capacidad de generar niveles jugables y la flexibilidad para incorporar diferentes técnicas generativas y la intervención humana en el diseño de niveles. Sin embargo, presenta desventajas como la complejidad de asegurar la jugabilidad de los niveles generados y la necesidad de balancear adecuadamente las poblaciones viables e inviables. Las limitaciones del algoritmo se centran en la restricción de los tipos de segmentos disponibles y los objetivos predefinidos, aunque se sugiere la extensión con diferentes tipos de monstruos y tesoros. Finalmente, el algoritmo genera niveles de mazmorras que son jugables y cumplen con ciertos criterios de diseño especificados previamente.

En el trabajo presentado por [Alvarez *et al.*, 2018] se exploraron el uso de criterios estéticos en el proyecto EDD, Evolutionary Dungeon Designer, herramienta presentada por Baldwin y Holmberg [Baldwin *et al.*, 2017], para la generación de habitaciones y niveles de mazmorras 2D. Utilizan una representación basada en patrones micro y meso, con un algoritmo evolutivo fi-2pop que incluye simetría y similitud en la evaluación de la aptitud. Los operadores genéticos empleados son cruza y mutación, mientras que los diseñadores pueden bloquear secciones para preservar estructuras estéticas personalizadas. Entre las ventajas se destacan la reducción del espacio de búsqueda y la preservación de los cambios del diseñador; sin embargo, las limitaciones incluyen la complejidad de equilibrar contenido humano y generado. El algoritmo genera habitaciones de mazmorras que respetan la estética del diseñador, proporcionando soluciones novedosas basadas en los patrones de diseño establecidos.

En [Melotti y de Moraes, 2019] utilizaron la técnica de Novelty Search with Local Competition (NSLC) para la generación de niveles de mazmorras, priorizando la diversidad y la originalidad de los diseños generados. La representación utilizada fue basada en una matriz donde cada celda representa un tipo de terreno. Los operadores empleados incluyeron la mutación

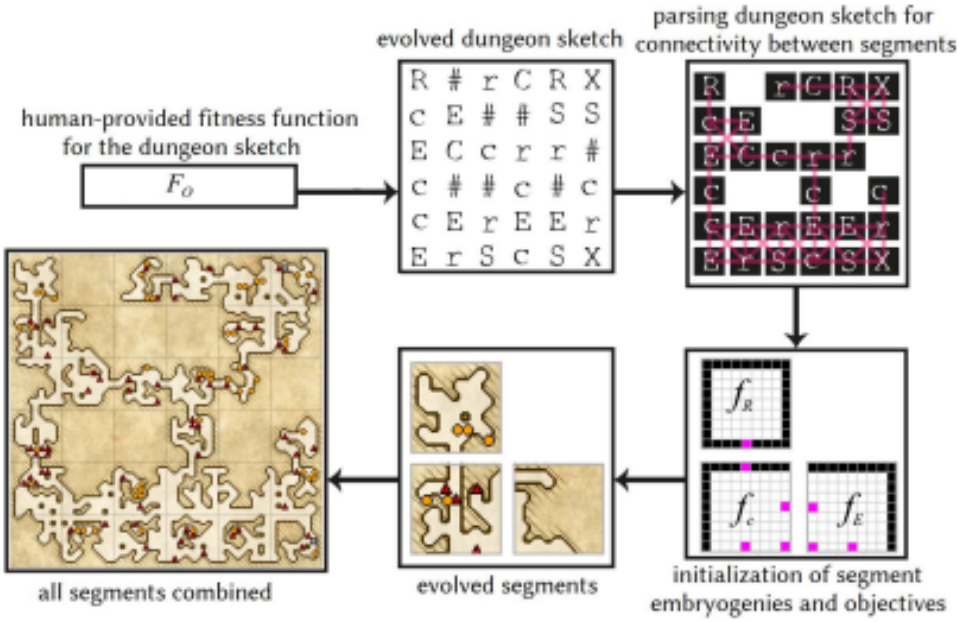


Figura 16: Pasos para la generación de niveles de mazmorras evolucionando segmentos desde la representación al mapa final

y el cruce. Entre las ventajas del algoritmo se destaca su capacidad para generar diseños diversos y originales, mientras que una de las desventajas es su complejidad computacional y la posible dificultad para ajustar parámetros. Finalmente, el algoritmo genera niveles de mazmorras con una alta diversidad morfológica, asegurando que los niveles sean tanto jugables como interesantes.

Se propone en [Pereira *et al.*, 2018] una nueva estructura de grafo para poder representar los niveles de mazmorras con misiones de puertas y barreras, donde cada nodo representa una habitación interconectada con el resto para crear un nivel representado (ver Fig. 17), idealmente siempre creando niveles factibles con la idea de verificar si siempre están los niveles bien conectados. Este método organiza las mazmorras en una jerarquía que incluye caminos bloqueados que los jugadores deben desbloquear con llaves. El algoritmo evoluciona esta representación mediante dos mutaciones y un cruce, seguido de una operación de reparación para corregir los niveles generados. Las ventajas del algoritmo incluyen la generación de mazmorras conectadas y semánticamente ricas, mientras que sus desventajas y limitaciones son la complejidad computacional y la posible necesidad de ajustes manuales. En última instancia, el algoritmo genera niveles de mazmorras con misiones viables que combinan el espacio de juego y las misiones de manera eficiente

Leonardo también propuso más adelante un Algoritmo Evolutivo Restringido [Pereira *et al.*, 2021],

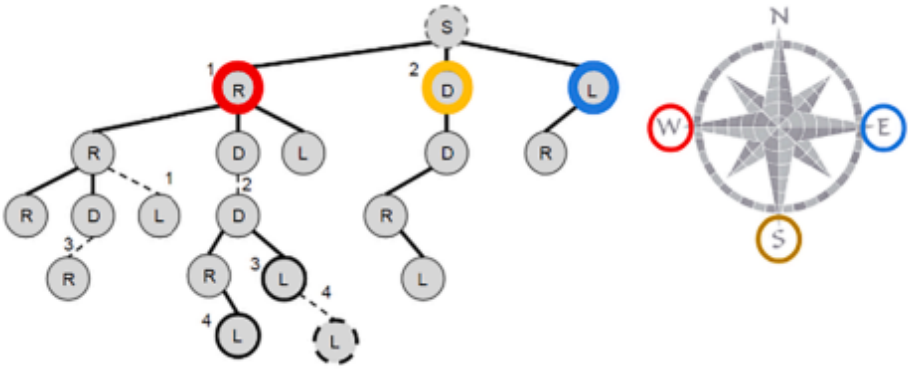


Figura 17: Representación de nivel en formato grafo

o por sus iniciales en inglés CEA (Constrained Evolutionary Algorithm), para la generación de mazmorras. Este algoritmo se enfoca en evolucionar niveles completos, aplicando restricciones específicas a las decisiones evolutivas de los mapas. El objetivo de estas restricciones es asegurar que las mazmorras generadas mantengan coherencia y jugabilidad, adhiriéndose a las reglas de diseño predeterminadas que guían la evolución hacia soluciones viables y entretenidas para los jugadores.

En el trabajo de [Ruela y Delgado, 2018] se presenta una representación similar, pero sin consideración de contexto para el nodo inicial, donde se aplicaron algoritmos evolutivos en representaciones basadas en redes para la creación de niveles 3D (ver Fig. 18). Los grafos permiten una representación abstracta y flexible de la estructura del nivel, donde los nodos representan áreas o habitaciones y los bordes representan conexiones o pasillos. Al aplicar algoritmos evolutivos a estos grafos, se pueden generar niveles tridimensionales complejos que mantienen una lógica espacial coherente y ofrecen una experiencia de juego inmersiva. La representación utilizada se basa en el modelo de Barabási-Albert, que describe redes de tipo ”scale-free”donde unos pocos nodos (hubs) tienen un alto grado de conexiones, mientras que la mayoría tienen pocas conexiones. Los operadores evolucionan la posición de los nodos y la probabilidad de creación de enlaces, determinada por una función probabilística decreciente. Las ventajas del algoritmo incluyen la flexibilidad para generar niveles variados y la capacidad de mantener una lógica espacial coherente. Sin embargo, las desventajas y limitaciones incluyen la posible complejidad computacional y la necesidad de ajuste fino de los parámetros del algoritmo. Finalmente, este enfoque genera niveles de juegos tridimensionales complejos y coherentes para juegos de aventuras tipo mazmorras.

El método Map of Elites se combinó con FI2Pop para la co-creación de niveles. Map of Elites es una técnica que mapea soluciones en un espacio de características, permitiendo identificar y preservar una variedad de soluciones óptimas. La representación utilizada en el algoritmo es la de habitaciones de mazmorras, donde se consideran diferentes patrones espaciales y meso-patrones. Los operadores del algoritmo incluyen la edición manual directa y las suge-

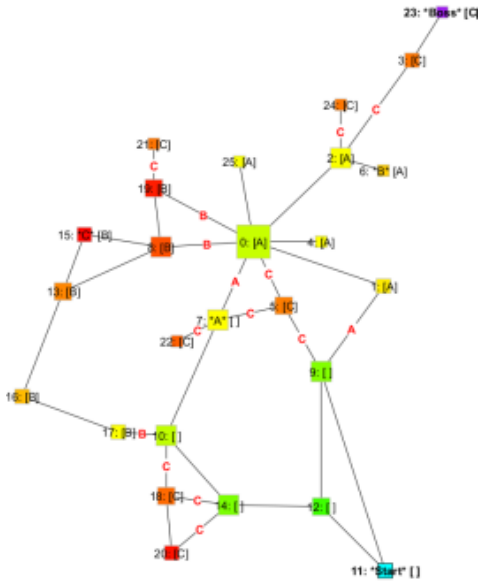


Figura 18: Representación de nivel conectado como red

rencias generadas usando procedimientos, que permiten la evolución continua y en tiempo real de los diseños (ver Fig. 19). Las ventajas del algoritmo son la capacidad de explorar múltiples dimensiones del diseño, como la simetría y la distribución de patrones, lo que resulta en una diversidad y calidad mejoradas de los niveles generados. Las desventajas y limitaciones incluyen la dificultad en generar habitaciones con ciertos valores de dimensión y la posible incompatibilidad entre patrones específicos. Finalmente, el algoritmo genera habitaciones de mazmorras diversas y equilibradas, facilitando una colaboración fluida entre el diseñador humano y la herramienta procedimental [Alvarez *et al.*, 2019].

En el trabajo de Zafar et al. [Zafar *et al.*, 2020], se emplea una representación de los niveles como una matriz 2D de baldosas. Los operadores del algoritmo, que incluyen crossover de un punto y mutación (crear, destruir e intercambiar), permiten la generación de niveles balanceados, simétricos, densos y alcanzables. Sin embargo, presentan ciertas desventajas, como la subjetividad en la evaluación de la estética y la dificultad del nivel. En última instancia, este algoritmo produce niveles de videojuegos con vista top-down que aseguran una experiencia entretenida y desafiante, evaluando tanto su estética como su dificultad.

En el estudio presentado en el paper [Viana *et al.*, 2022b], también se utiliza una estructura de grafo para representar los niveles de mazmorras. Los nodos del grafo representan habitaciones que pueden ser normales, con llaves o con puertas cerradas. Los operadores del algoritmo, como el *crossover* y la mutación, permiten la distribución de enemigos y la reparación de niveles generados. Aunque este enfoque permite la creación rápida y precisa de

Figura 19: Visualización del Evolutionary Dungeon Designer (EDD), y opciones de similitud en base a las métricas seleccionadas

mazmorras y la adición de enemigos, puede generar niveles con distribución inadecuada de enemigos que necesitan reparación adicional. No obstante, el algoritmo es capaz de generar niveles de mazmorras con misiones de puertas cerradas y una distribución adecuada de enemigos.

Finalmente, Viana et al. [Viana *et al.*, 2022a] utilizan una representación de matriz para los niveles de mazmorras, donde cada celda puede contener diferentes tipos de salas (ver Fig. 20). Los operadores del FI2Pop GA, que incluyen mutación y cruce, están diseñados para mantener la factibilidad de los niveles. Aunque este algoritmo puede generar una gran variedad de niveles jugables y equilibrados, y efectivamente integrar diferentes mecánicas de llaves y barreras, también enfrenta desafíos como la complejidad computacional y la necesidad de ajustes finos para evitar niveles no jugables. En conclusión, el algoritmo produce niveles de mazmorras variados y desafiantes, asegurando la accesibilidad de los objetivos a través de las mecánicas de llaves y barreras, como se puede observar en la definición de secciones del nivel representadas por distintos colores y en la verificación de que el posicionamiento de llaves y barreras funcionen correctamente (ver Fig. 21).

Figura 20: Representación de los niveles con llaves y barreras por genes

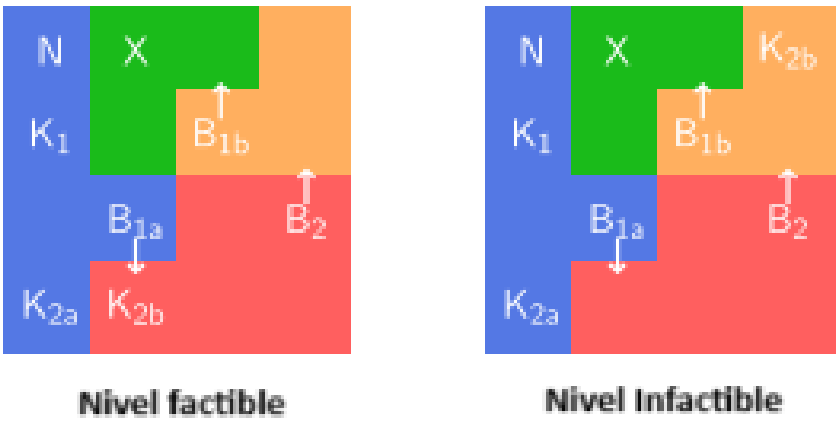


Figura 21: Representación de los niveles con llaves y barreras divididos por sectores, nivel factible o infactible

3.2.3. Aspectos relevantes en la generación de Mazmorras

3.2.3.1. Mecánicas de Llave y Barrera

Las mecánicas de llave y barrera son fundamentales en el diseño de mazmorras, ya que introducen elementos de puzzle y progresión que enriquecen la experiencia de juego. Estas mecánicas requieren que los jugadores encuentren ciertos objetos o completen ciertas tareas para desbloquear nuevas áreas del nivel. Varias investigaciones han explorado métodos avanzados para implementar estas mecánicas en la generación procedural de mazmorras.

[Pereira *et al.*, 2018, Pereira *et al.*, 2021] emplearon una estructura de grafo para generar mapas 2D con misiones de barrera. Este método organiza las mazmorras en una jerarquía que facilita la inclusión de caminos bloqueados que los jugadores deben desbloquear con llaves. Además, los autores mejoraron este enfoque utilizando una nueva técnica de cruce en un algoritmo evolutivo restringido, lo que permite una generación más eficiente y diversificada de mapas, manteniendo al mismo tiempo la coherencia y jugabilidad del diseño, la cual es reutilizada por el trabajo de [Viana *et al.*, 2022b]. Otra representación es utilizada en [Viana *et al.*, 2022a] para generar mazmorras con distintos tipos de relaciones de llaves y barreras.

[Weeks y Davis, 2022] adoptaron un algoritmo A* adaptado para generar matrices de llave y barrera. En su investigación, se centran específicamente en las misiones de barrera de tipo 1 → n, donde una llave puede desbloquear múltiples barreras. Este enfoque garantiza que las mazmorras tengan una estructura lógica y que la progresión del jugador sea fluida y satisfactoria. Al utilizar el algoritmo A*, se asegura que el camino óptimo y viable esté disponible para el jugador, manteniendo el balance entre desafío y accesibilidad.

En resumen, las mecánicas de llave y barrera no solo añaden un elemento de desafío y exploración, sino que también permiten una mayor profundidad en el diseño de mazmorras. Los avances en la generación procedural, como los aportados por Pereira, Viana y Weeks, demuestran cómo las técnicas evolutivas y los algoritmos de búsqueda pueden ser utilizados para crear niveles complejos y emocionantes que mantienen al jugador comprometido y motivado.

3.2.3.2. Aplicación de Simetría

En el diseño de niveles, la estética y la simetría juegan un papel crucial. La simetría no solo mejora el atractivo visual de los niveles, sino que también contribuye a la navegabilidad y

orientación del jugador dentro de mazmorras generadas aleatoriamente. Estudios como los de Marino et al. [Mariño y Lelis, 2016] que utilizan un branch and bound para la generación de niveles de tipo plataformas y Summerville et al. [Summerville *et al.*, 2017] que utilizan CNN para aprovechar la simetría central y axial entre los cuatro sectores de una pantalla como nivel, y utilizan la simetría como métrica para la estética en plataformas como Mario (ver Fig. 22). Estos estudios muestran cómo la simetría puede influir positivamente en la percepción del jugador, haciendo que los niveles sean más agradables y comprensibles.

En el diseño de niveles, la estética y la simetría juegan un papel crucial. La simetría no solo mejora el atractivo visual de los niveles, sino que también contribuye a la navegabilidad y orientación del jugador dentro de mazmorras generadas aleatoriamente. Estudios como los de Marino et al. [Mariño y Lelis, 2016], que utilizan un algoritmo de branch and bound para la generación de niveles de plataformas, y los de Summerville et al. [Summerville *et al.*, 2017], que emplean redes neuronales convolucionales (CNN) para aprovechar la simetría central y axial entre los cuatro sectores de una pantalla como nivel, utilizan la simetría como métrica para la estética en plataformas como Mario (ver Fig. 22). Estos estudios muestran cómo la simetría puede influir positivamente en la percepción del jugador, haciendo que los niveles sean más agradables y comprensibles.

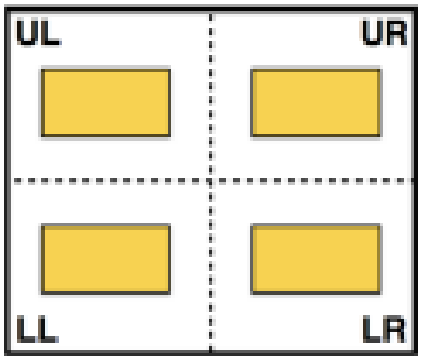


Figura 22: Uso de simetría completa, simetría con respecto a los ejes horizontales, verticales y con respecto al punto central.

De manera similar, en el contexto de juegos Roguelike y Roguelite, la aplicación de la simetría en la generación procedural de mazmorras tiene implicaciones significativas tanto para el diseño de juegos como para la experiencia del jugador. La simetría, además de mejorar el atractivo visual, también ayuda en la navegabilidad y orientación del jugador. En juegos donde las mazmorras se generan de manera procedural, una estructura simétrica puede proporcionar puntos de referencia visuales que faciliten la exploración y eviten que el jugador se sienta desorientado.

La simetría ha sido un criterio clave en la concepción de generación la de niveles de videojue-

gos, con aplicaciones exitosas en juegos de vista superior utilizando Fi-2pop [Zafar *et al.*, 2020], donde se utilizan métricas de estéticas. Fi-2pop es un algoritmo genético que se ha utilizado para generar niveles con criterios estéticos, incluyendo la simetría. [Alvarez *et al.*, 2018] se exploró este aspecto en su trabajo en el proyecto EDD [Baldwin *et al.*, 2017], integrando criterios estéticos en la generación de niveles. Su investigación no solo presenta métodos para preservar la simetría en las sugerencias de diseño, sino también discute cómo medir la simetría y evolucionar niveles a través de elementos simétricos. Este enfoque permite una creación más controlada y estéticamente agradable de mazmorras, manteniendo un equilibrio entre la jugabilidad y la estética.

Una simetría excesiva puede llevar a la previsibilidad, lo que puede reducir el desafío y la sorpresa que son fundamentales en la exploración de mazmorras. Por lo tanto, es necesario entender cómo implementar esta estética de manera efectiva en la generación de mazmorras.

Un equilibrio entre simetría y asimetría es clave para mantener tanto el atractivo visual como el elemento sorpresa, esenciales en la exploración de mazmorras. Mientras que la simetría puede proporcionar una base estructural y puntos de referencia claros, la asimetría puede introducir variaciones y elementos inesperados que mantienen al jugador interesado y desafiado. Este balance asegura que los niveles no solo sean visualmente agradables, sino también interesantes y desafiantes para explorar.

3.2.4. Otras Técnicas

La co-creación y el control del diseñador son aspectos críticos en la generación de mazmorras. Whitehead et al. (2020) propusieron métodos para la co-creación de mazmorras, permitiendo que los diseñadores humanos colaboren con algoritmos generativos para ajustar y refinar los diseños generados [Whitehead, 2020], utilizando diseños de estructura del nivel para luego utilizar restricciones lineales enteras, resueltas mediante un solucionador de teorías de módulo de satisfacibilidad (SMT), para ubicar las habitaciones en la estructura creada por el usuario (ver Fig. 23).

Los modelos de aprendizaje automático, como las redes neuronales y los modelos de aprendizaje profundo, han sido empleados para aprender patrones a partir de ejemplos de niveles de mazmorras existentes. Summerville y Mateas [Summerville *et al.*, 2015] demostraron cómo estos modelos pueden capturar la complejidad y diversidad de los niveles manualmente diseñados, generando nuevos niveles con alta calidad y jugabilidad utilizando Bayesian Networks para aprender la topología de los niveles. Estos modelos pueden aprender las características de diseño que hacen que un nivel sea interesante y desafiante, replicando esas

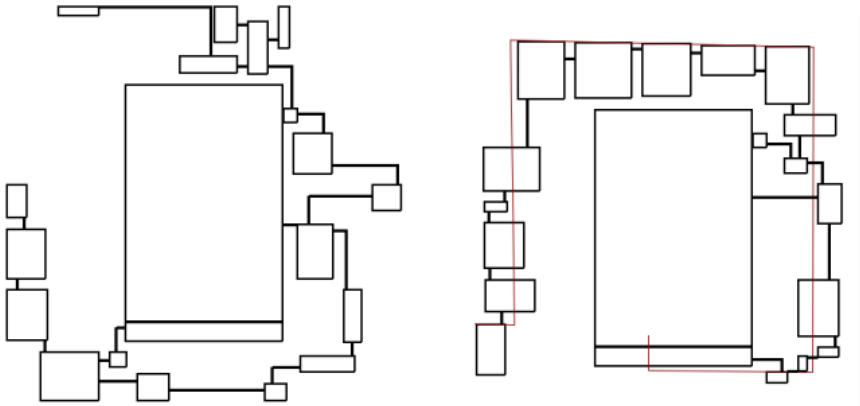


Figura 23: Espacio generado en base a la asignación de la línea roja, que guía la generación del nivel en base a estructura

características en nuevos niveles generados.

3.3. Representaciones en la Generación Procedural de Mazmorras

En la literatura, se han utilizado diversas representaciones para la generación procedural de mazmorras, cada una con sus propias ventajas y limitaciones:

- **Representación en grafo**: Las mazmorras se representan como un grafo donde cada nodo es una habitación y los enlaces son pasillos. Esta representación facilita la creación de caminos lineales y ramificados, permitiendo un control preciso sobre la estructura de la mazmorra. Cada nodo puede tener hijos que representan habitaciones o caminos secundarios, asegurando una jerarquía clara y fácil de manejar. Esta estructura es especialmente útil para diseñar niveles con un flujo dirigido, donde se desea controlar el orden de exploración del jugador.
- **Representación en Grilla**: Las mazmorras se generan en una grilla bidimensional donde cada celda puede ser un muro, un pasillo o una habitación. Este método es simple y flexible, adecuado para implementar algoritmos de búsqueda y de generación basados en grillas. La representación en grilla permite fácilmente aplicar técnicas como el algoritmo de vida artificial (cellular automaton) para crear cavidades y pasajes que imitan estructuras naturales. Esta representación es altamente visual y facilita la implementación de algoritmos que necesiten verificar la vecindad de celdas adyacentes.

Esta tesis se basa en el trabajo de Leonard (ver Sección 3.2.2) para la generación de mazmorras. Adopta tanto los ejemplos de generación proporcionados por Leonard como las deci-

siones de diseño integradas en el CEA. Este enfoque asegura que las mazmorras generadas sean consistentes con las mejores prácticas de diseño de niveles, proporcionando un marco sólido sobre el cual construir.

3.4. Resumen

La Generación Procedural de Mazmorras (PDG) es una herramienta crucial en el desarrollo de videojuegos contemporáneos, permitiendo la creación dinámica y variada de entornos de juego. La revisión bibliográfica resalta la continua necesidad de innovación en este ámbito. La combinación de algoritmos evolutivos con la propuesta del Algoritmo Evolutivo de 2 Pasos presenta nuevas oportunidades para mejorar la calidad y diversidad del contenido generado proceduralmente. Estas técnicas no solo prometen transformar la manera en que se crean y experimentan los videojuegos, sino también ofrecer experiencias de juego más inmersivas y atractivas para los jugadores.

Un resumen general de las técnicas observadas se puede apreciar en la Tabla 2 y Tabla 3.

En el siguiente capítulo, se presentan brevemente los algoritmos genéticos y evolutivos, explicando sus fundamentos y su aplicación en la generación de mazmorras. Se detallan conceptos como la dimensionalidad del espacio de búsqueda, la exploración y explotación, y la división en dos fases del espacio de búsqueda.

Paper	Técnica	Problema	Introduce
[Cerny y Dechterenko, 2015]	EA	Agentes para Niveles de Mazmorras	Agentes de juego
[Ashlock, 2015]	EA	Niveles de Caverna	CA para la evolución
[Liapis *et al.*, 2015]	Fi-2pop	Niveles de Mazmorra	Personas procedurales
[Baghdadi *et al.*, 2015]	EA	Niveles de Spelunky	Agentes de excavación
[Baldwin *et al.*, 2017]	Fi-2pop	Habitaciones de Mazmorra	Uso de patrones
[Liapis, 2017]	Fi-2pop	Niveles de Mazmorra	Creación y refinamiento
[Alvarez *et al.*, 2018]	Fi-2pop	Habitaciones de Mazmorra	Estética como criterio de evaluación
[Melotti y de Moraes, 2019]	DNS	Niveles de Mazmorras	Agrega NS para la generación
[Pereira *et al.*, 2018]	EA	Niveles de Mazmorras	Nueva representación
[Ruela y Delgado, 2018]	EA	Niveles de Mazmorras	Representación de redes
[Alvarez *et al.*, 2019]	Fi-2pop	Habitaciones de Mazmorra	Map of Elites con uso de estética
[Zafar *et al.*, 2020]	Fi-2pop	Niveles de Mazmorras	Agente como evaluador de niveles
[Pereira *et al.*, 2021]	EA	Niveles de Mazmorras	Nueva mutación y crossover restringidos
[Viana *et al.*, 2022b]	EA	Niveles de Mazmorra y Enemigos	Agrega Map of Elites
[Viana *et al.*, 2022a]	Fi-2pop	Niveles de Mazmorra	Representación de grilla y tipos de barreras

Tabla 2: Resumen comparativo de técnicas evolutivas.

Paper	Técnica	Problema	Introduce
[Dormans, 2010]	Generative Grammar	Niveles de Mazmorras	División de espacio de misión y nivel
[Lavender y Thompson, 2017]	Generative Grammar	Niveles de Mazmorras	Nuevas reglas de generación
[Green *et al.*, 2019]	2 Step Generative	Niveles de Mazmorras	Layouts y distribución de objetos
[Gellel y Sweetser, 2020]	Context-Free Grammar	Niveles de Mazmorras	CA para trabajar con el espacio generado
[Mariño y Lelis, 2016]	Branch and Bound	Niveles de Mazmorras	Emplean simetría para la generación
[Summerville *et al.*, 2017]	CNN	Niveles de Mazmorras	Simetría y otras métricas
[Weeks y Davis, 2022]	A*	Llaves y Barreras	Entendiendo las distancias entre elementos
[Summerville *et al.*, 2015]	Bayesian Network	Niveles de Mazmorras	Aprender topología

Tabla 3: Resumen comparativo de técnicas.

CAPÍTULO 4
Algoritmos Evolutivos

4.1. Introducción

Los algoritmos evolutivos son técnicas de optimización inspiradas en los procesos de selección natural y genética. Se utilizan ampliamente en problemas de búsqueda y optimización donde el espacio de búsqueda es complejo. Estos algoritmos emulan la evolución natural para encontrar soluciones aproximadas a problemas complejos, iterando a través de generaciones de posibles soluciones y aplicando operadores evolutivos como selección, cruce y mutación.

4.2. Algoritmos Genéticos (GA)

Un algoritmo genético (GA) fue propuesto por Holland [Holland, 1992], es una técnica heurística basada en la teoría de la evolución natural de Charles Darwin. Los GAs se utilizan para encontrar soluciones aproximadas a problemas de optimización y búsqueda. Consisten en una población de individuos (soluciones candidatas) que evolucionan a lo largo de generaciones.

El proceso de un GA generalmente incluye:

- **Inicialización:** Se crea una población inicial de individuos de manera aleatoria. Estos individuos representan posibles soluciones al problema.
- **Evaluación:** Cada individuo se evalúa utilizando una función de aptitud que mide la calidad de la solución. Esta función está diseñada para reflejar qué tan bien un individuo resuelve el problema.
- **Selección:** Se seleccionan los individuos más aptos para reproducirse, utilizando métodos como la selección por torneo o la ruleta. Los individuos seleccionados son aquellos que tienen mayor probabilidad de contribuir con sus genes a la siguiente generación.
- **Crossover (Cruce):** Se combinan pares de individuos para producir descendencia mediante operaciones de cruce. En este proceso intercambian segmentos de los padres para crear hijos con características de ambos.
- **Mutación:** Se introducen cambios aleatorios en algunos individuos para mantener la diversidad genética. La mutación asegura que el algoritmo no quede atrapado en óptimos locales al explorar nuevas áreas del espacio de búsqueda.

- **Reemplazo:** La nueva generación reemplaza a la anterior. Este ciclo se repite hasta que se cumple un criterio de parada, como alcanzar un número máximo de generaciones o una aptitud satisfactoria.

4.3. Algoritmos Evolutivos (EA)

Los algoritmos evolutivos (EAs) son una generalización de los algoritmos genéticos y pueden incluir estrategias evolutivas, programación genética y otros métodos basados en la evolución. Mientras que los GAs suelen representar soluciones como cadenas de bits, los EAs pueden utilizar representaciones más complejas, como vectores de números reales o grafos.

4.4. Espacio de Búsqueda

El espacio de búsqueda en problemas de optimización representa todas las posibles soluciones. En el contexto de la generación de mazmorras, el espacio de búsqueda incluye todas las configuraciones posibles de habitaciones, pasillos, enemigos y objetos. La exploración efectiva de este espacio de búsqueda es crucial para encontrar diseños de mazmorras que sean equilibrados, desafiantes y atractivos para los jugadores.

4.4.1. Dimensionalidad del Espacio de Búsqueda

La dimensionalidad del espacio de búsqueda se refiere al número de variables independientes que definen una solución. En la generación de mazmorras, esto puede incluir:

- **Diseño de la estructura:** La disposición y tamaño de las habitaciones y pasillos, que afectan la navegabilidad y la estrategia del juego.
- **Elementos de juego:** La ubicación de enemigos, trampas y tesoros, que determinan la dificultad y el atractivo del nivel.
- **Condiciones de victoria:** La ubicación de llaves y puertas cerradas, que establecen los objetivos y desafíos que el jugador debe superar.

Cada una de estas variables añade una dimensión al espacio de búsqueda, aumentando su complejidad y el número de posibles configuraciones.

4.5. Aplicaciones en la Generación de Mazmorras

En la generación procedural de mazmorras, los EAs han sido utilizados para:

- **Optimizar el diseño de niveles:** Crear mazmorras que sean coherentes, equilibradas y desafiantes. Los EAs ayudan a diseñar niveles que no solo sean interesantes y visualmente atractivos, sino también bien balanceados en términos de dificultad y jugabilidad.
- **Generar contenido dinámico:** Producir diferentes configuraciones de mazmorras en cada partida, aumentando la rejugabilidad. Esto permite a los desarrolladores ofrecer una experiencia de juego fresca y única cada vez que el jugador entra en una mazmorra.
- **Automatizar el proceso de diseño:** Reducir el trabajo manual de los diseñadores de niveles, permitiendo la creación rápida de contenido variado. La automatización facilita la generación de grandes volúmenes de contenido, asegurando que cada mazmorra sea única y desafiante.

Los algoritmos evolutivos proporcionan un marco robusto para abordar los desafíos en la generación procedural de mazmorras, permitiendo la creación de entornos ricos y diversos que mejoran la experiencia de juego.

4.5.1. División en Dos Fases del Espacio de Búsqueda

Una estrategia eficaz para manejar la complejidad del espacio de búsqueda en la generación de Mazmorras es dividir el problema en dos fases: generación de la estructura básica y detallado del contenido. Esto reduce la dimensionalidad y facilita la exploración y explotación del espacio de búsqueda en cada fase.

Primera Fase: Generación de la Estructura Básica En esta fase, el enfoque se centra en la disposición y tamaño de las habitaciones y pasillos. El espacio de búsqueda incluye configuraciones generales de la estructura de la mazmorra, asegurando que sea navegable y tenga sentido en términos de diseño de niveles. Las variables en esta fase pueden incluir:

- Número y tamaño de las habitaciones.
- Conectividad entre habitaciones y pasillos.
- Distribución espacial general de la mazmorra.

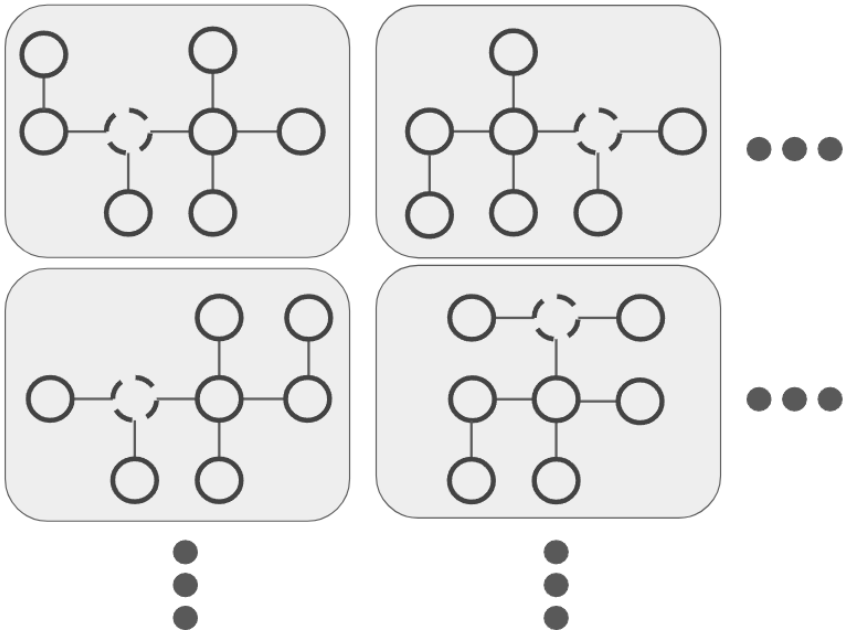


Figura 24: Espacio de búsqueda, enfocado en la forma del mapa

De esta manera el manejo en el espacio de búsqueda sólo se limita a la forma del nivel, dejando aparte la ubicación de llaves y barreras 24.

Segunda Fase: Detallado del Contenido En la segunda fase, se detalla el contenido dentro de la estructura generada. Esto incluye la ubicación de enemigos, trampas, tesoros, y otros elementos de juego. Las variables en esta fase incluyen:

- Posición de enemigos y su nivel de dificultad.
- Ubicación de trampas y su tipo.
- Distribución de tesoros y objetos.
- Condiciones de victoria específicas, como la ubicación de llaves y puertas.

Al dividir el espacio de búsqueda en estas dos fases, se simplifica el problema de optimización, permitiendo a los algoritmos evolutivos enfocarse en un subconjunto más manejable de variables en cada etapa, primero seleccionando este espacio (ver Fig. 25), para luego abordar la evolución dentro de este espacio de búsqueda (ver Fig. 26). Esto mejora la eficiencia del proceso de búsqueda y aumenta la probabilidad de encontrar soluciones óptimas o cercanas a lo óptimo.

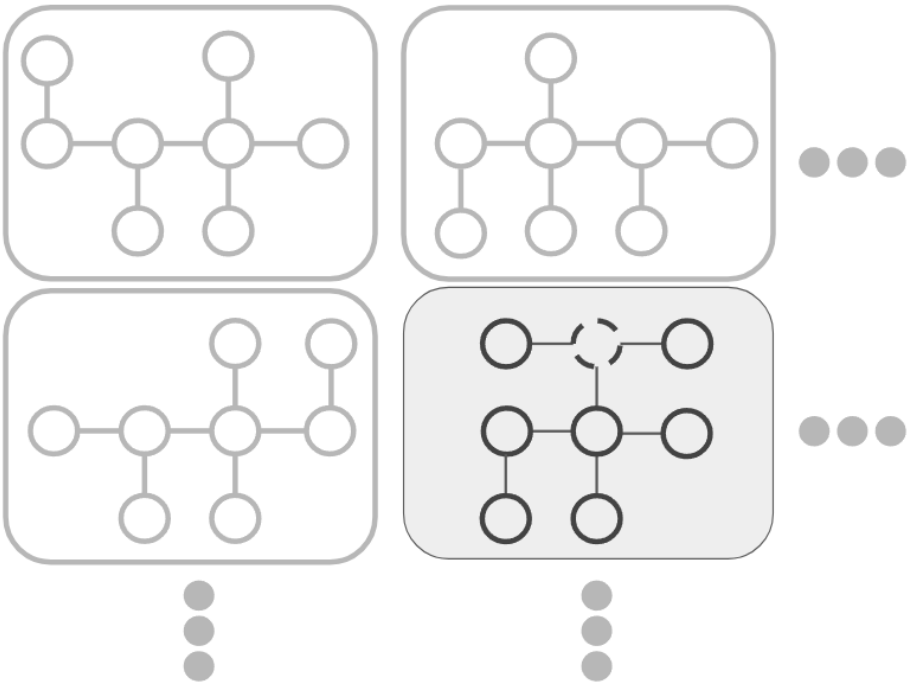


Figura 25: Espacio de búsqueda, enfocado en la selección de un subespacio de búsqueda para la segunda fase.

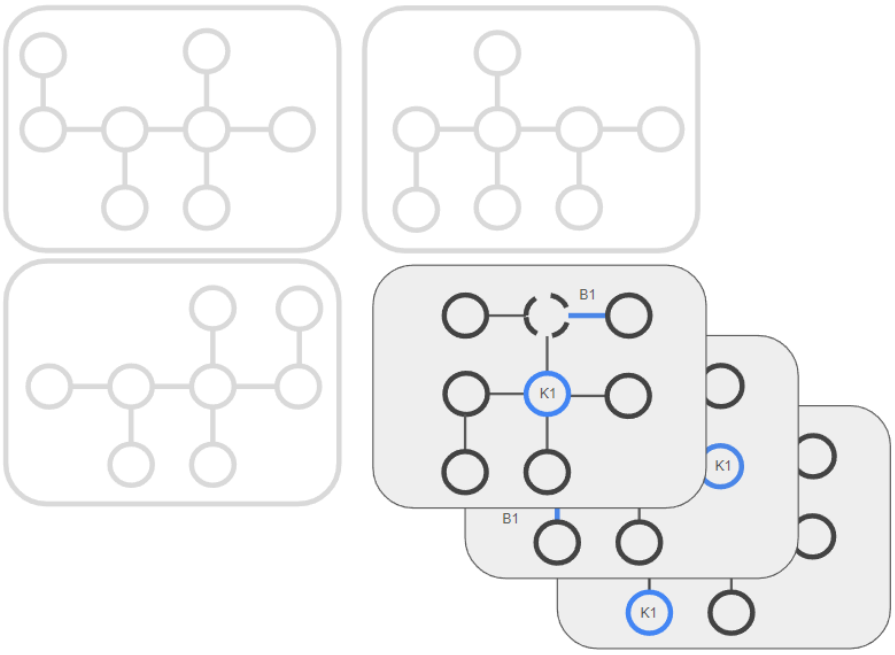


Figura 26: Espacio de búsqueda, enfocado en la ubicación de llaves y barreras en el mapa.

4.6. Resumen

En resumen, los algoritmos evolutivos ofrecen poderosas herramientas para abordar problemas complejos de optimización, como la generación procedural de mazmorras. Su capacidad para explorar y explotar eficientemente un vasto espacio de búsqueda, junto con la flexibilidad para representar y manipular diversas configuraciones de soluciones, los hace ideales para diseñar niveles de juego que sean atractivos, equilibrados y dinámicos. Al dividir el proceso de generación en dos fases, se logra una gestión más efectiva de la complejidad, permitiendo a los algoritmos enfocarse en aspectos específicos del diseño de mazmorras y mejorar la calidad general de las soluciones encontradas.

En el próximo capítulo, se propondrá un algoritmo evolutivo para la generación de Mazmorras. Se detalla cada uno de sus componentes y decisiones de diseño.

CAPÍTULO 5
Detalles Técnicos del Algoritmo Propuesto

En este capítulo se describe en detalle el algoritmo propuesto, incluyendo las decisiones de diseño tomadas para cada componente. En el contexto de la generación procedural de mazmorras, la representación de la mazmorra es crucial para definir la estructura y las propiedades de los niveles generados. Este trabajo utiliza una representación basada en grafos.

Se presenta un algoritmo evolutivo de dos pasos con simetría horizontal aplicado a la generación procedural de mazmorras. La representación de la mazmorra se basa en un grafo $G = (V, E)$, donde los nodos V representan habitaciones y las aristas E representan las conexiones entre ellas. Este enfoque asegura que no haya habitaciones superpuestas y determina la ubicación de cada una, comenzando siempre desde la posición inicial (0,0).

El algoritmo evolutivo se implementa en dos fases:

- **Fase Estructural:** Se generan múltiples soluciones iniciales utilizando un vector de posicionamiento para distribuir las habitaciones sin traslape, buscando mejorar la conectividad y la estructura general de la mazmorra.
- **Fase de alocación de llaves y barreras:** Las soluciones se refinan mediante la alocación de llaves y barreras dentro de los mapas estructurados, evitando crear mapas imposibles y acercándose a las entradas ingresadas por diseñador (el usuario) de juegos, en cuanto a la cantidad de llaves y barreras.

5.1. Representación

Para garantizar la coherencia y jugabilidad de las mazmorras generadas, se imponen ciertas restricciones en la representación de grafos. Estas restricciones son esenciales para asegurar que la mazmorra resultante no solo sea navegable y funcional, sino también desafiante y entretenida para los jugadores.

- **Conectividad**: El grafo debe ser conexo, lo que implica que todas las habitaciones deben ser accesibles desde cualquier otra habitación. Esta conectividad asegura que no existan habitaciones aisladas o inalcanzables, lo que podría frustrar a los jugadores. Para lograr esto, se utiliza un algoritmo de generación que añade habitaciones y corredores, de tal manera que cada nueva habitación se conecta con al menos una ya existente, formando un único componente conexo.
- **Ciclicidad controlada**: Se implementa un control estricto, evitando la generación de rutas que pueden conectar distintas secciones del grafo, y de esta forma impedir ca-

minos dentro del mapa final que incluir ciclos. Aunque en muchos casos los ciclos pueden añadir interés y complejidad a una mazmorra, en este contexto se busca mantener una estructura acíclica para simplificar la navegación y la lógica del juego. Esto se logra mediante la construcción del grafo, donde cada nodo (habitación) tiene una ruta única desde el nodo inicial (entrada) hasta el nodo final (salida).

- **Distribución de elementos clave**: Se imponen restricciones adicionales para la colocación de elementos clave, como puertas bloqueadas y llaves. Estas restricciones aseguran que los jugadores encuentren estos elementos en un orden lógico y progresivo, evitando situaciones donde una puerta bloqueada aparezca antes de encontrar la llave correspondiente. La colocación de estos elementos sigue una lógica de progresión en el grafo, donde los nodos que contienen llaves o puertas están ubicados estratégicamente para mantener el flujo del juego.
- **Diversidad y variabilidad**: A pesar de las restricciones impuestas, se busca mantener un alto grado de diversidad y variabilidad en las mazmorras generadas. Esto se logra minduciendo variaciones aleatorias en la estructura del grafo y en la disposición de las habitaciones, asegurando que cada mazmorra sea única y ofrezca un nuevo desafío. Además, se incluyen parámetros configurables que permiten ajustar el nivel de complejidad de la mazmorra según los valores de entrada que pueden ser asignados por un diseñador.

Las restricciones en la estructura del grafo no solo garantizan la coherencia y jugabilidad de las mazmorras generadas, sino que también contribuyen a crear una experiencia de juego equilibrada, desafiante y variada. El control de la conectividad, ciclicidad y distribución de elementos, busca asegurar que cada mazmorra generada es funcionalmente correcta.

5.1.1. Ventajas de la Representación Basada en Grafos

El uso de una representación basada en grafos ofrece varias ventajas significativas para la generación procedural de mazmorras, proporcionando una estructura robusta y versátil que facilita tanto el diseño como la implementación de niveles de juego complejos y variados.

- **Flexibilidad**: La representación basada en grafos permite una fácil modificación y ajuste de la estructura de la mazmorra. Cada nodo del grafo representa una habitación, y las aristas representan los corredores o conexiones entre ellas. Esta representación modular facilita la adición, eliminación o reconfiguración de habitaciones y caminos sin necesidad de rehacer todo el diseño de la mazmorra. Esto es particularmente útil en el desarrollo iterativo de juegos, donde los cambios frecuentes son comunes.
- **Coherencia estructural**: Al usar un grafo para representar la mazmorra, se asegura que todas las habitaciones estén correctamente conectadas y sean accesibles desde

cualquier punto del mapa. Esta propiedad de conectividad es crucial para evitar habitaciones aisladas que no puedan ser alcanzadas por los jugadores, lo que podría resultar en una experiencia de juego frustrante. Además, la coherencia estructural del grafo ayuda a mantener una navegación lógica y fluida dentro del nivel, mejorando la jugabilidad y la satisfacción del jugador.

- **Adaptabilidad**: La representación en grafos facilita la incorporación de nuevas características y restricciones sin necesidad de cambiar fundamentalmente la representación subyacente. Por ejemplo, es posible agregar elementos como puertas, llaves, trampas, y enemigos simplemente añadiendo nuevos nodos y aristas o etiquetando los existentes con atributos adicionales. Esta capacidad de adaptación permite que el diseño de la mazmorra evolucione y se complejize conforme avanza el desarrollo del juego, permitiendo a los diseñadores experimentar con diferentes mecánicas y dinámicas de juego.

- **Escalabilidad**: Los grafos son inherentemente escalables, lo que significa que se pueden expandir fácilmente para acomodar mazmorras de mayor tamaño y complejidad, sin perder la coherencia estructural. Esta escalabilidad es particularmente importante en juegos que requieren niveles grandes y detallados, ya que permite mantener la consistencia en la estructura de la mazmorra a medida que se añaden nuevas áreas y desafíos.

- **Facilidad de implementación de algoritmos de búsqueda y generación**: Los grafos proporcionan una base sólida para implementar diversos algoritmos de búsqueda y generación, como la búsqueda en anchura (BFS), búsqueda en profundidad (DFS), algoritmos de caminos mínimos (como Dijkstra o A*), y algoritmos de generación procedural. Estos algoritmos pueden utilizarse para diseñar rutas óptimas, generar mazmorras que cumplan con ciertos criterios de dificultad, o para crear laberintos complejos con una estructura lógica y jugable.

- **Visualización y depuración**: La representación gráfica de los grafos permite una visualización clara y sencilla de la estructura de la mazmorra, lo que facilita la depuración y el análisis del diseño. Los desarrolladores pueden visualizar el grafo para identificar posibles problemas en la conectividad, la distribución de habitaciones, y la colocación de elementos clave, permitiendo ajustes rápidos y precisos para mejorar la calidad del nivel.

En resumen, la representación de la mazmorra como un grafo proporciona una base sólida para la generación procedural, garantizando que las mazmorras resultantes sean coherentes, variadas y jugables. La flexibilidad, coherencia estructural, adaptabilidad, escalabilidad, facilidad de implementación de algoritmos, y la capacidad de visualización y depuración hacen que los grafos sean una herramienta ideal para diseñar y generar niveles complejos en juegos de mazmorras.

5.2. Función de Evaluación

La función de evaluación juega un papel crucial en la evaluación de la calidad de las mazmorras generadas por el algoritmo. Su propósito es medir la disparidad entre la entrada deseada del diseñador y la mazmorra generada, con el objetivo de minimizar esta diferencia. Siguiendo el enfoque descrito en el artículo [Pereira *et al.*, 2021], se calcula la diferencia absoluta de cada parámetro como Δ, con una ponderación doble para los requisitos especificados por el usuario y la distancia a las llaves necesarias obtenidas por el algoritmo A*.

La fórmula de la función de evaluación se expresa de la siguiente manera:

$$f = 2(\Delta_{\text{habitaciones}} + \Delta_{\text{llaves}} + \Delta_{\text{barreras}} + \Delta_{\text{coef\_de\_linearidad}}) + \Delta_{\text{llaves\_necesarias}} \tag{11}$$

$$f = 2(\Delta_{\text{habitaciones}} + \text{llaves} + \text{barreras} + \Delta_{\text{coef\_de\_linearidad}}) + \text{llaves\_necesarias} \tag{12}$$

Donde:

- $\Delta_{\text{habitaciones}}$: Representa la diferencia en el número de habitaciones entre la mazmorra generada y el número deseado por el diseñador.
- Δ_{llaves}: Indica la discrepancia en la distribución de llaves en la mazmorra generada en comparación con las especificaciones del diseñador.
- Δ_{barreras}: Mide la discrepancia en la ubicación y distribución de las barreras en la mazmorra generada con respecto a los requisitos del diseñador.
- $\Delta_{\text{coef\_de\_linearidad}}$: Evalúa la diferencia en la linealidad de los pasillos y corredores de la mazmorra generada en comparación con la preferencia del diseñador.
- $\Delta_{\text{llaves\_necesarias}}$: Representa la distancia promedio a las llaves necesarias en la mazmorra generada, obtenida mediante el algoritmo A*.

La función de evaluación es la misma en ambas fases del algoritmo. Durante la generación de la mazmorra, se utilizan los valores de los parámetros calculados por esta función para ajustar dinámicamente la distribución de llaves, barreras y llaves necesarias según las entradas definidas por el diseñador, con el objetivo de minimizar la diferencia entre la mazmorra generada y la entrada deseada.

5.2.1. Aplicación de la Función de evaluación en cada fase

Dado que se dividió la operación de generación en dos fases, es crucial entender cómo se aplica la función de evaluación en cada una:

1. **Primera fase: Generación estructural**
    - En esta etapa inicial de generación, la función de evaluación se encarga principalmente de evaluar la coherencia estructural y la accesibilidad de las habitaciones en la mazmorra.
    - La función de evaluación pondera las diferencias en el número de habitaciones, la distribución de las llaves y las barreras iniciales para determinar la calidad de la estructura básica de la mazmorra generada.
    - Los parámetros evaluados en esta fase incluyen la disposición general de las habitaciones, la conectividad entre ellas y la distribución inicial de elementos como llaves y barreras.
2. **Segundo fase: Alocación de llaves y barreras**
    - En esta fase, la función de evaluación se enfoca en los detalles más finos de la mazmorra generada, como la colocación precisa de las llaves necesarias y la ubicación de las barreras específicas para garantizar la jugabilidad.
    - La función de evaluación ajusta los parámetros de la mazmorra para asegurar que todas las restricciones de jugabilidad definidas por el diseñador se cumplan adecuadamente. Esto incluye la optimización de la distancia a las llaves necesarias utilizando el algoritmo A*, para garantizar que la mazmorra sea desafiante pero alcanzable para el jugador.
    - Durante esta fase, la función de evaluación se utiliza para refinar la distribución de elementos como llaves y barreras, así como para ajustar la conectividad de las habitaciones para crear una experiencia de juego equilibrada y satisfactoria.

5.3. Estructura del Algoritmo

Se propone el algoritmo el cual se dividio en dos pasos (2-STEP EA) para la generación procedural de mazmorras, dos fases: la fase de generación estructural y la fase de evolución de distribución llave-barrera. Este enfoque permite una generación controlada y una evaluación iterativa de las mazmorras para asegurar que cumplan con los criterios de diseño deseados. Al separar la generación de la Mazmorra en dos partes, el algoritmo se encargará de generar la forma del mapa, sin tener que preocuparse de la ubicación de llaves y barreras, y los conflictos que esto puede generar a la hora de evolucionar estos niveles. La evolución de los niveles de mazmorras se define a continuación:

El algoritmo evolutivo en dos pasos funciona de la siguiente manera:

1. **Inicialización**: Se crea una población inicial de configuraciones de mazmorras, cada una representando una posible solución. Esta población inicial puede ser generada aleatoriamente o basada en patrones predefinidos que aseguren una diversidad adecuada de soluciones.

Algorithm 1 2-Step Evolutionary Algorithm

1: **procedure** 2StepEA($popsize$)
2: $generation \leftarrow CreateInitialPop(popsize)$
3: $generation \leftarrow EvolveForm(generation)$
4: $generation \leftarrow EvolveBK(generation)$
5: $best\_ind \leftarrow GetBest(generation)$
6: **return** $best\_ind$
7: **end procedure**

2. **Inicialización**: Se crea una población inicial a partur de la creación de mazmorras iniciales, las cuales parten de un nodo raíz para la estructura de la mazmorra. A partir de este nodo, se generan nuevas habitaciones mediante un proceso iterativo, agregándolas a una lista de nodos disponibles. La expansión de la mazmorra se realiza seleccionando nodos aleatoriamente y añadiendo habitaciones hijas en direcciones variadas, lo que garantiza diversidad en la estructura generada.

3. **Evolución de la Forma**: La población evoluciona para optimizar la forma de la mazmorra. En esta etapa, se evalúa y selecciona a los individuos basándose en su adecuación a los criterios estructurales definidos, como la conectividad y la distribución espacial. Se aplican operadores genéticos para mejorar las configuraciones, buscando soluciones que maximicen la conectividad y minimicen las redundancias.

4. **Evolución de Llaves y Barreras**: La población resultante de la fase de forma se somete a una segunda evolución para distribuir las llaves y barreras. Esta evolución asegura que los elementos críticos para la jugabilidad estén correctamente ubicados, siguiendo criterios como la accesibilidad y la progresión lógica. Se optimizan las ubicaciones de estos elementos para crear desafíos equilibrados y coherentes.

5. **Selección del Mejor Individuo**: Se selecciona el mejor individuo de la población final, el cual representa la mazmorra generada que mejor cumple con los criterios de diseño especificados. Este individuo es evaluado exhaustivamente para asegurar que cumple con todos los requisitos de estructura y jugabilidad, garantizando una experiencia de juego óptima.

Este enfoque en dos pasos permite una separación clara entre la generación de la estructura y la optimización de los elementos interactivos, facilitando un control preciso sobre el diseño de la mazmorra.

5.3.1. Fase 1: Generación estructural

La **fase de generación estructural** se centra en crear la forma básica de la mazmorra. En esta etapa, los nodos y sus conexiones se desarrollan para formar la estructura global, asegurando que la configuración cumpla con los requisitos de habitaciones y coeficiente lineal

especificados. Este paso es fundamental para establecer una base sólida sobre la cual se pueda construir una mazmorra coherente y navegable. Los criterios estructurales incluyen la conectividad de las habitaciones, evitando la creación de caminos muertos o habitaciones inaccesibles, y la distribución espacial de los nodos para asegurar una exploración lógica y satisfactoria por parte del jugador.

- **Creación de la Población Inicial**: Se genera una población inicial de configuraciones de mazmorras, cada una con una estructura básica que incluye nodos y conexiones iniciales.
- **Evaluación y Selección**: la función de evaluación evalua las configuraciones de acuerdo con criterios de conectividad y distribución, seleccionando aquellas que mejor cumplan con los requisitos especificados.
- **Aplicación de Operadores Evolutivos**: Se aplican operadores evolutivos de mutación y cruce para generar nuevas configuraciones a partir de las soluciones seleccionadas.
- **Iteración**: Este proceso de evaluación, selección y aplicación de operadores se repite durante varias generaciones hasta que la población converge a una mejor solución en términos de estructura.

En esta fase se evoluciona la forma del nivel, ignorando la generación de llaves y barreras para concentrarse únicamente en la estructura. Se inicia con la generación de una población inicial y se evoluciona esta población, enfocándose en las características de la cantidad de habitaciones y el coeficiente lineal.

5.3.1.1. Ejemplo de Generación

Para comprender de mejor manera el proceso de generación de mazmorras, primero se definen los requerimientos de generación de mapas, como se muestra en el siguiente ejemplo.

Habitaciones	Llaves	Barreras	Coeficiente Lineal	Llaves necesarias
7	0	0	3.0	0

Tabla 4: Requerimientos de habitaciones, llaves, barreras y coeficiente lineal para la generación de un mapa.

Este ejemplo de requerimientos se enfoca únicamente en la generación de la forma del mapa, el cual requiere que el mapa tenga siete habitaciones conectadas y que tenga un coeficiente lineal de 3.0. Esto implica que se intentará conectar completamente cada uno de los nodos que tiene la mazmorra, creando una estructura interconectada que garantice la accesibilidad y jugabilidad de cada sección del mapa.

El proceso de generación sigue una serie de pasos bien definidos para asegurar que los criterios especificados se cumplan. A continuación, se detalla el procedimiento:

- **Inicialización**: Se comienza con un nodo raíz o nodo padre (P), que representa el punto de inicio de la mazmorra. Este nodo servirá como el origen desde el cual se generarán y conectarán las demás habitaciones.

- **Generación de Nodos**: Se crean seis nodos adicionales para cumplir con el requerimiento total de siete habitaciones. Cada nodo representa una habitación distinta en la mazmorra. Los nodos son designados con etiquetas para su identificación y referencia durante el proceso de conexión.

- **Conexión de Nodos**: Utilizando el coeficiente lineal de 3.0, se asegura una alta conectividad entre los nodos. Este coeficiente determina la densidad de las conexiones entre los nodos, lo que implica que cada nodo intentará maximizar sus conexiones hasta alcanzar un promedio de tres conexiones por nodo. Este nivel de conectividad es crucial para evitar caminos muertos y asegurar que todas las habitaciones sean accesibles.

- **Distribución Espacial**: Los nodos se ubican alrededor del nodo padre inicial (P). Las habitaciones se distribuyen en tres direcciones principales: izquierda (L), abajo (D) y derecha (R) del nodo padre, formando una estructura equilibrada y fácil de recorrer. Esta distribución se muestra en la Figura 27.

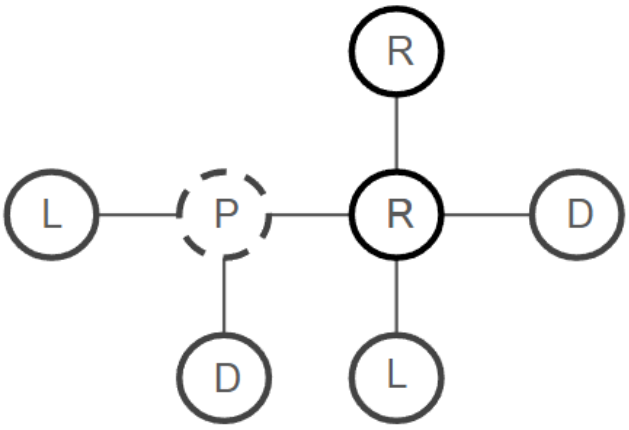


Figura 27: Representación de un laberinto, indicando las habitaciones a la izquierda (L), abajo (D) y derecha (R) de cada nodos, partiendo desde el nodo P. Configuración [7, 0, 0, 3.0, 0]

- **Verificación de Requisitos**: Una vez generada la estructura inicial, se verifica que todos los nodos cumplan con los criterios de conectividad y distribución especificados. Se

asegura que no existan habitaciones inaccesibles y que la estructura general del grafo mantenga la coherencia necesaria para la jugabilidad.

Este ejemplo ilustra cómo un conjunto de requerimientos claramente definidos puede guiar el proceso de generación procedural de mazmorras, resultando en una estructura de nivel que es a la vez coherente y jugable.

5.3.2. Población Inicial

La población inicial de la mazmorra se genera mediante un proceso iterativo que comienza con la creación de un nodo raíz. Desde este nodo raíz, se seleccionan sucesivamente nodos no utilizados para expandir la estructura de la mazmorra. Cada nodo seleccionado se convierte en un padre potencial para nuevas habitaciones, y se determina aleatoriamente cuántas habitaciones hijas se añadirán a este nodo. Estas habitaciones hijas se agregan a la lista de nodos disponibles para su uso posterior en la expansión del mapa de la mazmorra.

El algoritmo de generación del mapa inicial, mostrado en el pseudocódigo del algoritmo 2, describe este proceso en detalle. Comienza creando un nodo inicial y agregándole a la lista de nodos disponibles. Luego, mientras el número de habitaciones generadas sea menor o igual al doble del tamaño de la habitación especificada, se selecciona aleatoriamente un nodo de la lista de nodos disponibles. Se elige al azar un conjunto de direcciones (izquierda, abajo, arriba) para agregar nuevas habitaciones, lo que permite la expansión multidireccional de la mazmorra.

Para garantizar la diversidad en la estructura de la mazmorra, se barajan las direcciones antes de seleccionar aleatoriamente cuántas habitaciones hijas se agregaran en cada iteración. Luego, para cada dirección seleccionada, se crea una nueva habitación que se adjunta como hijo del nodo actualmente seleccionado. Estas nuevas habitaciones se añaden a la lista de nodos disponibles para futuras expansiones.

Algorithm 2 Generate Initial Map

```
1:  procedure GenerateMap(RoomSize)
2:      InitialNode = new Node()
3:      availableNodes.Add(InitialNode)
4:      while GeneratedRooms ≤ (RoomSize * 2) do
5:          CurrentNode = SelectRandom(availableNodes)
6:          RoomsToAdd = [Left, Down, Up]
7:          RoomsToAdd.shuffle()
8:          AmountToAdd = Random(3)
9:          while count ≤ AmountToAdd do
10:             NewRoom = new Node()
11:             NewRoom.AssignParent(RoomsToAdd[count])
12:             availableNodes.Add(NewRoom)
13:             count = count + 1
14:         end while
15:     end while
16:     return Self
17: end procedure
```

5.3.2.1. Operadores Genéticos de la fase 1

En la evolución de los niveles, se utilizan tres operadores genéticos, cada uno con su propio objetivo dentro de la generación de los niveles:

1. **Crossover:** Este operador combina ramas del grafo de forma aleatoria y luego las intercambia, con la opción de realizar un movimiento simétrico.
2. **Mutación de adición de habitación:** Este operador añade una nueva habitación a un nodo seleccionado aleatoriamente, actualizando los datos del nivel.
3. **Mutación de eliminación de habitación:** Este operador selecciona una habitación hoja y la elimina del grafo, actualizando los datos del nivel.

5.3.2.1.1. Crossover

El cruce es un paso fundamental en la evolución de las mazmorras, donde se combinan características de dos mazmorras padres para producir una nueva mazmorra descendiente. Este proceso se realiza seleccionando subgrafos de los dos padres y combinándolos para crear una nueva estructura que herede las características de ambos progenitores. El objetivo es

generar variabilidad y explorar diferentes combinaciones de diseño de mazmorras para mejorar la diversidad y calidad del conjunto de mazmorras generadas.

El algoritmo de cruce, como se muestra en el pseudocódigo del Algoritmo 3, comienza seleccionando aleatoriamente secciones de los dos padres que no sean el nodo inicial. Estas secciones seleccionadas se denominan “cortes de habitación” y servirán como puntos de intercambio entre los padres. Luego, los nodos hijo de cada padre, en las áreas de corte se intercambian, lo que permite que las características de cada padre se mezclen en la descendencia.

Es importante tener en cuenta que durante el cruce, puede ocurrir que los nodos de una mazmorra se superpongan con habitaciones de la otra mazmorra. Cuando esto sucede, se elimina la sección asociada al agregar, asegurando que la nueva mazmorra sea coherente y no contenga traslapes no deseados. El proceso de cruce se completa con la actualización de los datos de los padres.

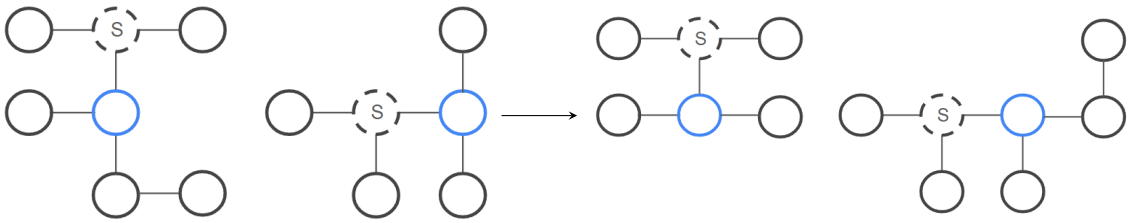


Figura 28: Crossover, los padres se encuentran a la izquierda generando las mazmorras hijas mostradas a la derecha

Algorithm 3 Crossover Algorithm

procedure Crossover($ParentA$, $ParentB$)
    $roomCutA \leftarrow GetRoom(1, \text{size}(ParentA))$
    $roomCutB \leftarrow GetRoom(1, \text{size}(ParentB))$
    $ChangeChildren(roomCutA, roomCutB)$
    $ChangeParents(roomCutA, roomCutB)$
    $ParentA$.RemoveFromGrid($roomCutA$)
    $ParentB$.RemoveFromGrid($roomCutB$)
    $useSym \leftarrow \text{Prob} < Symmetric\_move$
    $UpdateCut(roomCutA, ParentB, useSym)$
    $UpdateCut(roomCutB, ParentA, useSym)$
    $UpdateData(ParentA)$
    $UpdateData(ParentB)$
    if $useSym$ **then**
        $SymmetricMove(roomCutA)$
        $SymmetricMove(roomCutB)$
    end if
    return $ParentA, ParentB$
end procedure

5.3.2.1.2. Mutación de Simetría

La mutación de simetría aplica movimientos simétricos a cortes de habitación seleccionados, permitiendo explorar configuraciones de mazmorras que conservan ciertos aspectos simétricos. Antes del cruce, se determina si la mutación de simetría se realizará. En caso afirmativo, se aplica un movimiento simétrico horizontal a los nodos seleccionados, intercambiando los nodos asignados a la izquierda como nodos de derecha, y viceversa, para cada nodo dentro de la sección a intercambiar. Esto amplía el espacio de búsqueda, generando más combinaciones sin necesidad de aumentar el número de individuos.

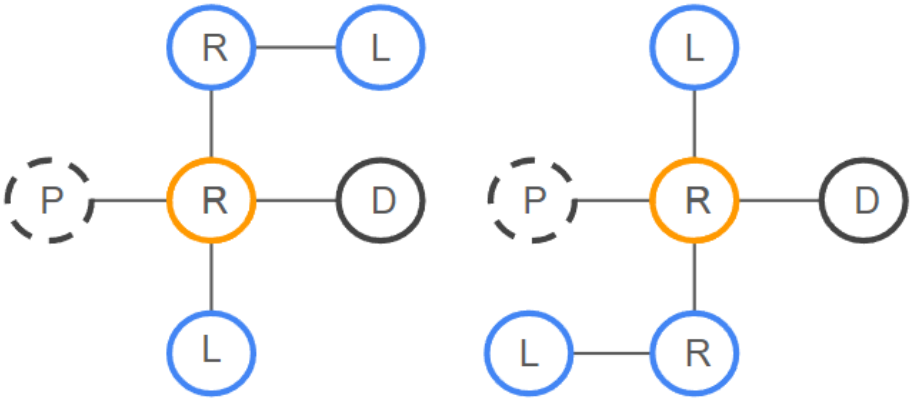


Figura 29: Ejemplo de movimiento simétrico, desde el nodo padre P, utilizando la simetría entre izquierda y derecha de los nodos hijo

5.3.2.1.3. Mutación de adición de habitación

La mutación de adición de habitación es un mecanismo que introduce variaciones en una mazmorra individual para mejorar su diversidad y calidad. Este proceso se centra en la inserción de nuevos nodos en la mazmorra, lo que puede conducir a cambios significativos en la estructura y la jugabilidad del nivel generado.

El algoritmo de mutación se denomina Add Node, como se muestra en el pseudocódigo del algoritmo 4, comienza seleccionando aleatoriamente un nodo dentro de la mazmorra. A partir de este nodo seleccionado, se exploran las posibles direcciones de movimiento (izquierda, derecha o abajo) para encontrar un nodo con la posibilidad de agregar hijos. Este proceso de exploración se realiza de manera aleatoria.

Una vez que se selecciona un nodo con la posibilidad de agregar hijos, se evalúa si agregar un nuevo nodo, como nodo hijo de éste entre los espacios aún no ocupados mejora el coeficiente lineal de la mazmorra y no cause un traslape. Este coeficiente se refiere a una métrica que puede representar diferentes aspectos de la mazmorra, como la conectividad, la complejidad o la jugabilidad. Si la adición del nuevo nodo mejora este coeficiente, se procede con la inserción del nodo en la mazmorra; de lo contrario, se continúa explorando otros nodos dentro de la mazmorra.

Algorithm 4 Add Node Mutation Algorithm

```
1:  procedure MutationAddNode(Dungeon)
2:      CurrentNode ← SelectRandom(Dungeon.Nodes)
3:      while True do
4:          AvailableMoves ← GetAvailableMoves(CurrentNode)
5:          if AvailableMoves ≠ ∅ then
6:              Move ← SelectRandom(AvailableMoves)
7:              if Rand() < Mutation_Prob then
8:                  NewNode ← CreateNode()
9:                  if ImprovesLinearCoefficient(Dungeon, NewNode, Move) then
10:                     Dungeon.AddNode(CurrentNode, NewNode, Move)
11:                 else
12:                     Continue
13:                 end if
14:             end if
15:             CurrentNode ← MoveToNextNode(CurrentNode, Move)
16:         else
17:             break
18:         end if
19:     end while
20:     return Dungeon
21: end procedure
```

Es importante destacar que la operación de mutación Add Node puede ser costosa, especialmente en mazmorras grandes, ya que implica recorrer la estructura de la mazmorra para identificar nodos adecuados donde insertar nuevos nodos. Sin embargo, esta complejidad es fundamental para garantizar que las mutaciones conduzcan a mejoras significativas en la calidad y diversidad de las mazmorras generadas. Esta mutación de explotación busca siempre mejorar la mazmorra; por lo tanto, antes de agregar un nodo, se verifica si la mazmorra necesita más habitaciones y luego se evalúa si la adición de la habitación mejora el coeficiente lineal de la mazmorra, acercándose a la configuración deseada.

La Figura 30 ilustra un ejemplo de mutación de adición de habitación denominada Add Node, donde un nuevo nodo (marcado en azúl) se agrega a la mazmorra inicial para introducir variaciones en la estructura del nivel.

5.3.2.1.4. Mutación de eliminación de habitación

La mutación de eliminación de habitación denominada Remove Node es otro mecanismo en la generación de mazmorras que introduce variaciones en una mazmorra individual, al eliminar nodos específicos de la estructura. Este proceso busca mejorar la calidad y la diversidad de las mazmorras al eliminar nodos que pueden ser redundantes o no contribuir significati-

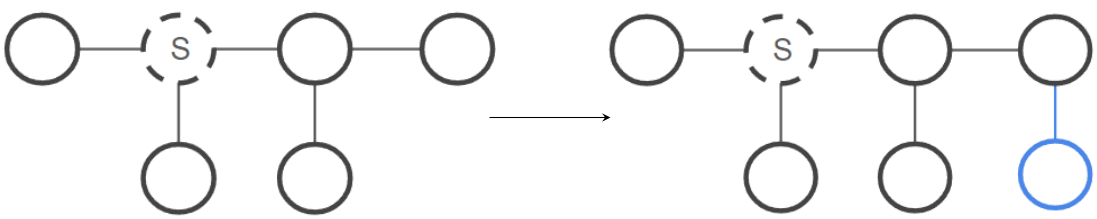


Figura 30: Mutación de adición de habitación, a la izquierda se presenta el laberinto inicial, al cual se le agrega un nodo marcado en azúl

vamente a la jugabilidad del nivel.

El algoritmo de mutación Remove Node, como se muestra en el pseudocódigo del Algoritmo 5, comienza seleccionando aleatoriamente un nodo dentro de la mazmorra. A partir de este nodo seleccionado, el algoritmo se mueve a través de la estructura de la mazmorra hasta encontrar una hoja del grafo. Las hojas del grafo son nodos que no tienen hijos y, por lo tanto, pueden eliminarse sin afectar la conectividad de la mazmorra. Esta decisión se toma de forma aleatoria, controlada por un parámetro de probabilidad. Si la probabilidad de eliminar el nodo es satisfactoria, se elimina el nodo de la mazmorra. Sin embargo, antes de realizar la eliminación, se verifica si la eliminación del nodo reduce el exceso de habitaciones en la mazmorra. El exceso de habitaciones se refiere a la presencia de nodos redundantes o no esenciales que pueden afectar negativamente la jugabilidad o la estética del nivel. Si la eliminación del nodo contribuye a reducir el exceso de habitaciones, se procede con la eliminación del nodo.

Es importante destacar que la mutación denominada Remove Node solo se aplica cuando la mazmorra tiene un exceso de habitaciones, es decir, cuando existen nodos que pueden eliminarse sin comprometer su integridad estructural. Esta mutación busca optimizar la estructura de la mazmorra. El exceso de habitaciones suele ser resultado del cruce, por lo que esta mutación contribuye a mejorar los resultados de dicho proceso y la calidad de las mazmorras generadas.

La figura 31 ilustra un ejemplo de Mutación de eliminación de habitación denominada Remove Node, donde un nodo superior (marcado en azúl) se elimina de la mazmorra inicial para introducir variaciones en la estructura del nivel.

Algorithm 5 Remove Node Mutation Algorithm

```
1:  procedure MutationRemoveNode(Dungeon)
2:      CurrentNode ← SelectRandom(Dungeon.Nodes)
3:      while True do
4:          if IsLeaf(CurrentNode) then
5:              if Rand() < Mutation_Prob then
6:                  Dungeon.RemoveNode(CurrentNode)
7:                  if ReducesExcessRooms(Dungeon) then
8:                      return Dungeon
9:                  end if
10:             end if
11:             break
12:         else
13:             CurrentNode ← SelectRandom(CurrentNode.Children)
14:         end if
15:     end while
16:     return Dungeon
17: end procedure
```

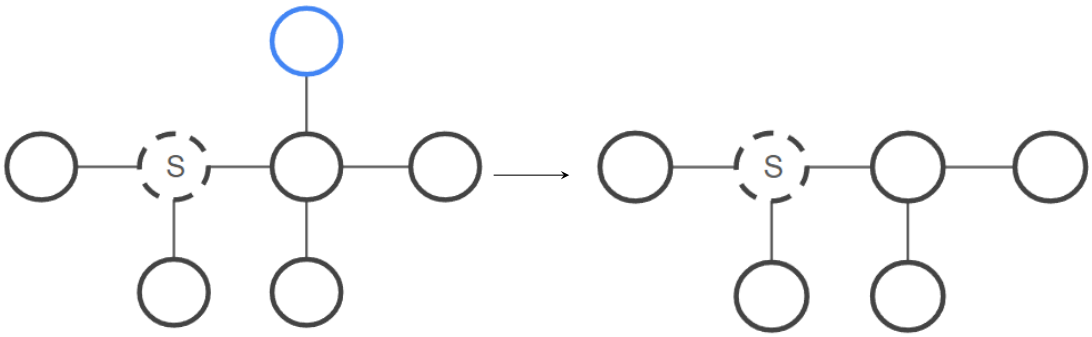


Figura 31: Mutación de eliminación de habitación, a la izquierda se presenta el laberinto inicial, al cual se le remueve un nodo superior demarcado de color azúl

El Algoritmo 6 describe el proceso de evolución para la generación estructural basado en una población inicial de individuos, utilizando mecanismos de selección por toreneo, cruce y mutación para generar nuevas generaciones hasta alcanzar un número predefinido de generaciones.

- En cada generación, el mejor individuo (*mejor_ind*) es seleccionado y guardado.
- Se realiza una seleccion de los individuos a cruzar mediente una selección por torneo, obteniendo dos individuos padres (*PadreA* y *PadreB*) para seleccionar a los individuos que serán cruzados o mutados.

- Si la probabilidad de cruce es satisfactoria, se seleccionan cortes en las habitaciones de ambos padres, se intercambian y se actualizan los datos relacionados.
- Las mutaciones incluyen la eliminación de habitaciones (si el tamaño del padre es mayor que uno) y la adición de nuevas habitaciones, siempre actualizando los datos correspondientes.
- Al final de cada iteración, se aplica elitismo

El proceso continúa hasta completar el número especificado de generaciones.

Algorithm 6 Generación Estructural

procedure EvolveForm($generation$)
  while $generation \leq Generations\_Form$ **do**
    $best\_ind = GetBest(generation)$
    $newGeneration = []$
    for $i = 0$ to popsize/2 **do**
      Tournament(generation, ParentA, ParentB)
      if $Prob < CrossOver\_Form$ and $size(ParentA) > 1$ and $size(ParentB) > 1$ **then**
        $roomCutA = GetRoom(1, size(ParentA))$
        $roomCutB = GetRoom(1, size(ParentB))$
        $ChangeChildren(roomCutA, roomCutB)$
        $ChangeParents(roomCutA, roomCutB)$
        $ParentA.RemoveFromGrid(roomCutA)$
        $ParentB.RemoveFromGrid(roomCutB)$
        $useSym = Prob < Symmetric_move$
        $UpdateCut(roomCutA, ParentB, useSym)$
        $UpdateCut(roomCutB, ParentA, useSym)$
        $UpdateData(ParentA)$
        $UpdateData(ParentB)$
      end if
      if $Prob < Mutation\_Remove\_Room$ and $size(Parent) > 1$ **then**
        $SelectLeafRoom(Parent)$
        $RemoveLeaf(Parent)$
        $UpdateData(Parent)$
      end if
      if $Prob < Mutation\_Add\_Room$ **then**
        $room = GetRoom(1, size(ParentA))$
        $AddRoom(room)$
        $UpdateData(Parent)$
      end if
      $newGeneration.Add(ParentA, ParentB)$
    end for
    $newGeneration[0] = best\_ind$
    $generation = NewGeneration$
  end while
  return $generation$
end procedure

5.3.3. Fase 2: Alocación de llaves y Barreras

La **fase de evolución de distribución llave-barrera** ajusta la ubicación de las llaves y barreras dentro de la estructura predefinida. Esta fase garantiza que todas las restricciones relacionadas con la jugabilidad, como la accesibilidad de las llaves y barreras, se cumplan correctamente. Esta etapa es crucial para asegurar que la mazmorra no solo sea estructuralmente sólida, sino también jugable y desafiante, proporcionando una experiencia equilibrada para los jugadores.

- **Distribución Inicial**: Se distribuyen llaves y barreras de manera inicial en la estructura generada, respetando las reglas básicas de colocación para asegurar una jugabilidad mínima.
- **Validación de distribución**: Se evalúan las configuraciones de acuerdo con criterios de jugabilidad, como la accesibilidad y la progresión lógica a través de la mazmorra.
- **transformación**: Se aplican operadores evolutivos para ajustar y optimizar la ubicación de llaves y barreras, asegurando que cada elemento esté colocado estratégicamente para maximizar el desafío y la jugabilidad.
- **Iteración**: Este proceso se repite hasta que se alcanza una distribución óptima que cumpla con todos los requisitos de jugabilidad y diseño.

Primero se realiza una distribución inicial de llaves y barreras de manera inicial, asegurando que exista una ruta posible. Para luego, evolucionar esta distribución inicial utilizando el algoritmo denominado EvolveBK. Este algoritmo selecciona a los mejores individuos de la generación actual y los cruza para producir descendencia que mejore la distribución de llaves y barreras en el nivel. También se aplican operadores de mutación, como la eliminación y adición de llaves y barreras, para introducir variabilidad y explorar nuevas soluciones. Esto se puede denotar en el Algoritmo 7.

Algorithm 7 Evolve BK Algorithm

```
1:  procedure EvolveBK(generation)
2:      generation = AddBK(generation)
3:      while generation ≤ Generations_Form do
4:          best_ind = GetBest(generation)
5:          for i = 0 to popsize/2 do
6:              Tournament(generation, ParentA, ParentB)
7:              // MutationRemoveBK(Parent)
8:              if Prob < Mutation_Remove_BK and BKQuantity(Parent) ≥ 1 then
9:                  SelectBKPair(Parent)
10:                 RemoveBK(Parent)
11:                 UpdateData(Parent)
12:             end if
13:             // MutationAddBK(Parent)
14:             if Prob < Mutation_Add_Room then
15:                 toVisit = GetChildren(Parent.firstRoom)
16:                 currentType = Key
17:                 while toVisit ≠ Empty do
18:                     toVisit.Shuffle()
19:                     room = toVisit[0]
20:                     if room.type = Normal and prob(100) ≤ 30 then
21:                         room.changeType(currentType)
22:                         if currentType = Barrier then
23:                             break
24:                         else
25:                             currentType = Barrier
26:                         end if
27:                     end if
28:                     toVisit.Add(GetChildren(room))
29:                 end while
30:             end if
31:         end for
32:     end while
33:     return generation
34: end procedure
```

5.3.3.1. Distribución Inicial de Llaves y Barreras

En esta etapa, cada mapa generado en la fase 1 experimenta una distribución inicial de llaves y barreras. Este proceso es esencial para crear niveles desafiantes y bien estructurados, donde el jugador debe recoger llaves para acceder a áreas bloqueadas por barreras.

Proceso de asignación de Llaves y Barreras: El algoritmo de distribución inicial, como se muestra en el Algoritmo 8, comienza seleccionando todos los nodos hijos del nodo inicial del mapa, excluyendo el nodo inicial mismo. Estos nodos hijos se consideran habitaciones potenciales para la asignación de llaves y barreras. Luego, el algoritmo procede a asignar llaves y barreras a estas habitaciones de manera aleatoria. Para cada habitación seleccionada, se determina aleatoriamente si se le asignará una llave o una barrera. Esta asignación aleatoria se realiza con cierta probabilidad, lo que permite introducir variabilidad en la distribución de llaves y barreras en el nivel.

Asignación de Llaves: Si se está asignando una llave y la probabilidad de asignación es satisfecha, se asigna una llave a la habitación seleccionada. Esto significa que el jugador deberá recoger esta llave para desbloquear áreas posteriores del nivel.

Asignación de Barreras: Si se está asignando una barrera y la probabilidad de asignación es satisfecha, se asigna una barrera a la conexión entre la habitación seleccionada y su nodo padre. Esto significa que el acceso a la habitación desde su nodo padre estará bloqueado hasta que el jugador obtenga la llave correspondiente.

Iteración a Través del Mapa: El algoritmo continúa este proceso iterativo hasta que se asignen todas las barreras requeridas por el diseñador. Durante cada iteración, se seleccionan nuevas habitaciones y se les asignan llaves o barreras según corresponda.

Resultados de la Distribución Inicial: Al finalizar el algoritmo, el mapa tendrá una distribución inicial de llaves y barreras que crea desafíos y obstáculos para el jugador. Esta distribución sienta las bases para la evolución posterior de llaves y barreras en la fase 2 del algoritmo.

Algorithm 8 Distribución Inicia

```
1:  procedure InitialDistribution(Map)
2:      aviableRooms.append(GetSons(Map.InitialNode))
3:      while GeneratedBarriers ≤ DesignerBarriers do
4:          CurrentNode = SelectRandom(aviableRooms)
5:          UseSelected = (Random(100) ≤ 30)
6:          if AssigningKey and UseSelected then
7:              AssignKey(CurrentNode)
8:          end if
9:          if ¬AssigningKey and UseSelected then
10:             AssignBarrier(CurrentNode)
11:             GeneratedBarriers ← GeneratedBarriers + 1
12:         end if
13:         aviableRooms.append(GetSons(CurrentNode))
14:     end while
15:     return Map
16: end procedure
```

5.3.3.2. Validación de Distribución

En esta etapa crítica del proceso de generación de niveles, se asegura que la distribución de llaves y barreras permita al jugador completar la mazmorra sin dejar áreas inaccesibles. Se detectan problemas de accesibilidad, se realizan ajustes en la distribución hasta obtener una configuración válida. La falta de atención a estos casos podría resultar en mapas como se muestra en la Figura 32, donde se requiere obtener llaves antes de poder abrir las barreras que contienen las llaves necesarias para avanzar, teniendo con un mapa incompleto.

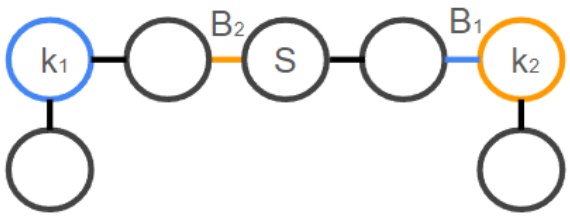


Figura 32: Ejemplo de un laberinto imposible de pasar dada la distribución de llaves y barreras

Para verificar adecuadamente que las llaves y las barreras estén asignadas de manera correcta, se aplica un algoritmo A* qué busca dentro del mapa la salida entre las habitaciones disponibles. Durante este proceso, el algoritmo libera los espacios bloqueados por barreras

una vez que se encuentra la llave asociada, permitiendo al jugador avanzar de manera fluida por el nivel. Además, este algoritmo proporciona el valor de llaves necesarias para completar el nivel con éxito.

5.3.4. Transformación

En la evolución de los niveles, se utilizan dos operadores genéticos, cada uno con su propio objetivo dentro de la generación de los niveles:

1. **Mutación de eliminación de llave-barrera:** Este operador tiene como objetivo mejorar la calidad y la jugabilidad del nivel al eliminar selectivamente llaves y barreras, este es un proceso de exploración, dado que elimina si existen combinaciones de llaves y barreras, sin importar cómo afectan al mapa
2. **Mutación de adición de llave-barrera:** Contrario al operador anterior, este operador se encarga de introducir variabilidad y complejidad en el nivel mediante la adición selectiva de llaves y barreras. Su objetivo es mejorar la calidad de los niveles, por lo que no se aplicará si el nivel ya cumple con los requisitos de diseño.

5.3.4.1. Mutación de eliminación de llave-barrera

La mutación de eliminación de llaves denominada Remove Key se centra en la exploración y eliminación de una llave dentro de la mazmorra. Este proceso comienza seleccionando un nodo aleatorio dentro del mapa y explorando todas las habitaciones a partir de este nodo. Dado que no se consideran ciclos en la exploración, el objetivo es revisar todas las habitaciones del nivel. Una vez que se encuentra una llave, se continúa explorando el resto del nivel desde ese punto para eliminar las barreras asociadas a esa llave. Es importante destacar que esta mutación puede conducir a la generación de mazmorras con barreras, pero sin las llaves asociadas, lo cual puede generar problemas de accesibilidad en el nivel.

El Algoritmo 9 describe este proceso de eliminación de llaves. Se inicia seleccionando un nodo aleatorio como nodo inicial para comenzar la exploración. Luego, se crea una lista de nodos por visitar y se inicia esta lista con el nodo inicial seleccionado. A continuación, se continuará mientras existan nodos por visitar en la lista. En cada iteración, se saca un nodo de la lista y se verifica si es una llave. Si se encuentra una llave, se cambia a una habitación normal y se eliminan las barreras asociadas a esa llave mediante la función `RemoveAssociatedBarriers`. Una vez que se elimina la llave y sus barreras asociadas, se termina el bucle. Si el nodo no es una llave, se expande la exploración agregando sus nodos hijos a la lista de nodos por visitar. Este proceso continúa hasta que se elimina una llave o se exploran todas las habitaciones del nivel.

La Figura 33 ilustra un ejemplo de esta mutación, donde se muestra cómo se elimina una llave y la barrera asociada a esa llave en la mazmorra. Esto ejemplifica cómo esta mutación puede afectar la estructura del mapa, eliminando elementos clave y ajustando la accesibilidad del nivel.

Algorithm 9 MutationRemoveKey

```
1:  procedure MutationRemoveKey(Map)
2:      startNode = SelectRandom(Map.Nodes)
3:      toVisit ← [startNode]
4:      while toVisit ≠ Empty do
5:          currentNode = toVisit.pop()
6:          if currentNode.type = "Key" then
7:              currentNode.type ← "Normal"
8:              RemoveAssociatedBarriers(currentNode)
9:              break
10:         end if
11:         toVisit.extend(GetChildren(currentNode))
12:     end while
13:     return Map
14: end procedure
```

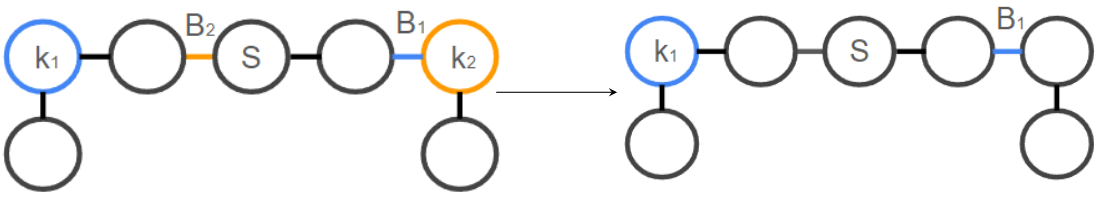


Figura 33: Ejemplo de la mutación de eliminación de llaves, el cual elimina la barrera 1 y la llave 1

5.3.4.2. Mutación de adición de llave-barrera

La mutación de adición de llave-barrera denominada Add Key es una mutación de explotación está enfocada en agregar un par de llaves y barrera en la mazmorra. Este proceso comienza seleccionando un nodo al azar dentro del mapa y moviéndose a través de la mazmorra a partir de esta ubicación. El objetivo es buscar nodos que no tengan llaves ni barreras. Si se encuentra una habitación vacía, se le asigna una llave de manera aleatoria. Luego, el algoritmo sigue explorando el nivel hasta encontrar un nodo vacío donde intentar colocar una barrera. Si se coloca una barrera, el algoritmo termina; de lo contrario, puede generar un exceso de llaves en el mapa, que no abren barreras específicas.

El Algoritmo 10 describe este proceso de adición de llave y barrera. Se inicia seleccionando un nodo aleatorio como nodo inicial para comenzar la exploración. Luego, se crea una lista de nodos por visitar y se inicia esta lista con el nodo inicial seleccionado. Durante la exploración, se verifica si el nodo actual es una habitación normal y si un número aleatorio cumple cierta condición. Si ambas condiciones se cumplen, se asigna una llave a esa habitación y se marca que se ha colocado una llave. A continuación, se continúa la exploración del nivel hasta encontrar un nodo normal donde intentar colocar una barrera. Si se coloca una barrera, el algoritmo termina; de lo contrario, se continúa explorando el nivel en busca de un lugar adecuado para colocar la barrera.

Algorithm 10 MutationAddKey

```
1:  procedure MutationAddKey(Map)
2:      startNode = SelectRandom(Map.Nodes)
3:      toVisit ← [startNode]
4:      keyPlaced ← False
5:      while toVisit ≠ Empty do
6:          currentNode = toVisit.pop()
7:          if currentNode.type = ´´Normal" and prob(100) ≤ 30 then
8:              currentNode.type ← ´´Key"
9:              keyPlaced ← True
10:         end if
11:         if keyPlaced then
12:             barrierPlaced ← False
13:             toVisitBarrier ← [currentNode]
14:             while toVisitBarrier ≠ Empty and not barrierPlaced do
15:                 barrierNode = toVisitBarrier.pop()
16:                 if barrierNode.type = "Normal" then
17:                     barrierNode.type ← "Barrier"
18:                     barrierPlaced ← True
19:                 end if
20:                 toVisitBarrier.extend(GetChildren(barrierNode))
21:             end while
22:             if barrierPlaced then
23:                 break
24:             end if
25:         end if
26:         toVisit.extend(GetChildren(currentNode))
27:     end while
28:     return Map
29: end procedure
```

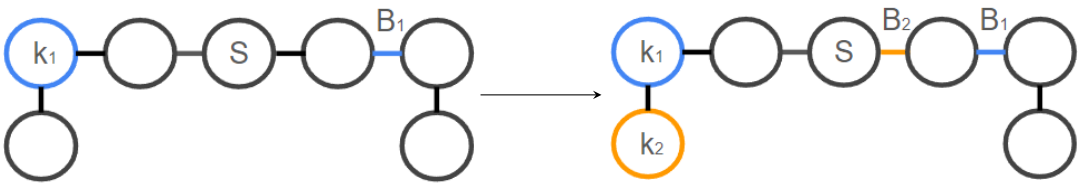


Figura 34: Ejemplo de la mutación de adición de llave-barrera, se agrega una nueva combinación llave y barrera dentro del laberinto

5.4. Conclusiones

En este capítulo, se ha presentado un algoritmo evolutivo de dos pasos con simetría horizontal para la generación procedural de mazmorras. La representación de las mazmorras como grafos ha demostrado ser una metodología eficaz para asegurar una estructura bien definida y evitar superposiciones de habitaciones.

Los puntos clave de este capítulo incluyen:

- **Representación basada en grafos:** La utilización de grafos para representar las mazmorras permite una organización clara de habitaciones y conexiones, facilitando la implementación de restricciones espaciales y topológicas.
- **Algoritmo evolutivo de dos fases:** La división del proceso en una fase de estructural y una fase de alocación de llaves y barreras, ha mostrado ser eficaz para generar mazmorras con una buena conectividad y simetría, mejorando la calidad de las soluciones iniciales a través de técnicas evolutivas.

En el siguiente capítulo, se presenta la evaluación del algoritmo propuesto y su comparación con otros métodos. Se describirá la configuración experimental y se presentarán los resultados obtenidos en términos de calidad de las mazmorras generadas, eficiencia computacional y comparación de niveles generados.

CAPÍTULO 6
Evaluación y Comparación

En este capítulo se evaluará el rendimiento del aloritmo propuesto (2-Step EA) para la generación de mazmorras y se compara los resultados de [Pereira *et al.*, 2021]. El objetivo es analizar y comparar las capacidades de generación de mazmorras de ambos algoritmos y el generador de referencia bajo configuraciones similares. Para garantizar una comparación justa y precisa, se replico el generador de referencia, modificándolo para producir mazmorras idénticas para una semilla dada. Este enfoque permite comparar directamente los resultados de ambos algoritmos utilizando la misma semilla, asegurando así la consistencia en las condiciones experimentales[1].

6.1. Métricas de Evaluación

Para evaluar la efectividad del algoritmo, se utilizaron las siguientes métricas:

1. **Tiempo de Generación**: El tiempo requerido por el algoritmo para generar una mazmorra completa, desde la inicialización hasta la obtención de la solución final. Además de medir el tiempo total requerido, es importante analizar las diferentes fases del proceso de generación para identificar posibles cuellos de botella y optimizar el rendimiento del algoritmo.

2. **Satisfacción de las Restricciones**: La medida en que las mazmorras generadas cumplen con las restricciones impuestas por los parámetros de entrada, asi como el número de habitaciones, llaves necesarias, y la disposición lineal. Se puede evaluar qué tan bien cumplen las mazmorras generadas las restricciones definidas por los parámetros de entrada. Por ejemplo, se puede comparar el número de habitaciones generadas con el número objetivo especificado por el diseñador, verificar si se han colocado las llaves necesarias en ubicaciones estratégicas para desbloquear las barreras, y asegurarse de que la distribución lineal del nivel cumpla con los requisitos establecidos.

3. **Estudio de Soluciones**: Evaluada mediante el análisis de los mapas generados por el algoritmo con distintas semillas, para observar su comportamiento y los límites en la generación de múltiples niveles. Analizar los mapas generados por el algoritmo con diferentes semillas permite observar la diversidad y la variabilidad de las soluciones. Esto incluye la complejidad de los diseños, la cohesión temática de las mazmorras, y la capacidad del algoritmo para generar niveles interesantes y jugables bajo diferentes condiciones.

[1] La implementación de ambos algoritmos y los experimentos se pueden encontrar en `https://github.com/FelipeDumont/PDGwB`

6.1.1. Resultados de la Evaluación

A continuación, se presenta un ejemplo de una mazmorra generada utilizando el algoritmo 2-STEP EA, mostrando la distribución de habitaciones, llaves y barreras para una configuración específica (ver Figura 35).

Este mapa contempla 15 habitaciones, la habitación inicial es (0,0) asignada como S, la Salida se encuentra en la habitación señalada como E. Para salir de esta mazmorra se debe ir a buscar la llave en la habitación 1. En el camino a la barrera que se abre por la llave 1, se encuentra con la llave 9, lo que da la opción de salir de la mazmorra.

A continuación, se presenta la descripción detallada del mapa generado:

- **Habitaciones**: La mazmorra está compuesta por 15 habitaciones dispuestas de manera que cada una es accesible desde al menos una habitación. La disposición asegura que el jugador siempre tenga un camino claro y lógico para seguir.
- **Llave 1**: Ubicada en la habitación 1. Esta llave es necesaria para abrir la primera barrera que bloquea el acceso a la salida.
- **Llave 9**: Ubicada en el camino hacia la barrera de la llave 1. Esta llave ofrece una opción adicional para el jugador, proporcionando acceso a rutas alternativas o recursos adicionales dentro de la mazmorra.
- **Barrera**: La barrera correspondiente a la llave 1 está posicionada de manera que el jugador deba regresar a la habitación inicial después de obtener la llave 1 y antes de continuar hacia la salida.

El camino crítico para completar este nivel de mazmorra, es decir, la secuencia mínima de pasos necesarios para llegar a la salida, es el siguiente: [S, 1, S, 2, 5, 9, 10, E]. Este camino asegura que el jugador recoja todas las llaves necesarias y utilice las barreras de manera efectiva para progresar.

Aunque el algoritmo 2-STEP EA no evalúa explícitamente el camino crítico, los resultados indican que la generación de mazmorras produce diseños jugables y desafiantes que cumplen con los criterios establecidos para la coherencia y la jugabilidad.

Los detalles adicionales de la evaluación incluirán:

- **Calidad de las Mazmorras Generadas**: Se comparará la calidad de las mazmorras generadas por el algoritmo 2-STEP EA con las generadas por otros enfoques, como el

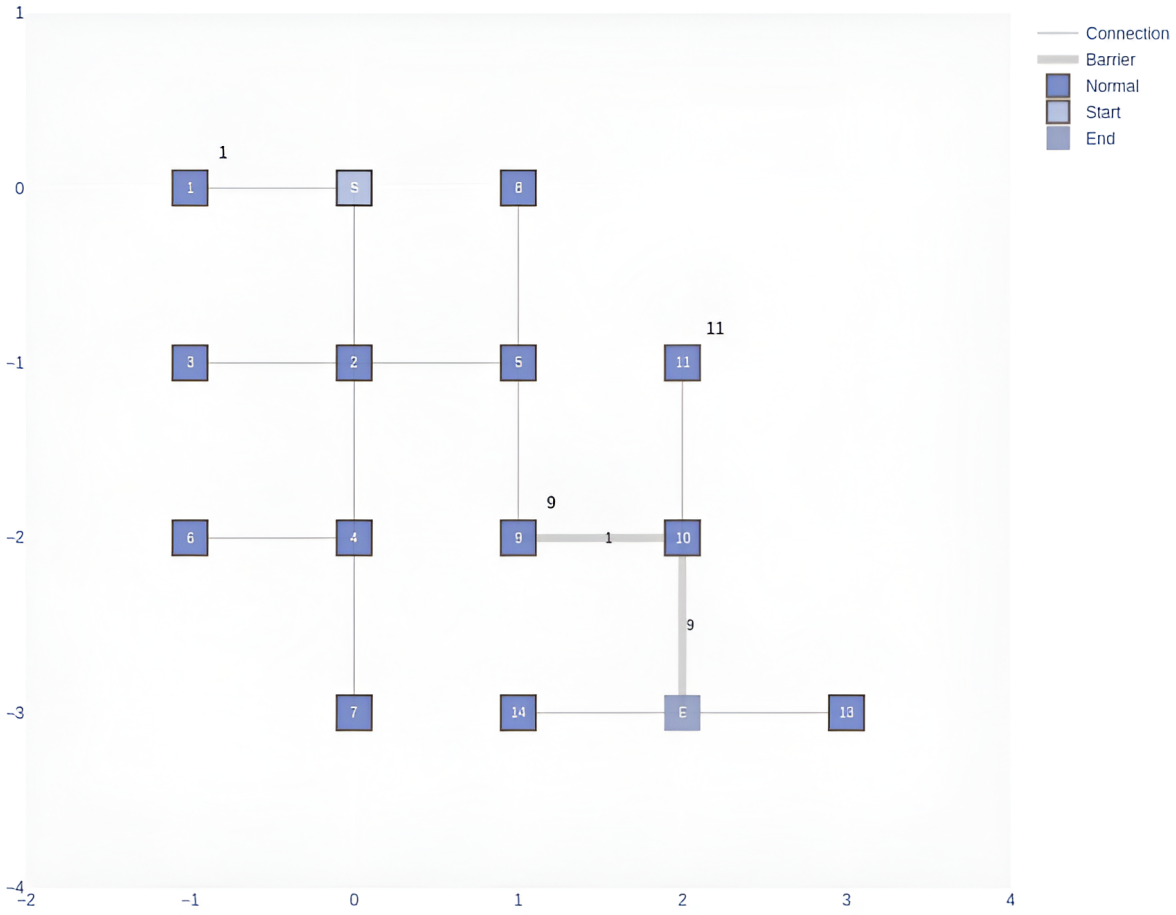


Figura 35: Ejemplo de una mazmorra generada utilizando el algoritmo 2-STEP EA, mostrando la distribución de habitaciones, llaves y barreras para un mapa con la siguiente configuración: [15,3,2,2.0,2]

CEA. La calidad de una mazmorra se mide mediante la evaluación de parámetros específicos de generación asignados por un diseñador, como el número de habitaciones, llaves, barreras, coeficiente lineal y llaves necesarias.

- **Eficiencia Computacional**: Se analizará la eficiencia computacional del algoritmo en términos de tiempo de generación, comparándolo con otros enfoques para determinar su escalabilidad y rendimiento en diferentes configuraciones de mazmorras.
- **Comparación del mapa de calor de Niveles Generados**: Se realizará una comparación visual de la distribución de elementos clave en las mazmorras generadas por el algoritmo 2-STEP EA y otros enfoques utilizando mapas de calor, para identificar diferencias en el diseño de las mazmorras.

Estas pruebas proporcionarán una evaluación exhaustiva de la efectividad y eficiencia del algoritmo 2-STEP EA en la generación de mazmorras para su aplicación en el diseño de videojuegos y la creación de contenido procedural.

A continuación se presenta el algoritmo con el cual se realizarán las comparaciones.

6.2. Algoritmo Evolutivo Restringido (CEA)

El Algoritmo Evolutivo Restringido (Constrained Evolutionary Algorithm, CEA) propuesto en [Pereira *et al.*, 2021] ofrece un enfoque innovador para la generación de niveles de mazmorras mediante una representación en forma de grafo, similar al algoritmo propuesto en esta tesis.

Este algoritmo realiza una búsqueda en profundidad (DFS), que permite una exploración exhaustiva de los caminos, simulando cómo un jugador podría interactuar con el nivel generado. Nuestro análisis preliminar sugiere que tanto el CEA como el algoritmo propuesto en esta tesis producen resultados de calidad comparable en la generación de mazmorras, pero con una mejora notable en la velocidad computacional al evitar el uso de DFS. Para evaluar aspectos fundamentales en la generación de mazmorras, como la eficiencia de los algoritmos para ajustarse a los parámetros definidos por el diseñador, se utilizan versiones de los algoritmos sin DFS como base de comparación en los experimentos.

En esta sección, se evaluará el rendimiento del Algoritmo Evolutivo de Dos Pasos (2-Step EA) propuesto para la generación de mazmorras, comparándolo con el Algoritmo Evolutivo Restringido (CEA) descrito en [Pereira *et al.*, 2021]. Para garantizar una comparación justa, ambos algoritmos fueron implementados y probados bajo las mismas condiciones experimentales, utilizando la misma semilla para asegurar consistencia en los resultados.

Los experimentos se centraron en:

- La eficacia de la generación de mazmorras.
- La calidad de las mazmorras producidas.
- La eficiencia temporal de los algoritmos.

Se realizaron ajustes, como la eliminación de la Búsqueda en Profundidad (DFS) del CEA, para mejorar la eficiencia y centrarse en las capacidades fundamentales de generación de mazmorras.

6.2.1. Funcionamiento del CEA

El CEA funciona mediante la generación y evolución de una población de individuos, donde cada individuo representa una posible mazmorra. Los pasos básicos del CEA son los siguientes:

1. **Inicialización**: Se genera una población inicial de mazmorras aleatorias.
2. **Evaluación**: Cada mazmorra se evalúa utilizando una función de fitness que mide su adecuación a los parámetros especificados (habitaciones, llaves, barreras, coeficiente lineal).
3. **Selección**: Se seleccionan las mejores mazmorras de la población actual para ser padres de la siguiente generación.
4. **Crossover y Mutación**: Se aplican operadores genéticos de crossover y mutación para generar nuevos individuos (mazmorras) a partir de los padres seleccionados.
5. **Reemplazo**: La nueva generación de mazmorras reemplaza a la antigua, y el proceso se repite hasta que se cumpla un criterio de parada (por ejemplo, un número máximo de generaciones o una convergencia en la función de eveluación).

6.2.2. Ventajas del CEA sin DFS

La versión sin DFS del CEA tiene varias ventajas que motivaron su selección para nuestros experimentos:

- **Eficiencia Computacional**: La eliminación del DFS reduce significativamente el tiempo de ejecución, permitiendo una evaluación más rápida de cada generación.
- **Simplicidad**: Sin la complejidad adicional del DFS, el algoritmo es más fácil de implementar y ajustar.

- **Calidad Comparable**: En nuestros análisis preliminares, la calidad de las mazmorras generadas sin DFS fue comparable a las generadas con DFS, pero con un menor costo computacional.

Se ejecuto el algoritmo CEA sin DFS con respecto a su versión con DFS (CEA+DFS), para establecer como esto esta afectando los resultados de este algoritmo, lo cual se puede observar en la tabla 36, la cual presenta los resultados de la ejecución de ambas versiones del algoritmo 100 veces. Los resultados nos presenta que en us mayoria el algoritmo sin DFS tiene mejores resultados que el algoritmo con DFS.

	Valor de aptitud				Tiempo (s)			
	CEA		CEA+DFS		CEA		CEA+DFS	
Configuraciones	Avg.	st. dv.	Avg.	st. dv.	Avg.	st. dv.	Avg.	st. dv.
(15,3,2,2.0,2.0)	**0.45**	0.26	0.92	0.39	**3.86**	0.22	5.32	0.47
(20,4,4,1.0,4.0)	**0.25**	0.23	0.94	0.29	**5.3**	0.59	5.84	0.56
(20,4,4,2.0,4.0)	**0.68**	0.23	0.94	0.28	**4.59**	0.35	6.44	0.76
(25,8,8,1.0,8.0)	**0.58**	0.2	1.08	0.44	**5.1**	0.39	7.01	0.66
(30,4,4,2.0,4.0)	**0.25**	0.2	0.85	0.3	**4.3**	0.31	6.43	0.71
(30,6,6,1.5,6.0)	**0.05**	0.0	0.11	0.22	**4.7**	0.25	5.79	0.3
(100,20,20,1.5,20.0)	**2.87**	2.85	6.35	3.14	**5.46**	0.28	12.45	0.49
(500,100,100,1.5,100.0)	**314.34**	7.61	359.87	19.5	**7.54**	1.03	212.1	884.51

Figura 36: Metricas de comparación del Algoritmo Evolutivo Restringido (CEA) sin DFS y con DFS

6.2.3. Implementación del Algoritmo Original y Comparación entre Versiones

Para realizar una comparación justa entre algoritmos y evitar sesgos causados por diferencias en los lenguajes de programación, se emprendió la tarea de implementar el algoritmo original implementado en C# en Unity a C++. Esta implementación se realizó meticulosamente desde cero para garantizar una correspondencia precisa y sin distorsiones.

La implementación implicó una comprensión exhaustiva del algoritmo original en C# y su posterior reescritura en C++, asegurando que la lógica y el comportamiento fueran idénticos en ambas versiones. Se prestó especial atención a los detalles de la estructura de datos y los algoritmos utilizados para mantener una equivalencia funcional entre las implementaciones.

Una vez completada la implementación, surgió un desafío relacionado con el manejo de números de punto flotante, que podrían generar discrepancias en los resultados de generación de niveles. Para abordar este problema, se decidió restringir el cálculo de la función de

evaluación en ambas versiones para evitar posibles divergencias, debido a diferencias en la precisión de los números de punto flotante.

Además, para garantizar una comparación justa, se generaron archivos de valores aleatorios obtenidos por el algoritmo CEA cada vez que se solicitaba un número aleatorio. Estos archivos de valores aleatorios se utilizaron como semillas en ambas versiones del algoritmo para asegurar que la generación de niveles se basara en la misma secuencia de números aleatorios en ambas implementaciones.

Finalmente, se llevaron a cabo pruebas comparativas generando niveles de mazmorras utilizando ambas versiones del algoritmo y comparando los resultados. El objetivo fue verificar si las implementaciones en C# y C++ producían los mismos niveles de mazmorras, lo que indicaría una traducción precisa y una equivalencia funcional entre ambas versiones del algoritmo. Este enfoque riguroso garantizó que la comparación entre las versiones se llevara a cabo de manera justa y precisa, sin sesgos ni discrepancias.

6.3. Configuración Experimental

Los siguientes experimentos se realizaron en una máquina virtual con Ubuntu 22.04.1 LTS, con 10 GB de RAM DDR4 3200MHz, utilizando un procesador AMD Ryzen 7 5800H. Todos los resultados se basan en 100 ejecuciones independientes, reportando los resultados promedio para la generación de mazmorras.

6.4. Resultados de la Comparación

En esta sección se presentan algunos ejemplos de evaluación que consideran la comparación de la calidad de las mazmorras generadas y la eficiencia computacional de ambos algoritmos.

6.4.1. Calidad de las Mazmorras Generadas

Para evaluar la calidad de las mazmorras generadas, se comparan los parámetros especificados con los obtenidos en las mazmorras finales. Los parámetros considerados fueron el número de habitaciones, llaves, barreras, coeficiente lineal, y las llaves necesarias. Los resultados muestran el promedio de ejecución del algoritmo 100 veces con dististintas semillas los cuales se presentan en la Tabla 5.

Los resultados indican que 2-Step EA cumple más consistentemente con los parámetros especificados en comparación con el CEA, especialmente en términos del número exacto de

Algoritmo	Habitaciones	Llaves	Barreras	Coef. Lineal
Requerimientos	500.0	100.0	100.0	1.5
CEA	499.95	23.81	20.21	1.4
2-Step EA	498.88	100.0	99.99	1.44

Tabla 5: Comparación de la calidad de las mazmorras generadas.

habitaciones y el coeficiente lineal.

6.4.2. Eficiencia Computacional

También es evalúa la eficiencia computacional de ambos algoritmos midiendo el tiempo promedio de ejecución del algoritmo 100 veces con dististintas semillas. Los resultados se presentan como ejemplo en la Tabla 6.

Algoritmo	Tiempo de Generación (s) Problema 1	Tiempo de Generación (s) Problema 2
CEA	3.65	4.83
2-Step EA	3.37	4.41

Tabla 6: Ejemplo de comparación de la eficiencia computacional de los algoritmos.

Los resultados muestran que el 2-Step EA es más eficiente en términos de tiempo de generación en los problemas 1 y 2, lo que sugiere que nuestro enfoque mejora los tiempos de generación para los problemas 1 y 2, por lo que para este analicis requerimos analizar todos los casos disponibles para establecer conclusiones con respecto a todos los casos.

6.4.3. Comparación de mapas de calor de los niveles Generados

Para visualizar y comparar la distribución de las habitaciones generadas por ambos algoritmos, se usarán mapas de calor. Estos mapas representan la frecuencia con la que se generan habitaciones en cada posición del mapa, considerando el nodo inicial al centro de estos.

6.5. Proceso de Ajuste

El proceso de ajuste de las probabilidades de aplicación de los operadores evolutivos es crucial para garantizar el rendimiento óptimo del Algoritmo Evolutivo de Dos Pasos (2-Step EA) y para el CEA también. Para este propósito se usó la herramienta Irace [López-Ibáñez *et al.*, 2016], que es ampliamente utilizada para la sintonización automática de algoritmos.

Dado que el algoritmo se divide en dos fases distintas, el proceso de ajuste también se realizó en dos etapas para abordar las características específicas de cada fase. Primer se ajustarón los valores de las probabilidades para los operadores para la generación de la forma de la mazmorra, y luego las probabilidades de los operadores para la evolución de la distribución de llaves y barreras.

Además, se sintonizó el algoritmo CEA tanto con como sin DFS para los resultados obtenidos en esta tesis.

6.5.1. Ajuste de la Generación de la Forma

La primera etapa del ajuste se centró en la generación de la forma de la mazmorra, donde se ignoran las barreras y las llaves para simplificar el problema y enfocarse en la estructura básica. Este enfoque permitió optimizar las tasas de mutación para agregar y eliminar habitaciones, así como los operadores de cruce, sin la complejidad añadida por las llaves y barreras.

Se utiliza una amplia variedad de configuraciones de problemas, variando el número de habitaciones y el coeficiente lineal para garantizar que los parámetros ajustados fueran robustos y aplicables a una diversidad de casos como se pueden ver en la tabla 7. Los parámetros ajustados incluyeron:

- **Mutación para eliminar habitaciones**: Se ajusta la probabilidad de eliminar habitaciones para equilibrar la estructura de la mazmorra y evitar un exceso de nodos.
- **Mutación para agregar habitaciones**: Ajustamos la probabilidad de agregar habitaciones para asegurar que la mazmorra cumpliera con los requisitos de tamaño y forma.
- **Operador de cruce**: Se ajusta la probabilidad del operador de cruce para combinar efectivamente las características de dos mazmorras y explorar nuevas configuraciones.
- **Movimientos simétricos**: Se ajusta los movimientos simétricos para tener más opciones de cruce que cumplan con las características deseadas, pero en distinta configuración.

El proceso de ajuste se ejecuta iterativamente, evaluando el rendimiento del algoritmo en términos de la calidad de la forma de la mazmorra generada y la eficiencia computacional. Este enfoque permitió identificar las combinaciones óptimas de parámetros para la primera fase del 2-Step EA.

Habitaciones	Llaves	Barreras	Coeficiente Lineal	Llaves necesarias
10	2	2	2.0	2.0
15	3	2	2.0	2.0
20	4	4	1.0	4.0
20	4	4	2.0	4.0
30	6	6	1.0	4.0
30	6	6	1.5	6.0
30	6	6	2.5	6.0
100	20	20	1.5	20.0

Tabla 7: Configuraciones utilizados para sintonizar el algoritmo.

6.5.2. Ajuste de la Evolución de Barreras y Llaves

Una vez que se completó el ajuste de la generación de la forma, se procede a la segunda etapa: la evolución de la distribución de llaves y barreras. En esta fase, el problema se volvió más complejo, ya que ahora se debían considerar las restricciones impuestas por las llaves y las barreras.

El ajuste en esta fase se centró en las mutaciones específicas para agregar y eliminar llaves y barreras, así como en mantener la coherencia del mapa generado. Al igual que en la primera fase, se usó Irace para ajustar los siguientes parámetros:

- **Mutación para agregar barreras y llaves**: Se ajustó la probabilidad de agregar barreras y llaves para asegurar que la mazmorra tuviera la cantidad adecuada de obstáculos y elementos necesarios.
- **Mutación para eliminar barreras y llaves**: Se ajustó la probabilidad de eliminar barreras y llaves para evitar configuraciones excesivamente restrictivas que pudieran impedir la jugabilidad.

Nuevamente, se probó una amplia variedad de configuraciones de problemas, incluyendo distintas combinaciones de habitaciones, llaves, barreras y coeficientes lineales. Este enfoque permitió validar la robustez de los parámetros ajustados y asegurar que el algoritmo pudiera manejar una diversidad de escenarios.

6.5.3. Importancia de las Probabilidades de Mutación

Las probabilidades de mutación son elementos críticos para 2-Step EA. Estas probabilidades determinan la frecuencia con la que se realizan cambios en las soluciones durante la evolución del algoritmo. En el caso del 2-Step EA, estas probabilidades tienen un impacto directo

en la forma de las mazmorras generadas y en la distribución de llaves y barreras.

Un aspecto relevante es que tanto la adición como la eliminación de habitaciones se aplican solo cuando es necesario ajustar el número de habitaciones en la mazmorra. Esto significa que los operadores de mutación no se aplicarán de forma predeterminada en cada iteración, sino que se activarán cuando sea necesario adaptar la estructura de la mazmorra a las necesidades específicas.

La Tabla 8 proporciona una visión detallada de los parámetros ajustados para el 2-Step EA, incluyendo las generaciones, las probabilidades de mutación y cruce para ambas fases del algoritmo.

Este proceso meticuloso de ajuste garantiza que el 2-Step EA esté finamente calibrado para manejar la generación de mazmorras de manera efectiva, optimizando tanto la forma de la mazmorra como la distribución de llaves y barreras. Estos valores de probabilidad reflejan una combinación cuidadosamente equilibrada de exploración y explotación, lo que permite al algoritmo buscar soluciones nuevas y prometedoras mientras mantiene la coherencia y la jugabilidad de las mazmorras generadas.

Parámetro	Valor
Generaciones (Forma)	180
Población	60
Mutación (Eliminar Habitación)	0.838
Mutación (Agregar Habitación)	0.848
Crossover (Forma)	0.826
Movimiento Simétrico	0.271
Generaciones (Barreras y Llaves)	100
Mutación (Agregar BK)	0.300
Mutación (Eliminar BK)	0.700

Tabla 8: Parámetros ajustados para el Algoritmo Evolutivo de Dos Pasos.

6.6. Resultados

Al analizar las métricas obtenidas en los ocho escenarios mencionados en [Pereira *et al.*, 2021], podemos observar diferencias significativas entre el 2-Step EA y el Algoritmo Evolutivo Restringido (CEA). Las Tablas 9 y 10 presentan un resumen detallado de estos resultados.

Al examinar la Tabla 9, que se centra en el número de habitaciones, llaves y barreras, y su diferencia estadística entre el 2-Step EA y el CEA, se puede observar patrones interesantes. En los escenarios con 30 habitaciones, el rendimiento del 2-Step EA se asemeja más a los resultados obtenidos del CEA, lo que podría indicar desafíos en el rendimiento específico de este escenario o que ambos algoritmos están cerca del óptimo. Sin embargo, para problemas más grandes (100 y 500 habitaciones), el 2-Step EA supera significativamente al CEA en la asignación de llaves y barreras. Esta diferencia se puede atribuir a la función dual del cruce del CEA, que prioriza mantener el número de habitaciones especiales, lo que lo hace menos eficiente en escenarios con un alto número de llaves y barreras.

Considerando los resultados en la segunda tabla, se analiza los resultados relacionados con la linealidad (expresada como Coeficiente L.), la aptitud del algoritmo y el tiempo necesario para generar las mazmorras. El 2-Step EA produce resultados más cercanos a la linealidad deseada en casi todos los casos excepto para el escenario de 100 habitaciones, donde el CEA proporciona una coincidencia más cercana. Notablemente, la función de aptitud para los casos de 500 habitaciones y 100 habitaciones es significativamente peor en el CEA, lo que sugiere que sacrifica la asignación de barreras y llaves en favor del coeficiente lineal. Es importante destacar que estas comparaciones indican mejores soluciones logradas en menos tiempo por el 2-Step EA.

En resumen, los resultados demuestran que el 2-Step EA tiene ventajas significativas sobre el CEA en la mayoría de los escenarios probados, especialmente en problemas de mayor tamaño. Esto lo convierte en una opción más robusta y eficiente para la generación de mazmorras complejas.

6.6.1. Análisis del Coeficiente Lineal

El análisis del coeficiente lineal proporciona una comprensión más profunda del impacto en el rendimiento de los algoritmos cuando se enfrentan a coeficientes variables en un escenario de problema fijo. Esto permite evaluar cómo se comportan los algoritmos frente a diferentes niveles de linealidad en la generación de mazmorras.

Al observar los resultados de aptitud para una configuración de problema de 20 habitaciones, como se muestra en la Figura 37, se puede obtener información comparativa sobre los comportamientos algorítmicos. El 2-Step EA muestra una tendencia general hacia resultados más favorables. Esto se evidencia claramente por la forma y la posición de los elementos en el diagrama de caja. Por otro lado, el CEA también produce resultados aceptables pero con una mayor variabilidad y, en algunos casos, un rendimiento inferior.

Este análisis resalta el desafío creciente de la generación de mazmorras a medida que el

	Habitaciones				Llaves				Barreras			
	2-Step EA		CEA		2-Step EA		CEA		2-Step EA		CEA	
Configuraciones	Avg.	st. dv.	Avg.	st. dv.	Avg.	st. dv.	Avg.	st. dv.	Avg.	st. dv.	Avg.	st. dv.
(15,3,2,2.0,2.0)	**15.0**	0.0	**15.0**	0.0	**3.0**	0.0	**3.0**	0.0	**2.0**	0.0	**2.0**	0.0
(20,4,4,1.0,4.0)	**20.0**	0.0	**20.0**	0.0	**4.0**	0.0	**4.0**	0.0	**4.0**	0.0	**4.0**	0.0
(20,4,4,2.0,4.0)	**20.0**	0.0	**20.0**	0.0	**4.0**	0.0	**4.0**	0.0	**4.0**	0.0	**4.0**	0.0
(25,8,8,1.0,8.0)	24.99	0.44	**25.0**	0.0	**8.0**	0.0	**8.0**	0.0	**8.0**	0.0	**8.0**	0.0
(30,4,4,2.0,4.0)	**30.0**	0.0	**30.0**	0.0	**4.0**	0.0	**4.0**	0.0	**4.0**	0.0	**4.0**	0.0
(30,6,6,1.5,6.0)	**30.0**	0.0	**30.0**	0.0	**6.0**	0.0	**6.0**	0.0	**6.0**	0.0	**6.0**	0.0
(100,20,20,1.5,20.0)	99.93	0.55	**100.0**	0.0	**20.0**	0.0	20.08	0.27	**20.0**	0.0	18.73	1.28
(500,100,100,1.5,100.0)	498.88	1.94	**499.95**	0.43	**100.0**	0.0	23.81	2.03	**99.99**	0.1	20.21	2.15

Tabla 9: Métricas principales de la mazmorra, considerando el número de habitaciones, llaves y barreras, entre el 2-Step EA y el Algoritmo Evolutivo Restringido (CEA).

	Coeficiente Lineal				valor de aptitud				Tiempo (s)			
	2-Step EA		CEA		2-Step EA		CEA		2-Step EA		CEA	
Configuraciones	Avg.	st. dv.	Avg.	st. dv.	Avg.	st. dv.	Avg.	st. dv.	Avg.	st. dv.	Avg.	st. dv.
(15,3,2,2.0,2.0)	**1.99**	0.05	1.77	0.13	**0.02**	0.1	0.45	0.26	**3.37**	0.14	3.65	0.22
(20,4,4,1.0,4.0)	**1.01**	0.03	1.12	0.11	**0.02**	0.05	0.25	0.23	**4.41**	0.21	4.83	0.49
(20,4,4,2.0,4.0)	**1.89**	0.03	1.66	0.11	**0.21**	0.07	0.68	0.23	**3.46**	0.16	4.11	0.29
(25,8,8,1.0,8.0)	**1.04**	0.02	1.28	0.09	**0.31**	0.85	0.58	0.2	**4.01**	0.19	4.47	0.27
(30,4,4,2.0,4.0)	**1.91**	0.06	1.88	0.1	**0.18**	0.11	0.25	0.2	**3.71**	0.11	3.91	0.18
(30,6,6,1.5,6.0)	**1.52**	0.01	1.53	0.0	**0.05**	0.01	**0.05**	0.0	**3.9**	0.1	4.31	0.2
(100,20,20,1.5,20.0)	1.48	0.02	**1.49**	0.03	**0.34**	1.07	2.87	2.85	5.85	0.29	**5.68**	0.4
(500,100,100,1.5,100.0)	**1.44**	0.03	1.4	0.04	**4.6**	2.86	314.34	7.61	17.34	0.85	**8.47**	0.81

Tabla 10: Coeficiente Lineal de la mazmorra, aptitud obtenida y tiempo en segundos para que el algoritmo genere la mazmorra, entre el 2-Step EA y el Algoritmo Evolutivo Restringido (CEA).

coeficiente lineal aumenta. A medida que se incrementa este coeficiente, tanto el 2-Step EA como el CEA experimentan dificultades adicionales para alcanzar soluciones óptimas. Sin embargo, el 2-Step EA parece ser más robusto frente a este desafío en comparación con el CEA, lo que sugiere una mayor capacidad para adaptarse a la variabilidad de los problemas con coeficientes lineales más altos.

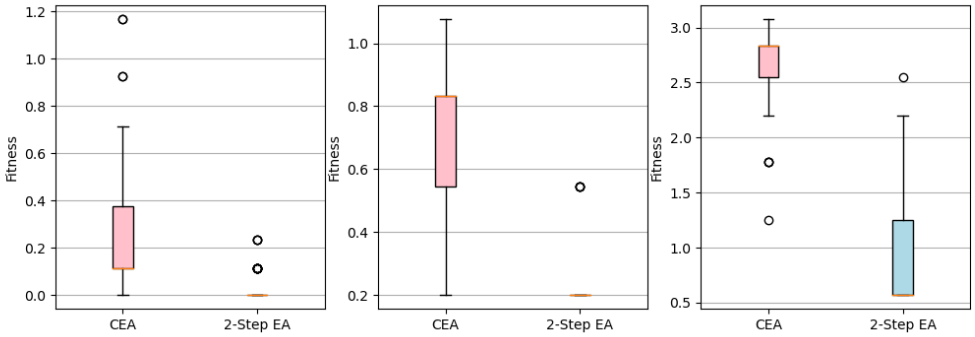


Figura 37: Boxplot del Fitness resultante en el problema [20,4,4,X,4], con un coeficiente Linear (X) variable entre 1.0, 2.0, y 3.0, comparando el algoritmo Constrained Evolutionary Algorithm (CEA) con el 2-Step EA.

Un análisis más detallado de la asignación de habitaciones a lo largo de 100 generaciones, con coeficientes lineales variables, ofrece una visión más profunda de la adaptabilidad de cada algoritmo. Al examinar los mapas de calor en las Figuras 38 y 39 para el CEA y el 2-Step EA respectivamente, se puede discernir la distribución espacial y la frecuencia de las habitaciones generadas.

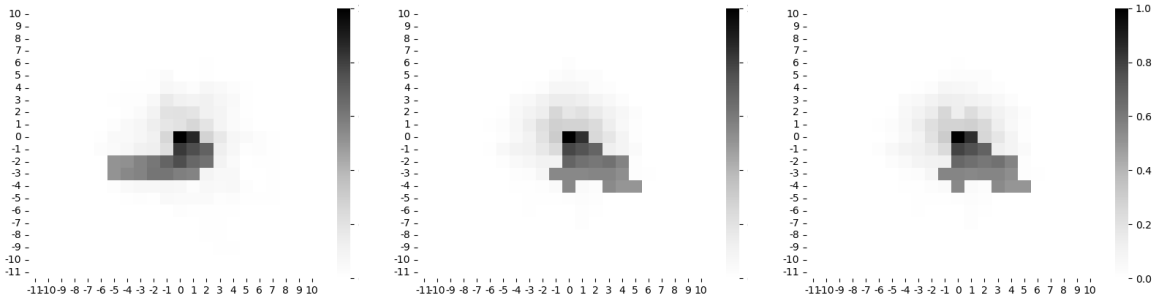
El CEA parece concentrar la generación de habitaciones dentro de un rango de coeficientes lineales de 2.0 a 3.0, lo que sugiere una menor diversidad en las configuraciones específicas. En contraste, el 2-Step EA exhibe una mayor variabilidad, especialmente entre coeficientes lineales de 2.0 o más, indicando una adaptabilidad superior en la distribución de habitaciones.

Este análisis comparativo no solo resalta las diferencias en los enfoques y resultados de cada algoritmo, sino que también subraya el desafío sutil presentado por coeficientes lineales variables en la generación de mazmorras. Cuando el coeficiente lineal supera los 2.0, tiende a generar superposiciones de nodos ”hijos”que no pueden colocarse en el mismo lugar, limitando las configuraciones posibles y resultando en mazmorras forzadas. Este fenómeno se aprecia claramente en los resultados, ya que ninguno de los problemas tiene una generación para un coeficiente lineal mayor de 2.0.

Además, la Figura 39 proporciona una comparación directa del coeficiente lineal obtenido por cada algoritmo en varios escenarios de problema. Se observa que, aunque ambos algo-

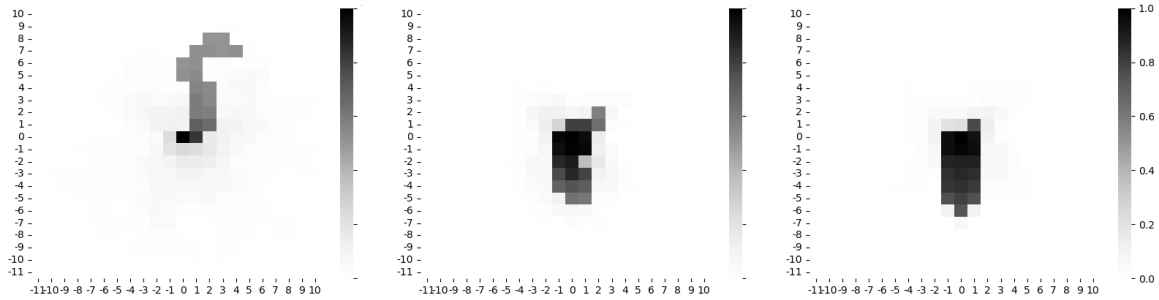
ritmos tienden a mantener el coeficiente lineal dentro de un rango aceptable, el 2-Step EA logra una mayor estabilidad y menor variabilidad en los coeficientes lineales generados. Esto sugiere que el 2-Step EA no solo es más efectivo en la generación de mazmorras, sino que también es más robusto frente a cambios en los parámetros del problema.

En resumen, el análisis del coeficiente lineal resalta la capacidad del 2-Step EA para adaptarse mejor a variaciones en el problema y generar mazmorras más eficientes y menos forzadas. La menor variabilidad y la mayor estabilidad en los coeficientes lineales obtenidos refuerzan la superioridad del 2-Step EA en la generación de mazmorras complejas, particularmente en escenarios con mayores exigencias de linealidad.



(a) L. Coefficient $= 1{,}0$ (b) L. Coefficient $= 2{,}0$ (c) L. Coefficient $= 3{,}0$

Figura 38: Mapa de calor para visualizar la ubicación de las habitaciones dentro de un laberinto por CEA, donde el eje y representa la posición en Y y el eje X representa la posición de la habitación en X, considerando la primera habitación como la habitación (0,0). Las generaciones fueron realizadas para el problema [20,4,4,X,4], con un coeficiente Linear (X) variable.



(a) L. Coefficient $= 1{,}0$ (b) L. Coefficient $= 2{,}0$ (c) L. Coefficient $= 3{,}0$

Figura 39: Mapa de calor para visualizar la ubicación de las habitaciones dentro de un laberinto por 2-StepEA, donde el eje y representa la posición en Y y el eje X representa la posición de la habitación en X, considerando la primera habitación como la habitación (0,0). Las generaciones fueron realizadas para el problema [20,4,4,X,4], con un coeficiente Linear (X) variable.

6.7. Análisis con el Test de Wilcoxon

El análisis del Test de Wilcoxon compara las soluciones proporcionadas por el 2-Step EA y el CEA para diversos escenarios de problemas. Este test estadístico no paramétrico es útil para evaluar si existen diferencias significativas entre los dos algoritmos en términos de las soluciones generadas.

Este análisis revela que el 2-Step EA tiene un rendimiento significativamente mejor en casi todos los casos, excepto para los problemas de 30 habitaciones, donde ambos algoritmos presentan resultados excelentes. En estos escenarios específicos, los valores p no indican una diferencia estadísticamente significativa, lo que sugiere que ambos algoritmos son igualmente efectivos.

Problemas con 15, 20, 25, 100 y 500 habitaciones: Los valores p extremadamente bajos indican una diferencia significativa en el rendimiento de los algoritmos, con el 2-Step EA superando al CEA. Esto demuestra la robustez del 2-Step EA en la generación de mazmorras más complejas y diversas.

Problemas con 30 habitaciones (30,4,4,2.0,4.0 y 30,6,6,1.5,6.0): Los valores p indican que no hay diferencias significativas entre los dos algoritmos. Ambos alcanzan un rendimiento óptimo y consistente, lo que sugiere que estos problemas no son suficientemente desafiantes

para distinguir el rendimiento entre los algoritmos.

Estos resultados proporcionan una validación estadística sólida del rendimiento superior del 2-Step EA en la mayoría de los escenarios. Sin embargo, también resaltan la necesidad de considerar la complejidad y los parámetros del problema al evaluar y comparar algoritmos de generación de mazmorras. El 2-Step EA no solo muestra una mayor adaptabilidad y eficacia en problemas más grandes y complejos, sino que también mantiene una consistencia robusta en su desempeño, lo que lo hace una elección preferida para aplicaciones que requieren soluciones optimizadas en entornos de generación de mazmorras.

Configuración	Wilcoxon p value
(15,3,2,2.0,2.0)	**7.7834e-18**
(20,4,4,1.0,4.0)	**5.8892e-19**
(20,4,4,2.0,4.0)	**5.3492e-18**
(25,8,8,1.0,8.0)	**2.9069e-10**
(30,4,4,2.0,4.0)	6.5911e-01
(30,6,6,1.5,6.0)	1.0000e+00
(100,20,20,1.5,20.0)	**2.4416e-17**
(500,100,100,1.5,100.0)	**1.0080e-18**

Tabla 11: Wilcoxon test results. Bold values indicate statistical significance in the obtained results.

6.8. Conclusiones

Los experimentos realizados en esta sección han permitido obtener varias conclusiones sobre el rendimiento y la eficacia del Algoritmo Evolutivo de Dos Pasos (2-Step EA) en comparación con el Algoritmo Evolutivo Constrained (CEA).

Al dividir el problema, se puede mejorar significativamente la eficiencia temporal, aproximadamente de un 5 % a un 18 % más rápido que el algoritmo original sin la Búsqueda en Profundidad (DFS) en los casos de habitaciones inferiores. La reducción en la necesidad de extensas búsquedas DFS para determinar los caminos del usuario contribuyó a esta ventaja de rendimiento. Sin embargo, incorporar búsquedas DFS en el algoritmo sí condujo a disminuciones en la eficiencia temporal y la calidad de la solución, lo que indica un compromiso entre exhaustividad y velocidad.

Las conclusiones principales son:

- **Eficiencia Temporal:** La eliminación de la Búsqueda en Profundidad (DFS) en ambos

algoritmos resultó en una mejora significativa en la velocidad computacional sin comprometer la calidad de las mazmorras generadas.

- **Calidad de las Mazmorras:** Ambos algoritmos demostraron ser capaces de generar mazmorras de alta calidad, aunque el 2-Step EA mostró una ligera ventaja en términos de conectividad y estructura.
- **Capacidades de Generación:** El 2-Step EA mostró capacidades superiores en la generación de mazmorras complejas y bien conectadas, mientras que el CEA proporcionó un control detallado sobre la colocación de llaves y barreras.

En resumen, el Algoritmo Evolutivo de Dos Pasos (2-Step EA) ha demostrado ser una solución eficiente y eficaz para la generación procedural de mazmorras, ofreciendo ventajas significativas en términos de velocidad y calidad. Estos resultados sugieren que el 2-Step EA es una herramienta prometedora para aplicaciones futuras en este campo.

En el siguiente capítulo, se analizará en detalle los mapas generados para cada problema específico. Se evaluará la linealidad de los mapas generados.

CAPÍTULO 7
Análisis de mapas generados

Para validar la efectividad del algoritmo propuesto, se realizaron una serie de pruebas utilizando diferentes configuraciones de mazmorras y parámetros de evolución. Las métricas de evaluación incluyeron la complejidad de la mazmorra, el tiempo de generación y la satisfacción de las restricciones impuestas.

7.1. Mapas Generados para Cada Problema

En las figuras se presentan los mapas generados para las siguientes especificaciones de usuario.

Habitaciones	Llaves	Barreras	Coeficiente Lineal	Llaves necesarias
15	3	2	2.0	2.0
20	4	4	1.0	4.0
20	4	4	2.0	4.0
25	8	8	1.0	8.0
30	4	4	2.0	4.0
30	6	6	1.5	6.0

Tabla 12: Configuraciones de usuario.

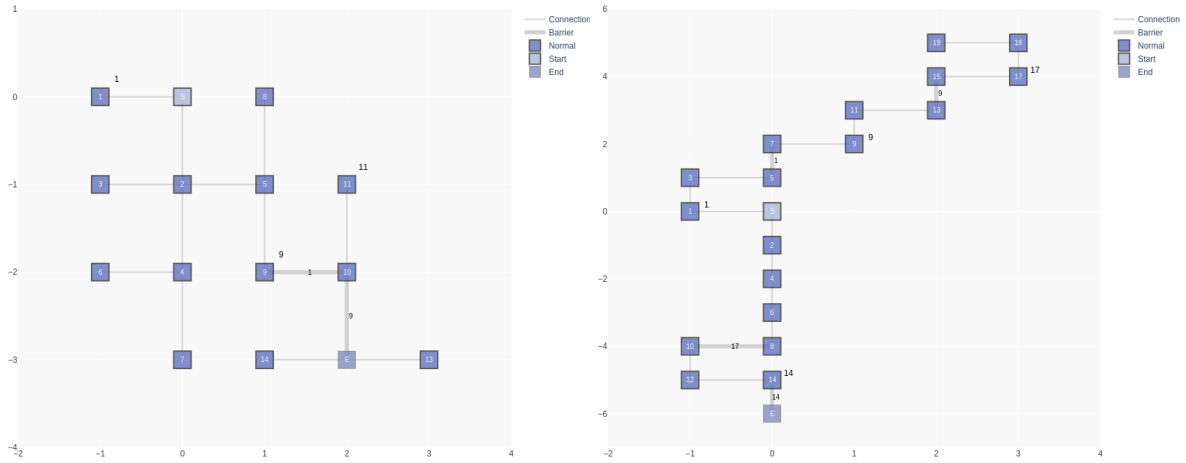
Al analizar los mapas generados para cada problema, se puede observar una tendencia clara, a medida que se agregan más habitaciones, la complejidad de los mapas aumenta y los caminos críticos para pasar los niveles se vuelven más intrincados. Esta observación es fundamental para comprender cómo la estructura de la mazmorra evoluciona con respecto al tamaño y la complejidad del problema.

Es importante destacar que, para los niveles con un coeficiente lineal de 1.0, el establecimiento de llaves y barreras puede parecer menos significativo en comparación con niveles con coeficientes lineales más altos. Esto se debe a que, en estos casos, las llaves necesarias para desbloquear barreras suelen encontrarse en el mismo camino hacia la salida. Esta disposición reduce la necesidad de una exploración exhaustiva, ya que la linealidad inherente al coeficiente lineal de 1.0 facilita el progreso a través del nivel sin la necesidad de desviarse hacia diferentes áreas de la mazmorra.

Por lo tanto, en los niveles con coeficientes lineales más bajos, la generación de llaves y barreras puede no tener un impacto tan notable en la estructura y la jugabilidad del nivel, ya

que el diseño lineal del nivel naturalmente guía al jugador a través de las secciones relevantes. Esto resalta la importancia de considerar el coeficiente lineal en conjunto con otros parámetros al evaluar la complejidad y la experiencia del jugador en un nivel generado por el algoritmo.

Para ilustrar estos puntos, se analizan algunos ejemplos



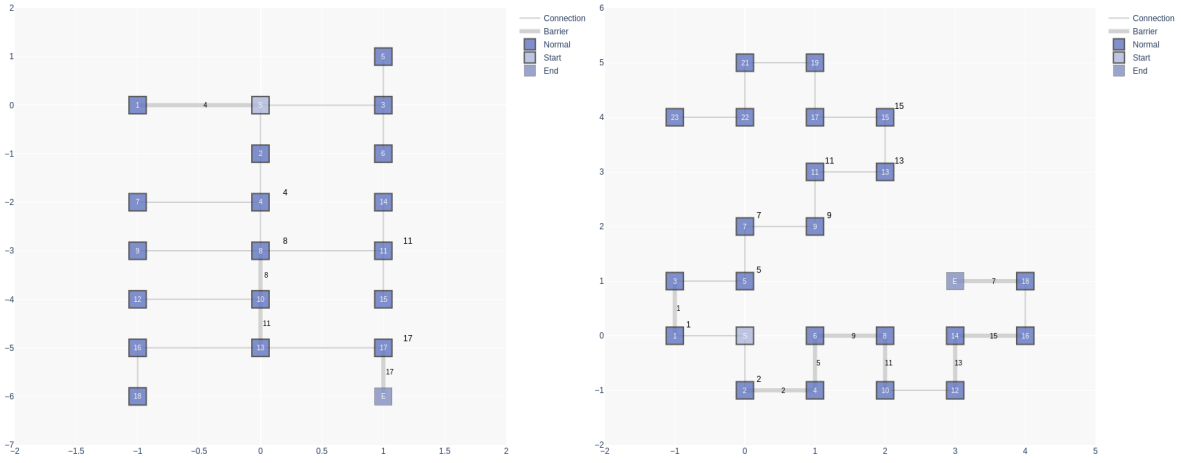
(a) (15,3,2,2.0,2.0) (b) (20,4,4,1.0,4.0)

Figura 40: Mapas generados por el algoritmo 2-StepEA para los problemas (15,3,2,2.0,2.0) y (20,4,4,1.0,4.0).

En el mapa generado para la configuración (15,3,2,2.0,2.0), se observa un diseño más intrincado con múltiples caminos y obstáculos (ver Figura 40a). Este nivel de complejidad exige al jugador una mayor exploración para encontrar las llaves y avanzar hacia la salida. En contraste, el mapa generado para el problema (20,4,4,1.0,4.0) presenta una estructura más lineal (ver Figura 40b). Las llaves necesarias para desbloquear barreras están ubicadas a lo largo del camino principal hacia la salida, reduciendo así la necesidad de exploración.

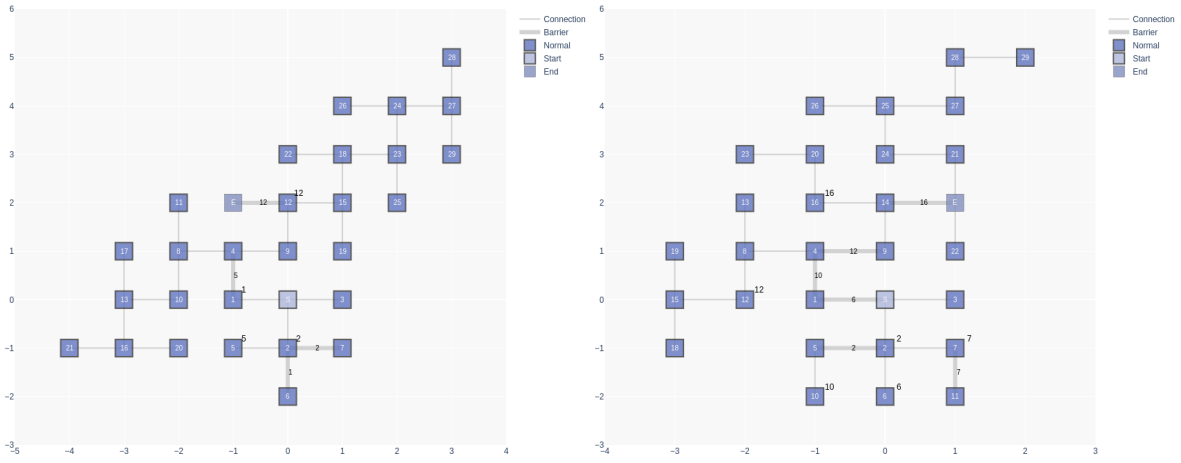
En la Figura 41 se presenta el mapa correspondiente a la configuración (20,4,4,2.0,4.0) muestra una mayor complejidad, con más bifurcaciones y áreas que explorar (ver Figura 41a). Este diseño incrementa el desafío para el jugador, al requerir una navegación más detallada del entorno. Por otro lado, el mapa para la configuración (25,8,8,1.0,8.0) es más sencillo y directo debido a su coeficiente lineal de 1.0, lo que facilita un avance más rápido y menos complejo a través del nivel (ver Figura 41b).

En el mapa del problema (30,4,4,2.0,4.0), la complejidad es notablemente mayor, con una estructura muy ramificada que demanda una exploración exhaustiva por parte del jugador (ver Figura 42a). El mapa del problema (30,6,6,1.5,6.0) tiene una complejidad intermedia. Su coeficiente lineal de 1.5 permite una mezcla de elementos lineales y ramificados, proporcionando un equilibrio entre exploración y progresión directa (ver Figura 42b).



(a) (20,4,4,2.0,4.0) (b) (25,8,8,1.0,8.0)

Figura 41: Mapas generados por el algoritmo 2-StepEA para los problemas (20,4,4,2.0,4.0) y (25,8,8,1.0,8.0).



(a) (30,4,4,2.0,4.0) (b) (30,6,6,1.5,6.0)

Figura 42: Mapas generados por el algoritmo 2-StepEA para los problemas (30,4,4,2.0,4.0) y (30,6,6,1.5,6.0) .

7.2. Análisis de Linealidad en Mapas Generados

En esta sección se analizan los mapas generados con distintos coeficientes lineales para configuraciones del tipo $(15, 2, 2, L_c, 2)$.

Cuando el coeficiente lineal es de 1.0, los mapas generados tienden a presentar una estructura más directa y lineal. En estos casos, la colocación de llaves y barreras puede parecer menos significativa, ya que el jugador puede encontrar todas las llaves necesarias a lo largo de un camino casi recto hacia la salida. Este diseño lineal facilita la progresión a través del nivel sin la necesidad de explorar múltiples ramificaciones o áreas divergentes de la mazmorra.

Sin embargo, a medida que el coeficiente lineal aumenta, los mapas generados se vuelven más complejos y ramificados. Esto se traduce en niveles que requieren que el jugador explore más para encontrar las llaves necesarias, lo que aumenta el nivel de desafío y la profundidad del juego. La mayor linealidad impuesta por un coeficiente lineal más alto conduce a una distribución menos predecible de llaves y barreras, lo que fomenta una experiencia de juego más rica y diversa.

En resumen, el análisis de la linealidad en los mapas generados destaca cómo el coeficiente lineal influye en la estructura y la jugabilidad de los niveles. Desde mapas más lineales y directos hasta niveles más complejos y desafiantes, la variación en el coeficiente lineal permite al algoritmo adaptarse a una amplia gama de preferencias de diseño y estilos de juego.

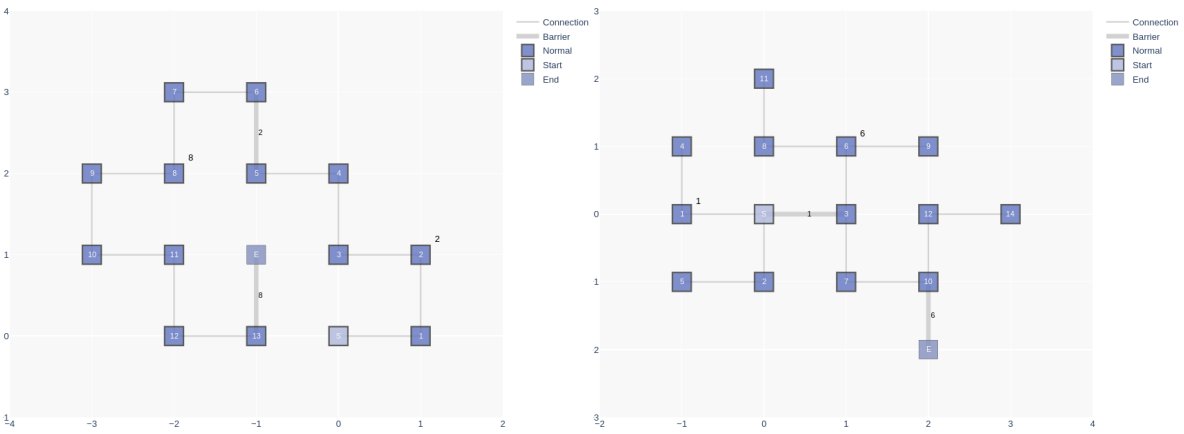


Figura 43: Mapas generados por el algoritmo 2-StepEA para el problema (15,2,2, Lc, 2) con coeficientes lineales 1.0 y 2.5.

7.2.1. Sentido de 1.0 como coeficiente lineal

Los mapas con un coeficiente lineal de 1.0 tienden a ser más sencillos y directos, facilitando una progresión rápida. Sin embargo, esta simplicidad puede hacer que el juego sea menos desafiante y menos interesante para jugadores que buscan una experiencia más rica y diversa.

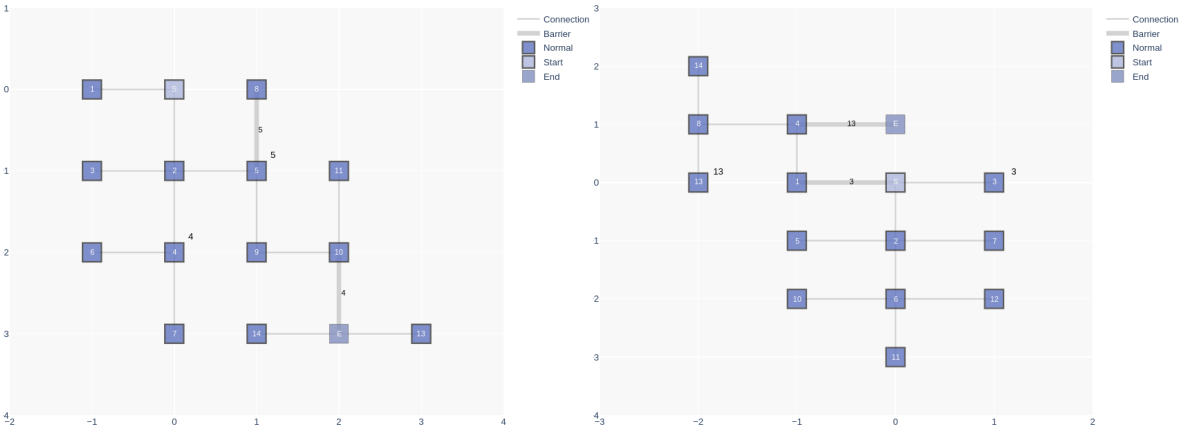


Figura 44: Mapas generados por el algoritmo 2-StepEA para el problema (15,2,2,Lc,2) con coeficientes lineales 2.0 y 2.5.

Problema (15,2,2,3.0,2):

Finalmente, el mapa generado para el problema (15,2,2,3.0,2) muestra el mayor nivel de complejidad entre todos los ejemplos presentados. Con un coeficiente lineal de 3.0, el diseño es altamente ramificado y desafiante, requiriendo una exploración extensa y cuidadosa para avanzar.

7.3. Conclusión del capítulo

Para evaluar la efectividad del algoritmo 2-StepEA, se realizaron pruebas con distintas configuraciones de mazmorras y parámetros de evolución, analizando la complejidad, el tiempo de generación y la satisfacción de las restricciones. Los resultados demostraron que el algoritmo genera mazmorras coherentes y jugables en un tiempo razonable.

El análisis de los mapas reveló una tendencia significativa: a mayor número de habitaciones y coeficiente lineal, la complejidad de los mapas aumenta, y los caminos críticos se vuelven

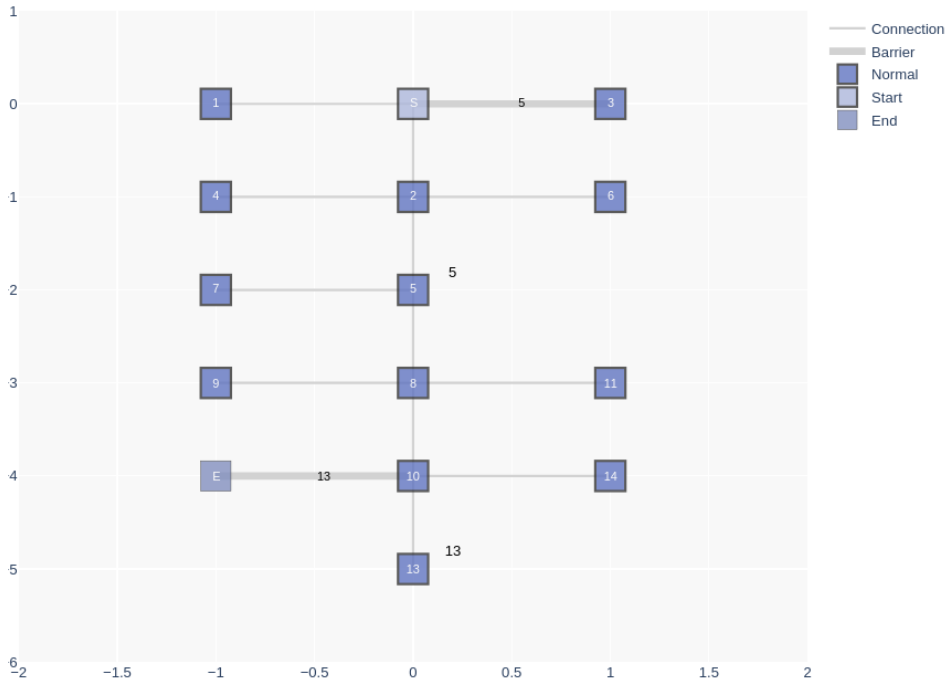


Figura 45: Mapa generados por el algoritmo 2-StepEA para el problema (15,2,2,3.0,2)

más intrincados. En niveles con un coeficiente lineal bajo (e.g., 1.0), las llaves y barreras se encuentran en caminos directos, lo que facilita el progreso sin mucha desviación. Por otro lado, coeficientes lineales altos generan mapas más complejos que requieren mayor exploración, aumentando así el desafío y la profundidad del juego.

Además, el estudio de la linealidad en los mapas mostró que coeficientes lineales bajos crean estructuras más directas y predecibles, mientras que coeficientes altos producen niveles más ramificados y exigentes. Esto permite al algoritmo adaptarse a diversas preferencias de diseño y estilos de juego, desde niveles sencillos hasta desafíos complejos que requieren exploración exhaustiva.

En el siguiente capítulo, se presentan las conclusiones del trabajo realizado, discutiendo los alcances y limitaciones del algoritmo propuesto. Se hace un análisis sobre la complejidad y adaptabilidad del método, así como sobre la naturaleza de los nodos y habitaciones generados. Se presenta además posibles direcciones para investigaciones futuras.

CAPÍTULO 8
Conclusiones

Las conclusiones de este trabajo representan el cierre de un proceso de investigación, donde se reflexiona sobre los resultados obtenidos, se evalúan los logros alcanzados y se identifican las posibles direcciones futuras para la investigación en el área de generación de mazmorras en videojuegos. En esta sección, se abordarán los alcances y limitaciones de la propuesta de solución, se discutirá sobre la validez de los objetivos planteados, se identificarán las principales contribuciones y aplicaciones del trabajo realizado, y se destaca el impacto o aporte potencial a la comunidad interesada en este campo. Además, se proporcionarán recomendaciones para aquellos que deseen profundizar en el tema y se esbozan posibles líneas de investigación futuras.

8.1. Alcances y Limitaciones

El enfoque adoptado en este trabajo, basado en el algoritmo evolutivo de dos pasos (2-Step EA) para la generación de mazmorras, ha demostrado ser efectivo en abordar los parámetros específicos del problema, como el número de habitaciones, llaves, barreras y coeficiente lineal. La estructura del 2-Step EA permitió dividir el problema en dos fases distintas: la primera fase se centró en la optimización de la estructura de las habitaciones y el coeficiente lineal, mientras que la segunda fase se encargó de la colocación eficiente de llaves y barreras.

Sin embargo, es importante reconocer las limitaciones actuales que presenta este enfoque:

8.1.1. Complejidad y Adaptabilidad:

A pesar de su efectividad, el 2-Step EA tiene dificultades para manejar coeficientes lineales superiores a 2.0, lo que provoca superposiciones de nodos “hijo” que no pueden colocarse en el mismo lugar. Esto reduce la diversidad de los mapas generados y puede resultar en mazmorras forzadas. Esto puede deberse a cómo se evalúa el coeficiente lineal, que contempla el promedio de los nodos hijos, dado que en mapas más conectados y, por tanto, sobrepuestos cuesta lograrlo en su mayoría, la generación actual no contempla la creación de loops, lo que limita la complejidad y la exploración no lineal de las mazmorras. No se han implementado conexiones de llaves de tipos 1->n, m->1, y m->n, lo que podría enriquecer las mecánicas de juego.

8.1.2. Naturaleza de los Nodos y Habitaciones:

La conexión del nodo inicial es establecida de manera fija, lo que puede limitar la variabilidad inicial de las mazmorras. Los niveles generados son principalmente planos y bidimensionales. No se ha explorado la generación de mazmorras más orgánicas o en 3D, lo que podría ofrecer experiencias de juego más inmersivas.

8.2. Resultados y Contribuciones

Los resultados obtenidos de la aplicación del 2-Step EA muestran mejoras significativas respecto a CEA en varios aspectos clave:

- **Eficiencia Temporal:** El 2-Step EA es hasta un 18 % más rápido que el algoritmo original en ciertos casos, lo que permite una generación de niveles más eficiente.
- **Adaptabilidad y Diversidad:** La mayor adaptabilidad en la distribución de habitaciones sugiere que el enfoque propuesto puede generar mazmorras con una mayor variedad de configuraciones. Este es un avance significativo en la generación de contenido procedural.
- **Calidad de las Mazmorras Generadas:** A través de experimentos y análisis comparativos, se ha demostrado que el 2-Step EA supera al CEA en la mayoría de los escenarios, con diferencias significativas en problemas que involucran 15, 20, 25, 100 y 500 habitaciones. Sin embargo, en problemas con 30 habitaciones, ambos algoritmos muestran un rendimiento excelente sin diferencias significativas, lo que sugiere que para problemas de esta escala, ambos enfoques son igualmente efectivos.
- **Impacto del Coeficiente Lineal:** Se ha observado que coeficientes lineales superiores a 2.0 tienden a limitar la generación de mazmorras debido a la superposición de nodos. Esto sugiere la necesidad de reinterpretar el coeficiente lineal, quizás como distancias o rutas alternativas, para mejorar la diversidad y calidad de las mazmorras generadas.

8.3. Validez de los Objetivos

Los objetivos establecidos al inicio de la investigación se han cumplido satisfactoriamente:

- **Desarrollo y Eficiencia del Algoritmo:** Se logró desarrollar un algoritmo evolutivo que aborda los parámetros específicos del problema de generación de mazmorras, demostrando su efectividad a través de una serie de experimentos y análisis comparativos. La separación en dos fases del proceso de generación permitió una optimización más enfocada y eficiente, lo que respalda la validez de la metodología propuesta.

8.4. Impacto y Aplicaciones

El impacto potencial de este trabajo se extiende más allá del ámbito académico, teniendo repercusiones directas en la industria del desarrollo de videojuegos:

- **Herramientas para Desarrolladores:** La capacidad de generar mazmorras de manera eficiente y diversa puede beneficiar a los desarrolladores al proporcionarles herramientas para crear experiencias de juego más atractivas y variadas.
- **Ampliación del Alcance:** Este enfoque podría aplicarse a otros tipos de entornos generados proceduralmente, como mapas de ciudades o mundos virtuales, ampliando así su alcance y utilidad.

8.5. Conclusiones sobre el uso del Algoritmo Evolutivo y la Representación Actual

El uso del algoritmo evolutivo en este trabajo ha sido fundamental para abordar la complejidad de la generación de mazmorras. La estructura evolutiva del algoritmo permite una exploración efectiva del espacio de soluciones, optimizando tanto la estructura de las habitaciones como la colocación de elementos clave como llaves y barreras. Este enfoque ha demostrado ser robusto y adaptable a diversos parámetros del problema, proporcionando una base sólida para futuras mejoras y expansiones.

La representación actual, que se centra en una estructura de grafos y habitaciones en un plano bidimensional, ha facilitado la implementación y el análisis del algoritmo. Esta representación permite una manipulación clara y directa de los elementos de la mazmorra, lo que es crucial para el ajuste fino de los parámetros evolutivos. Sin embargo, también presenta limitaciones significativas:

- **Limitación en la Complejidad Espacial:** La representación limita los mapas en cuanto a que el nodo inicial solo se encuentra conectado con tres otros nodos, dejando la conexión al norte vacía, por lo que no es capaz de producir todos los posibles mapas factibles, dado esto y para disminuir la influencia de este dilema se podría aplicar una transformación por rotación y espejo a la solución para obtener niveles más diversos, pero aún así no todos los posibles, por lo que abordar otra representación sería interesante para poder abordar todo el universo de mapas posibles.
- **Restricciones de llaves y barreras por habitación:** Las conexiones que tenemos actualmente tanto como las llaves, se definen por habitación por lo cual no se puede colocar una llave y una barrera en la misma habitación, limitando los posibles niveles

que contemplen estas situaciones, por lo que modificar esto sería interesante, además de abordar el límite de complejidad espacial.

El uso del algoritmo evolutivo así como la representación actual han sido efectivos para alcanzar los objetivos del trabajo, pero es evidente que hay margen para mejorar en términos de complejidad y adaptabilidad. Estos aspectos deben ser considerados en investigaciones futuras para avanzar en la generación procedural de mazmorras.

8.6. Recomendaciones y Futuras Líneas de Investigación

Para aquellos interesados en profundizar en este tema, se recomienda explorar las siguientes áreas:

- **Mejora de la Eficiencia y Diversidad:** Incorporar técnicas de aprendizaje automático o explorar enfoques híbridos que combinan algoritmos evolutivos con otras metodologías puede ser beneficioso para mejorar tanto la eficiencia como la diversidad en la generación de mazmorras.
- **Factores Adicionales:** Investigar el impacto de factores adicionales, como la simetría o la coherencia temática, en la generación de mazmorras puede proporcionar información valiosa para enriquecer la experiencia de juego y la calidad de los niveles generados.
- **Manejo de Coeficientes Lineales:** Optimizar el manejo de coeficientes lineales superiores a 2.0 para evitar la generación de mazmorras forzadas y mejorar la diversidad de las configuraciones posibles es crucial para garantizar la calidad y variedad de los niveles generados.
- **Generación de Mazmorras Orgánicas y en 3D:** Explorar la generación de mazmorras más orgánicas o en 3D puede ofrecer experiencias de juego más inmersivas y variadas, lo que contribuiría significativamente a la calidad del contenido generado, especialmente en 3D, dado que esta área está poco investigada en el campo de la generación procedural de contenido.
- **Población de Niveles:** Desarrollar métodos para poblar los niveles generados con enemigos y otros elementos, creando desafíos equilibrados y variados, es fundamental para garantizar una experiencia de juego completa y satisfactoria para los jugadores. Esta población de niveles también se podría implementar de manera dinámica dependiendo de cómo el jugador enfrente los niveles, apuntando a distintos requerimientos del diseñador como requerimientos de dificultad basada en tiempo, habilidades u otros aspectos del jugador.

- **Conclusiones adicionales sobre la representación de niveles:** Considerar la posibilidad de representar niveles en múltiples pasos para generar habitaciones más orgánicas o combinar habitaciones para crear mazmorras con diferentes tamaños y conexiones puede abrir nuevas oportunidades para la creación de niveles más interesantes y desafiantes. Explorar nuevas representaciones de los niveles para abordar el problema de manera más efectiva, incluyendo la combinación de habitaciones de distintos tamaños y conexiones, y la inclusión de enemigos y otros desafíos, puede llevar a mejoras significativas en la calidad y diversidad del contenido generado.

8.7. Aplicaciones Prácticas en el Desarrollo de Videojuegos

Además de identificar las mejoras y limitaciones del algoritmo propuesto, es fundamental discutir cómo estos hallazgos podrían aplicarse en contextos prácticos de desarrollo de videojuegos. Los diseñadores de juegos y los desarrolladores de software podrían utilizar esta propuesta para mejorar la experiencia de juego para los jugadores. La capacidad de generar mazmorras de manera eficiente y diversa podría proporcionar herramientas valiosas para crear experiencias de juego más atractivas y variadas, lo que podría aumentar la satisfacción del jugador y la longevidad del juego.

8.8. Impacto en la Experiencia del Jugador

La generación de mazmorras en videojuegos desempeña un papel crucial en la experiencia del jugador. La diversidad y la adaptabilidad de las mazmorras pueden influir significativamente en la percepción del juego por parte del jugador. Por lo tanto, es importante considerar cómo estas mejoras en la generación de mazmorras podrían traducirse en una experiencia de juego más inmersiva y satisfactoria para los jugadores. La capacidad de ofrecer mazmorras únicas y desafiantes podría aumentar la rejugabilidad y el disfrute general del juego.

8.9. Enfoque Interdisciplinario y Colaboración

Dado que la generación de mazmorras en videojuegos es un problema multifacético que combina conceptos de inteligencia artificial, diseño de juegos y experiencia del usuario, es fundamental destacar la importancia de un enfoque interdisciplinario en la investigación y el desarrollo en este campo. La colaboración entre diferentes disciplinas podría conducir a avances significativos en la generación de contenido en videojuegos, abriendo nuevas posibilidades y perspectivas innovadoras.

REFERENCIAS BIBLIOGRÁFICAS

[Alvarez *et al.*, 2018] Alvarez, A., Dahlskog, S., Font, J., Holmberg, J., y Johansson, S. (2018). Assessing aesthetic criteria in the evolutionary dungeon designer. En *Proceedings of the 13th International Conference on the Foundations of Digital Games*, pp. 1–4.

[Alvarez *et al.*, 2019] Alvarez, A., Dahlskog, S., Font, J., y Togelius, J. (2019). Empowering quality diversity in dungeon design with interactive constrained map-elites. En *Proceedings of the IEEE Conference on Games*, pp. 1–8. IEEE.

[Amato, 2017] Amato, A. (2017). *Procedural Content Generation in the Game Industry*, pp. 15–25. Springer International Publishing, Cham.

[Ashlock, 2015] Ashlock, D. (2015). Evolvable fashion-based cellular automata for generating cavern systems. En *Proceedings of the IEEE Conference on Computatational Intelligence in Games*, pp. 306–313. IEEE.

[Baghdadi *et al.*, 2015] Baghdadi, W., Eddin, F. S., Al-Omari, R., Alhalawani, Z., Shaker, M., y Shaker, N. (2015). A procedural method for automatic generation of spelunky levels. En *Proceedings of the Applications of Evolutionary Computation: 18th European Conference, EvoApplications*, pp. 305–317. Springer.

[Baldwin *et al.*, 2017] Baldwin, A., Dahlskog, S., Font, J. M., y Holmberg, J. (2017). Mixed-initiative procedural generation of dungeons using game design patterns. En *Proceedings of the IEEE conference on computational intelligence and games (CIG)*, pp. 25–32. IEEE.

[Cerny y Dechterenko, 2015] Cerny, V. y Dechterenko, F. (2015). Rogue-like games as a playground for artificial intelligence–evolutionary approach. En *Proceedings of the Entertainment Computing-ICEC 14th International Conference*, pp. 261–271. Springer.

[Dormans, 2010] Dormans, J. (2010). Adventures in level design: generating missions and spaces for action adventure games. En *Proceedings of the workshop on procedural content generation in games*, pp. 1–8.

[ESA, 2022] ESA (2022). Essential facts about the computer and video game industry. `https://www.theesa.com/resource/2022-essential-facts-about-the-video-game-industry/`. Accessed: 13 de noviembre de 2024.

[Gellel y Sweetser, 2020] Gellel, A. y Sweetser, P. (2020). A hybrid approach to procedural generation of roguelike video game levels. En *Proceedings of the 15th International Conference on the Foundations of Digital Games*, pp. 1–10.

[Green *et al.*, 2019] Green, M. C., Khalifa, A., Alsoughayer, A., Surana, D., Liapis, A., y Togelius, J. (2019). Two-step constructive approaches for dungeon generation. En *Proceedings of the 14th International Conference on the Foundations of Digital Games*, pp. 1–7.

[Holland, 1992] Holland, J. H. (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press.

[Lavender y Thompson, 2017] Lavender, B. y Thompson, T. (2017). A generative grammar approach for actionadventure map generation in the legend of zelda.

[Liapis, 2017] Liapis, A. (2017). Multi-segment evolution of dungeon game levels. En *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 203–210.

[Liapis *et al.*, 2015] Liapis, A., Holmgård, C., Yannakakis, G. N., y Togelius, J. (2015). Procedural personas as critics for dungeon generation. En *Proceedings of the Applications of Evolutionary Computation: 18th European Conference*, pp. 331–343. Springer.

[López-Ibáñez *et al.*, 2016] López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., y Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.

[Mariño y Lelis, 2016] Mariño, J. y Lelis, L. (2016). A computational model based on symmetry for generating visually pleasing maps of platform games. En *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volumen 12, pp. 65–71.

[Melotti y de Moraes, 2019] Melotti, A. S. y de Moraes, C. H. V. (2019). Evolving roguelike dungeons with deluged novelty search local competition. *IEEE Transactions on Games*, 11(2):173–182.

[newzoo, 2023] newzoo (2023). PC & Console Gaming Report 2023. `https://newzoo.com/resources/trend-reports/pc-console-gaming-report-2024`. Accessed: 13 de noviembre de 2024.

[Pereira *et al.*, 2021] Pereira, L. T., de Souza Prado, P. V., Lopes, R. M., y Toledo, C. F. M. (2021). Procedural generation of dungeons' maps and locked-door missions through an evolutionary algorithm validated with players. *Expert Systems with Applications*, 180:115009.

[Pereira *et al.*, 2018] Pereira, L. T., Prado, P. V., y Toledo, C. (2018). Evolving dungeon maps with locked door missions. En *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8. IEEE.

[Ruela y Delgado, 2018] Ruela, A. S. y Delgado, K. V. (2018). Scale-free evolutionary level generation. En *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8. IEEE.

[Shaker *et al.*, 2016] Shaker, N., Togelius, J., y Nelson, M. J. (2016). *Procedural content generation in games*. Springer.

[Smith y Mateas, 2011] Smith, A. M. y Mateas, M. (2011). Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200.

[Summerville *et al.*, 2017] Summerville, A., Mariño, J. R., Snodgrass, S., Ontañón, S., y Lelis, L. H. (2017). Understanding mario: an evaluation of design metrics for platformers. En *Proceedings of the 12th international conference on the foundations of digital games*, pp. 1–10.

[Summerville y Mateas, 2015] Summerville, A. y Mateas, M. (2015). Sampling hyrule: Sampling probabilistic machine learning for level generation. En *Proceedings of the Eleventh Artificial Intelligence and Interactive Digital Conference*.

[Summerville *et al.*, 2015] Summerville, A. J., Behrooz, M., Mateas, M., y Jhala, A. (2015). The learning of zelda: Data-driven learning of level topology. En *Proceedings of the FDG workshop on Procedural Content Generation in Games*, pp. 5–12.

[Togelius *et al.*, 2011] Togelius, J., Yannakakis, G. N., Stanley, K. O., y Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186.

[Viana *et al.*, 2022a] Viana, B. M., Pereira, L. T., Toledo, C. F., dos Santos, S. R., y Maia, S. M. (2022a). Feasible–infeasible two-population genetic algorithm to evolve dungeon levels with dependencies in barrier mechanics. *Applied Soft Computing*, 119:108586.

[Viana *et al.*, 2022b] Viana, B. M. F., Pereira, L. T., y Toledo, C. F. M. (2022b). Illuminating the space of dungeon maps, locked-door missions and enemy placement through map-elites.

[Weeks y Davis, 2022] Weeks, M. y Davis, J. (2022). Procedural dungeon generation for a 2d top-down game. En *Proceedings of the ACM Southeast Conference*, pp. 60–66.

[Whitehead, 2020] Whitehead, J. (2020). Spatial layout of procedural dungeons using linear constraints and smt solvers. En *Proceedings of the 15th International Conference on the Foundations of Digital Games*, pp. 1–9.

[Zafar *et al.*, 2020] Zafar, A., Mujtaba, H., y Beg, M. O. (2020). Search-based procedural content generation for gvg-lg. *Applied Soft Computing*, 86:105909.