Generación automática de contenidos para videojuegos mediante técnicas evolutivas

TRABAJO FIN DE GRADO
Ingeniería de Computadores/Ingeniería del Software
Dirigido por
Carlos Cervigón Ruckauer

Samuel Lapuente Jiménez
Álvaro Lázaro Sevilla

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Junio 2017

Documento maquetado con TEXIS v.1.0+.

Generación automática de contenidos para videojuegos mediante técnicas evolutivas

Memoria de Trabajo de Fin de Grado

Samuel Lapuente Jiménez
Álvaro Lázaro Sevilla

Dirigido por
Carlos Cervigón Ruckauer

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Junio 2017

A mi compañero y amigo Álvaro, por aguantarme en todo momento y realizar la mayor parte del trabajo.

A mi compañero y amigo Samuel, por aguantarme en todo momento y realizar la mayor parte del trabajo.

Agradecimientos

Primero, queremos agradecer a nuestras familias por aguantar estos años de esta travesía por el desierto que, al fin, acaban.
Agradecer a la cafetería, a Andrés, a Sánchez y a Richy por manternos con vida un poco más a base de comida, café y cerveza.
A los laboratorios, que aunque no tienen Github instalado y de los que nos han echado N + 1 veces, nos han permitido juntarnos y conseguir acabar este proyecto.
A StackOverflow, lugar de reunión de eruditos de la programación y fuente inagotable de conocimiento.

Ahora pongámonos serios, queremos agradecer a nuestro Director, Carlos, por todas las reuniones, información, correcciones y ayudas que nos ha prestado a lo largo del año.
También queremos agradecerle que aceptara este TFG propuesto y se uniese a nosotros en esta divertida aventura.

Por último, agradaceros a vosotros, queridos lectores, el que estéis echando un ojo a este trabajo, esperamos que lo disfrutéis.

Resumen

Cualquier tecnología suficientemente avanzada es indistinguible de la magia.

Arthur C. Clarke (1917-2008) Escritor inglés de ciencia ficción.

La computación evolutiva es una rama de la IA que engloba un conjunto de técnicas que, a través de la simulación de procesos naturales bioinspirados, son utilizados para la resolución de problemas complejos de búsqueda y aprendizaje.

Este trabajo presenta una serie de técnicas evolutivos aplicadas a la generación automática de contenidos en videojuegos. El objetivo de este Trabajo de Fin de Grado es automatizar procesos tediosos y repetitivos propios de un videojuego mediante el uso de estas técnicas y utilizarlas para crear un videojuego simple. Para ello hemos dividido el trabajo en dos bloques principales: un generador de mapas sobre los que se desarrollará el juego -formados por diferentes salas- y un generador de estrategias o Inteligencias Artificiales (IAs) para los enemigos contra los que se enfrenta el jugador en el videojuego.

Los mapas sobre los que se desarrolla el juego se generan utilizando un algoritmo evolutivo. La estructura de datos que se ha considerado utilizar para representar los mapas del videojuego es un grafo que representa el genotipo del individuo que haremos evolucionar. Las salas del mapa estarían representadas mediante los nodos del mismo, mientras que los pasillos que las unen serían las aristas.

Por otra parte, la IA de los enemigos se obtendrá utilizando Programación Genética, técnica evolutiva que permite evolucionar programas o

estrategias codificadas como expresiones.

También se presenta un Framework de Programación Genética que permite experimentar con las técnicas de generación de IAs, permitiendo modificar y ajustar cualquiera de los parámetros involucrados en el proceso.

Palabras clave

Las palabras son como monedas, que una vale por muchas como muchas no valen por una.

Francisco de Quevedo, Escritor español.

- Algoritmo evolutivo
- Algoritmo genético
- Mazmorra
- Inteligencia artificial
- Grafo
- Programación genética
- Árbol
- Selección
- Mutación
- Cruce

Siglas

La brevedad es el alma del ingenio.

William Shakespeare, Escritor inglés.

- IDE: Integrated Development Environment.
- AG: Algoritmo Genético.
- GA: Genetic Algorithm
- AE: Algoritmo Evolutivo.
- EA: Evolutionary Algorithm.
- SFML: Simple and Fast Multimedia Library.
- IA: Inteligencia Artificial.
- AI: Artificial Intelligence.
- PNJ: Personaje No Jugador.
- NPC: Non Playable Character.
- CC: Componente Conexa.
- 2D: Dos Dimensiones.
- GUI: Interfaz Gráfica de Usuario.

Abstract

Evolutionary computing is a branch of AI that includes a set of techniques that, through the simulation of natural processes and genetics, are used to solving complex problems of search and learning. These problems can be solved through EAs.

The aim of this project is to automate tedious process involved in a videogame using evolutionary techniques and design a simple video game using them. For this purpose an automatic map generator is used and some AIs will be generated representing the enemies in the game.

The maps will be generated through a GA. The data structure that has been considered to represent the video game maps is a graph. The rooms would be represented by the nodes, while the hallways that unite rooms would be the edges.

On the other hand, the AI of the enemies, will be managed through genetic programming. This algorithm is in charge of the behaviour, both before and after of detecting the player.

We have made a framework for genetic programming which allows to experiment and test different techniques for the AI generation, allowing to modify and adjust any parameter desired involved in the process.

Keywords

- Evolutionary algorithm
- Genetic algorithm
- Dungeon
- Artificial intelligence
- Graph
- Genetic programming
- Tree
- Selection
- Mutation
- Crossover

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

Lo último que uno sabe es por dónde empezar.

Blaise Pascal, Matemático y filósofo francés.

1.1. Motivación

A la hora de crear un videojuego, la mayoría de contenidos se generan de forma manual. Los escenarios, el comportamiento de los elementos controlados por el ordenador o la historia acostumbran a ser fijos. La distribución de los elementos de un mismo escenario, el comportamiento de un mismo enemigo o el argumento es siempre igual aunque se juegue varias veces.

En los últimos años han comenzado a utilizarse técnicas de generación automática de contenidos en la programación de videojuegos. Particularmente, nosotros vamos a diferenciar entre las técnicas utilizadas para la generación de escenarios y las utilizadas para la IA de los enemigos.

Destaca la Generación Procedural como método de generación automática de contenido en un videojuego, basada en el uso de algoritmos. Estos contenidos pueden ser desde los escenarios hasta otros más complejos como las armas o los objetos que se pueden encontrar. En los últimos años, está extendiéndose su uso sobre todo en la generación automática de escenarios. Ejemplos de esto último pueden encontrarse en videojuegos como *No*

Man's Sky (Games, 2016) o *Minecraft* (AB, 2011). Un ejemplo en la generación de botín procedural es el del videojuego *Borderlands* (Software, 2009). A menudo, este tipo de generación se basa en una semilla que determina cómo se crearán dichos elementos.

En cuanto a la inteligencia artificial, se empezó a hablar de ella en los años 50. Hay dos personas consideradas como los padres de la IA:

- Alan Turing ideó un sistema para determinar si una máquina puede hacerse pasar, de forma indistinguible, por un ser humano. Es el llamado "test de Turing" (Matemática Aplicada y Estadística, 2004).
- John McCarthy acuñó el término y consiguió muchos avances en dicho campo (Guillén Torres, 2016), entre los cuales se encuentra la creación del lenguaje LISP (EcuRed, 2016).

En materia de inteligencia artificial, los avances en generación automática de contenidos son más complicados de conseguir. Nuestra dificultad ha surgido al intentar emular el comportamiendo de un jugador humano que nos permitiese simular un entorno de juego para la función de evaluacion. En los problemas de generación de IAs, hemos observado que casi siempre se suele simular el comportamiendo del jugador, creando alguna estrategia para superar el juego. Algunos ejemplos son el *Unreal Tournament* (Games, 1999), el *Pacman* (Namco, 1980) o el *Space Invaders* (Corporation, 1978), que serán detallados más adelante.

La motivación de este trabajo es el intento de incorporar técnicas evolutivas a la generación de contenido para videojuegos que, a diferencia de la generación procedural, son más sencillas de implementar y menos costosas. Nuestra intención es analizar el uso de este tipo de técnicas permitiendo generar los aspectos principales de un videojuego, como los escenarios, la IA de los enemigos, la música o la historia. La motivación última sería la de conseguir generar un videojuego perfectamente funcional teniendo que implementar el menor contenido posible.

1.2. Objetivos

El objetivo principal de este trabajo es aportar técnicas que permitan automatizar el desarrollo de un videojuego, en nuestro caso particular,

un roguelike. Los videojuegos de este estilo están inspirados en el videojuego *Rogue* (Toy, Wichman y Arnold, 1980) que es un juego de exploración de mazmorras (Muñoz, 2014), donde cada nivel se crea aleatoriamente. Partiendo de ahí, nos planteamos los siguientes objetivos específicos:

- Generación completa de los mapas de un videojuego mediante un AG.
- Generación de la IA de los enemigos mediante programación genética, siendo ésta la única responsable del comportamiento y obtención de datos de las mismas en todo momento.
- Conseguir un videojuego divertido. De nada vale obtener técnicas para crear videojuegos si estas no cumplen el objetivo principal que es divertir al jugador. Por ello, dividimos el enfoque en dos grandes partes. En la parte de los mapas o mazmorras, proporcionar un objetivo al jugador y que este no sea ni muy fácil ni muy difícil. En la parte de las IAs intentamos conseguir que fueran divertidas y no demasiado evidentes. Con esto queremos decir que supongan un reto asequible para el jugador y que no hagan cosas que nos muestre de forma evidente que es un ordenador el que las está controlando. El reto asequible debe estar en un nivel de dificultad que suponga un reto no frustrante.
- La IA debe ser capaz de cumplir sus objetivos mediante el uso único de su árbol de decisión. En el árbol tendrá instrucciones que le permitirán obtener la información que necesita. Con esto queremos evitar otorgarle una ventaja a la IA, ya que no obtiene nada de información extra. Esto nos supone una mayor dificultad pero queríamos ver hasta qué punto era posible conseguir una IA correcta.
- Desarrollar un juego en 2D como prueba de concepto.

1.3. Estructura

A continuación se explica brevemente la estructura de la memoria:

Capítulo 1. Introducción

En este capítulo se incluyen la motivación y los objetivos que queremos cumplir con este trabajo.

Capítulo 2. Trabajo relacionado

En este capítulo se realiza una exposición de otros trabajos relacionados con este que nos ocupa, detallando las diferencias y similitudes, así como nuestra aportación al estado del arte.

Capítulo 3. Computación evolutiva y Programación genética

En este capítulo se detallan los conceptos de computación evolutiva y programación genética, así como todos los aspectos relacionados con ellos.

Capítulo 4. Técnicas para la generación de mazmorras

En este capítulo se profundiza en las técnicas utilizadas para la generación de mazmorras mediante técnicas evolutivas.

Capítulo 5. Técnicas para la generación de la inteligencia artificial

En este capítulo se explica el procedimiento que se ha llevado a cabo para la creación de la IA de este trabajo.

Capítulo 6. Framework de pruebas

En este capítulo se expone en qué consistirá el framework que hemos creado para la realización de las pruebas de este proyecto.

Capítulo 7. Videojuego

En este capítulo se explica en qué consiste el videojuego final que surge tras la realización de este proyecto.

Capítulo 8. Trabajo individual

En capítulo se detalla, dividido entre los 3 integrantes de este trabajo, el trabajo individual que ha realizado cada uno.

Capítulo 9. Conclusiones y trabajo futuro

En este capítulo se exponen las conclusiones que se han sacado tras realizar este trabajo, así como las posibilidades que existen de ampliarlo en el futuro.

Herramientas utilizadas

En este apartado comentamos brevemente las herramientas que hemos usado para el desarrollo de este proyecto, explicando para qué se ha utilizado cada una.

Bibliografía

Por último, enumeramos todas las referencias bibliográficas que hemos tomado a lo largo de la elaboración de este trabajo incluyendo libros, artículos, páginas web de interés u otros trabajos.

1.4. Motivation

When a videogame is made, its contents are created mostly directly by humans. The scenes, the behaviours or the history are usually fixed. The distribution of the elements in a scene, the AI behaviour or the story tend to be the same in every run played.

In the last few years some techniques of automatic generation are being used for game programming. In this memo, we are going to distinguish among the techniques used for scenes and map generation and the AI of the NPC.

On the one hand, procedural generation is a go-to method in the automated generation content for video games. It is based in algorithms. Those contents go from scenes to other much more complex like drops or loot. In the last years, its use is growing mainly for the scene generation. Some examples of these mention above are games like No Mans sky or Minecraft. An example of procedural generation for loot is Borderlands. This kind of automated generation is frequently based in a seed which determines the way this elements are made.

On the other hand, for the AI, we must go to the origins. The AI was first mentioned in the early 50s. There are two man responsible for this:

- Alan Turing invented a test which determines if a computer can not be distinguished from a human being. This test is called The Turings test.
- John McCarthy mentioned for the first time the term Artificial Intelligence and achieved a lot of progress in this field. Between those is placed the invention of the LISP language.

With regard to AI, it is harder to obtain advances, at least in our case. This is a consequence of simulating a human. The goals being achieved are often coming from evolving an existing AI making those compete between differents AIs. Some examples are unreal tournament, pacman or space invaders.

With all the above in mind, the main goal of this project is aim to include evolution techniques for the automated generation of content. This is easier than procedural generation and less costly. We intend to use this techniques and generate the scenes, the AI or even the music or the story. The final motivation would be generating a video game implementing manually the less content possible.

1.5. Goals

The main goal of the project is to give techniques which allows to automatize the development of a game, in our particular case, a roguelike. Games like this are inspired in the video game Rogue, a game of dungeon exploration, where every level is created randomly. Once a dungeon is com-

pleted, the player advances to the next one. From this perspective, we decided to achieve the following goals:

- Generate elements of a game using evolution techniques and assess how efficient they are. They must be efficient in time as well as in quality. We decided to do like this:
  - Complete generation of the scenes by GA.
  - Generate the AI by genetic programming. This AI must be entirely generated by the program.
- The elements generated have to be fun. It is worthless being able to generate automated content for video games if they are not fun. For the scene part, the main target was to give the player a reason to play and worth trying to achieve it. For the AI we tried to make them fun and not to obvious. With this statement we want to say that the AI must not be too hard either too easy. They also must not be too clear that they are executing constantly a decision tree.
- Develop a 2D game as a proof of concept.

1.6. Structure

In the next few lines we sum up the structure and content of this memo.

Chapter 1. Introduction

In this chapter we include the motivations and objectives we want to achieve in this project.

Chapter 2. Related work

In this chapter it is exposed a brief description of related work, detailing the differences and similarities as well as our contribution to the state of art.

Chapter 3. Evolution programming and genetic programming

In this chapter the main concepts of genetic programming are detailed.

Chapter 4. Dungeon generation techniques

In this chapter we explain deeper the techniques used for the dungeon genetic generation algorithms.

Chapter 5. Artificial intelligence generation techniques

In this chapter we explain the procedure and tests que have to do in order to develop the AI.

Chapter 6. Framework

In this chapter we detail the framework we have developed using SFML which we used for testing the AIs. This framework can be extended for other types of genetic program problems.

Chapter 7. Video game

In this chapter we explain the final video game developed with this techniques.

Chapter 8. Individual work

In this chapter it is detailed the individual work each one of us has done during this project.

Chapter 9. Conclusions and future work

In this chapter our final conclusions are exposed as well as the possibilities of expanding it.

Tools

In this chapter we describe briefly the tools we have used for developing this project.

Bibliography

Finally, we enumerate all the bibliographic references we have taken while we have worked on this project. This bibliography includes books, articles, web pages and other papers.

Capítulo 2

Trabajo relacionado

Me gusta y me fascina el trabajo. Podría estar sentado horas y horas mirando a otros cómo trabajan.

Jerome K. Jerome, Humorista inglés.

Para analizar el estado del arte de este proyecto, tenemos que tener en cuenta 3 aspectos fundamentales:

- La generación procedural de contenido para videojuegos.
- La generación de contenido para videojuegos mediante algoritmos evolutivos.
- Las diferentes implementaciones de IAs existentes.

2.1. Generación Procedural

Generación procedural es el método de creación de contenidos a través de algoritmos, en oposición a un método de creación manual, y se aplica tanto en simulaciones gráficas por computadora como en videojuegos, instalaciones, programación y en música. Los fractales son un ejemplo de animación mediante generación procedural: funciones matemáticas gráficas repetidas hasta el infinito. El sonido digital también se puede generar

de forma procedural, un uso que se ha desarrollado mucho para la música electrónica.

En videojuegos e instalaciones la generación procedural se utiliza en función de que el contenido se genere en la computadora que los contiene, en tiempo real, y no de manera previa y renderizado en paquetes gráficos predefinidos. Se utiliza principalmente para la generación de ambientes y mapas, aunque también se aplica para IA y jugabilidad. Se utiliza para generar de manera rápida y detallada espacios infinitos y patrones de singularidad en objetos, simulaciones y personajes. También se utiliza para la creación de sistemas de partículas para agua, fuego y gases; o para generar un sinfín de actores digitales únicos y diferentes como reemplazo de extras. Sin embargo, no se suele utilizar para el contenido final, el cual en general se muestra ya preestablecido. En (Gamasutra, 2014) se muestra una técnica para generar mazmorras aleatorias. Los pasos para ello son:

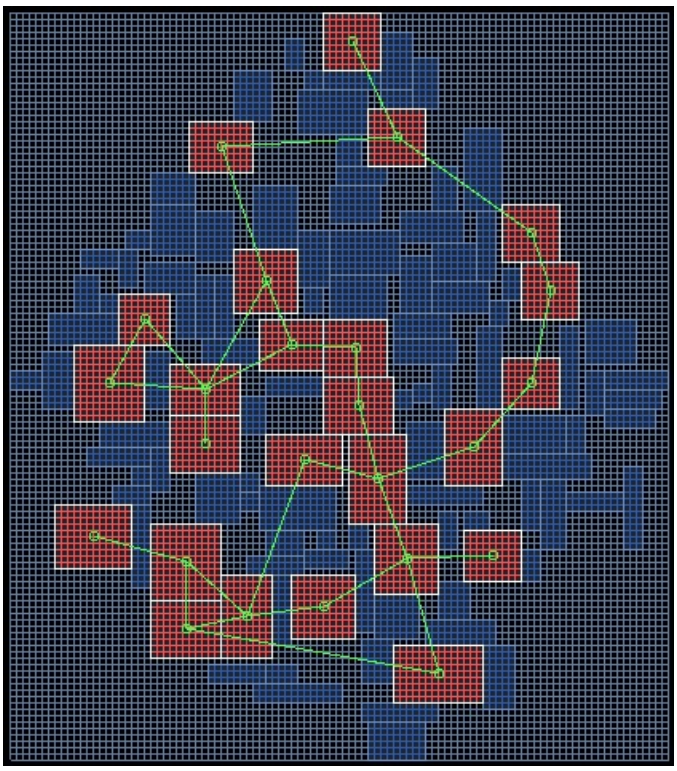

Figura 2.1: Escenario creado mediante Generación Procedural

- Generar una serie de salas con una determinada anchura y altura y colocarlas aleatoriamente dentro de un círculo.

- Separar las diferentes salas aplicando físicas de colisión, hasta que no se superpongan.
- Después, para escoger qué salas son las principales, se tendrán en cuenta aquellas que estén por encima de 1.25 en el ratio ancho/alto.
- Tomar los puntos medios de las salas principales y aplicarles el Procedimiento de Delaunay (Departamento de Matemática Aplicada, 2015) para obtener un grafo a partir de los triángulos obtenidos.
- A partir del grafo, se debe obtener el árbol de expansión mínimo (Algorithms y more, 2012). Esto hará que todas las habitaciones principales sean accesibles, pero que no estén conectadas directamente.
- Finalmente, se añaden los pasillos a la mazmorra. Si los nodos están cerca horizontalmente, se añade una línea horizontal. Lo mismo para los nodos cercanos verticalmente. Si no es así, se añaden dos líneas que formen una L.

2.2. Técnicas Evolutivas

Existen proyectos que, mediante gramáticas evolutivas, generan historias de fondo que aportan consistencia a la experiencia de juego, con la intención de evitar que éstas deban ser ideadas por desarrolladores de forma manual (García-Ortega y García-Sanchez, 2014).

Existen herramientas que permiten utilizar algoritmos genéticos para generar mazmorras en base a unos parámetros de diseño (Font, Izquierdo, Manrique y Togelius, 2016), que tienen que venir definidos previamente.

En este sentido cabe reseñar el ejemplo explicado en (Togelius, Preuss, Beume, Wessing, Hagelbäck, Yannakakis y Grappiolo, 2013), que cuenta cómo se pueden desarrollar funciones heurísticas que miden propiedades de los mapas y comprueban si afectan a la experiencia de juego. Se diseñaron dos representaciones diferentes de mapas, una para un juego de estrategia genérico y otro para el videojuego Starcraft. Mediante este método se puede automatizar completamente la generación de mapas o como herramienta de apoyo para diseñadores humanos.

Otro ejemplo más es el referido en (Pérez, Togelius, Samothrakis, Rohlfshagen y Lucas, 2013) que analiza 3 subcomponentes diferentes en el proceso de generación de mapas y propone soluciones a todos ellos. Los mapas

se representan mediante 2 conceptos distintos, la geometría y la disposición del contenido. La geometría se lleva a cabo tanto en el exterior como en el interior conectando segmentos de mapas pre-hechos para formar mapas más grandes y completos. La disposición del contenido a través del mapa se determina mediante el uso de características de la geometría como el uso de una Red de Producción de Patrones de Composición. Por último están las preferencias del jugador para diseño del contenido, que se capturan y se utilizan en un marco de sistema recomendador. Todas las soluciones se combinan y se prueban en el videojuego de acción y disparos *Angry Bots* (Unity, 2011) y se evalúan en un experimento a gran escala.

Existe un precedente de uso de AGs para evolucionar el comportamiento de los bots -enemigos jugadores controlados por el ordenador- en un videojuego más o menos actual, concretamente el *Unreal Tournament 2004* (Bullen y Katchabaw, 2004).Los resultados demostraron una mejora considerable en el rendimiento de los bots evolucionados, con respecto al grupo de control.

En otra ocasión se usó programación genética para la evolución de estrategias defensivas. En (Jackson, 2005) se utiliza la programación genética para evolucionar estrategias en un simple juego que emula los invasores del espacio 2.2.

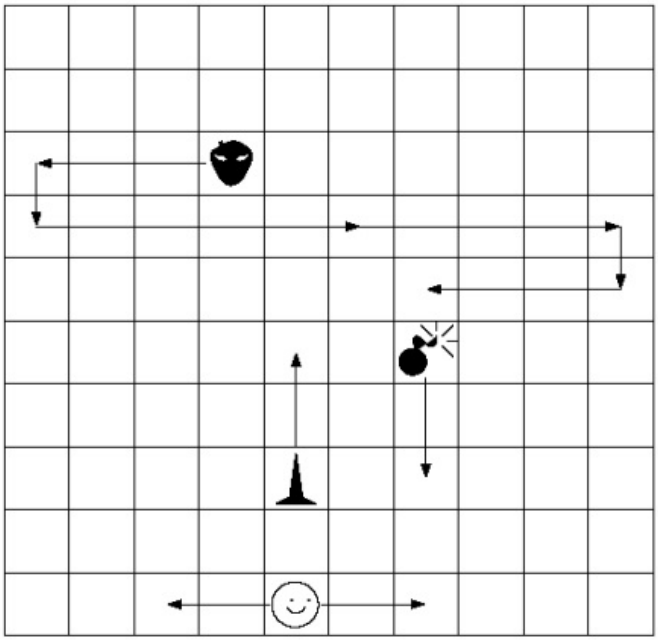

Figura 2.2: Emulador de *Space Invaders*

Para la función de evaluación se realizaron un conjunto de pruebas. En cada una, el defensor se colocaba en la parte inferior del mapa, mientras que los oponentes se colocaban en cualquiera de las filas superiores, desplazándose inicialmente en una dirección elegida al azar. Después, irían descendiendo de izquierda a derecha, bajando cuando llegasen a los límites laterales del tablero y lanzando bombas de vez en cuando. Se utilizaron los operadores genéticos más comunes.

Se demostró que mediante programación genética es posible desarrollar estrategias de defensa para hacer frente a juegos de una determinada dificultad. Esas estrategias incorporan muchos de los aspectos de la evasión, la búsqueda y la conducta de orientación que se encuentran en jugadores humanos.

Se define un conjunto de elementos terminales:

$$T = \{IZQUIERDA, DERECHA, FUEGO, DIST_Y, DIST_X\}$$

y un conjunto de funciones:

$$O = \{IF, EQ, PROGN2, PROGN3\}$$

Después de un proceso evolutivo se obtienen soluciones representadas de la siguiente forma:

$$(IF(EQ(DIST_Y\, DIST_X))$$

$$(PROGN3(FUEGO IZQUIERDA DIST_X))$$

$$(PROGN2(IZQUIERDA IZQUIERDA)))$$

El significado de esa expresión es el siguiente:

- Si la distancia en Y es igual que la distancia en X, se ejecuta $PROGN3(FUEGO\ IZQUIERDA\ DIST_X)$, que disparará, avanzará a la izquierda y calculará la distancia horizontal hasta el defensor.

- Si no es igual, se ejecuta $PROGN2(IZQUIERDA\ IZQUIERDA)$, que avanzará dos casillas hacia la izquierda.

Por otro lado, combinando el uso de algoritmos genéticos con Grafos tenemos el problema del coloreamiento mínimo de un grafo (Universidad de Oviedo, 1997). Se han de utilizar el menor número de colores cumpliéndose la restricción siguiente: Dos nodos adyacentes no pueden tener el mismo color.

Se trata de un problema NP-completo. No hay un método que asegure que se va a obtener el coloreamiento óptimo para todos los tipos de grafos que se presenten. Esto se debe a que con sólo añadir alguna arista o nodo a un grafo dado, la solución cambia radicalmente.

En este problema, un individuo es una permutación de todos los nodos del grafo. La población de individuos se selecciona mediante el método de la Ruleta. Dada la facilidad para evaluar si hay que añadir un color extra o no, los individuos suelen ser bastante buenos.

Dada una permutación de nodos de un grafo, hay que colorear dicho grafo recorriendo en orden los nodos, asignándoles a cada uno el primer color que puede tener cumpliendo las restricciones. La función de Fitness busca obtener el mínimo número de colores necesarios para el coloreamiento completo del grafo. Para la reproducción de los individuos se puede utilizar cruce monopunto, mientras que la mutación se puede cambiar de orden dos nodos del individuo.

2.3. Inteligencia Artificial

En un videojuego, se define la inteligencia artificial -IA- como la simulación de inteligencia -comportamiento, modo de actuar, etc.- de los PNJs y que normalmente incluirá algo relacionado con los siguientes aspectos:

- Conseguir comportamientos muy básicos en los PNJs -coger un elemento, golpear, etc-
- Moverse a través de unos escenarios en los que pueden aparecer obstáculos.
- Tomar decisiones para saber qué acciones realizar y en qué orden.

Hay diversas maneras de implementación de estas IAs (Alcalá):

- Algoritmos minimax: se utiliza normalmente en juegos de tablero como el Ajedrez o las damas. Está basado en la prueba de todas las posibilidades de jugada de cada jugador. Se puede optimizar utilizando la poda alfa-beta.
- Algoritmos de búsqueda de camino (Dijkstra, A*). Búsqueda del camino más corto para llegar de un punto A a otro punto B. A* es una combinación de recorridos de tipo *primero en anchura* con *primero en profundidad* y garantiza encontrar siempre el camino óptimo. Dijkstra, por su parte, determina en un grafo valorado el camino más corto para llegar desde un vértice a todos los demás del grafo.
- Agentes inteligentes: se aplica cuando los PNJs perciben mediante sensores la existencia de algún elemento extraño en el entorno -pared, esquina donde hay que girar, un enemigo- y actúa consecuentemente -dándose la vuelta, girando, pasando a modo ataque- con la mejor acción posible, gracias a unos *actuadores* que le indican cómo debe hacerlo. Dos tipos:
  - Reactivo: actúa promovido por un cambio en el entorno (Guardián de algún tesoro).
  - Proactivo: decide actuar antes de que se produzcan los sucesos (Jefe final de un juego).
- Máquinas de estados finitos: entidad abstracta que está formada por diferentes estados y las transiciones que se producen entre ellos. Estas transiciones están producidas por cambios que se van produciendo en el entorno. Según el estado en se encuentre, se pueden llevar a cabo una serie de acciones.
- Redes neuronales: están inspiradas en el comportamiento de las neuronas y conexiones del cerebro humano, tratando de crear programas capaces de solucionar problemas difíciles, actuando como haría un humano. Necesitan entrenamiento con muchos ejemplos.
- Algoritmos evolutivos y Programación genética: Son sistemas muy robustos que resuelven problemas de optimización, donde los individuos más capaces son los que finalmente sobreviven, tras aplicarles una serie de transformaciones.

2.4. Nuestra aportación

Nuestro proyecto pretende analizar el rendimiento de los AGs, y determinar si es viable usarlos como mecanismo de generación automática de elementos de un juego sin necesidad de un diseño previo, así como estudiar la calidad de los elementos generados y la dificultad que puede conllevar establecer unos buenos parámetros para estos algoritmos.

En la parte de generación de los mapas hemos optado por representar cada individuo mediante un grafo que representa el esqueleto de nuestra mazmorra. Se aplica un algoritmo evolutivo para transformar una de las CCs del grafo hasta obtener una mazmorra satisfactoria.

Dada la pequeña cantidad de proyectos y documentación en relación a la generación automática de IAs en videojuegos, nuestro proyecto trata de hacer un análisis sobre la posibilidad de aplicar técnicas evolutivas en este aspecto. De forma análoga al proyecto MADE (García-Ortega y García-Sanchez, 2014), intentaremos automatizar los procesos de desarrollo de elementos principales de un videojuego, centrándonos en analizar aspectos clave, como la calidad de los elementos generados y el tiempo que conlleva generarlos.

En este sentido, hay que tener en cuenta la generación de 2 árboles diferentes (patrulla y ataque) y las transiciones entre estados de uno y otro árbol, siendo en cierta manera cada uno de los estados un árbol diferente (exploración y ataque, como comentábamos antes) 2.3.

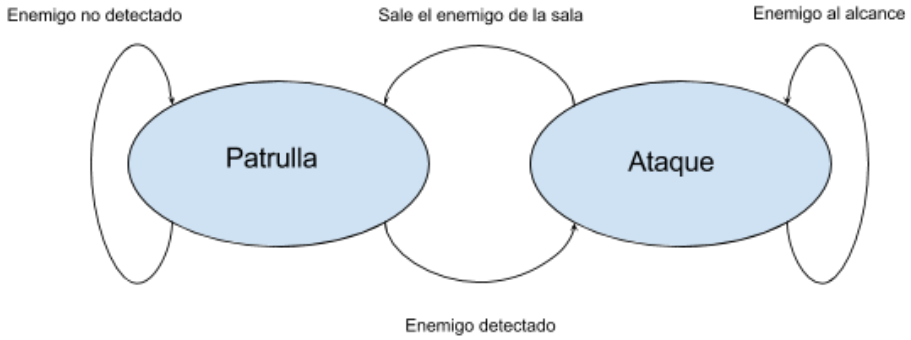

Figura 2.3: Ejemplo de máquina de estados y decisiones

Capítulo 3

Computación evolutiva y Programación genética

Todas las piezas deben unirse sin ser forzadas. Debe recordar que los componentes que está reensamblando fueron desmontados por usted, por lo que si no puede unirlos debe existir una razón. Pero sobre todo, no use un martillo.

Manual de mantenimiento de IBM, año 1925

3.1. Computación evolutiva

3.1.1. Antecedentes biológicos

Durante mucho tiempo, la teoría que se creyó válida acerca del origen de las especies fue el creacionismo: Todas las especies del planeta fueron creadas por Dios. Posteriormente, Georges Louis Leclerc y Jean-Baptiste Lamarck enunciaron sendas teorías que rebatían completamente la teoría creacionista, pero sin llegar a concretar bien cómo se desarrollaba este proceso de evolución. Más tarde, Charles Darwin indicó que la evolución se origina por medio de cambios aleatorios en rasgos hereditarios, combinados con un proceso de selección natural. Otros científicos, como August Weismann, Johann

Gregor Mendel y James Mark Baldwin, completaron la teoría Darwinista, diciendo que había células que podían transmitir información hereditaria, mientras que otras no. Llevaron a cabo un experimento con ratas que demostraba que, no por el hecho de perder una extremidad, los descendientes nacen sin ella; o proponiendo que, si el aprendizaje ayuda a la supervivencia, tendrán más descendencia los individuos con más capacidad de aprendizaje. Todo ello, dio origen al llamado Neo-Darwinismo.

Actualmente, el pensamiento evolutivo se explica mediante el Neo-Darwinismo, que promulga que hacen falta cuatro procesos para explicar toda vida en el planeta:

- Seleccción
- Reproducción
- Mutación
- Competencia

3.1.2. Representación

La computación evolutiva es una manera de solucionar problemas complejos, a través de la búsqueda y el aprendizaje, por medio de procesos inspirados en la evolución natural. La extensión de dichos modelos a la computación general es conocida como computación evolutiva. Dichos algoritmos tienen como propósito general transformar una determinada estructura, pasando por una serie de transformaciones de forma iterativa, para que ésta vaya siendo cada vez más apta para cumplir un determinado cometido.

Los AGs son parte fundamental dentro de la computación evolutiva. Son una técnica de búsqueda y optimización inspirada en el proceso genético de los seres vivos. Con el paso de las generaciones las poblaciones van evolucionando y prevalecen los más fuertes o más adaptados al medio. Las poblaciones se representan mediante cadenas binarias y cada uno de sus miembros tiene un número determinado de posiciones, a las que se denominan Genes.

En un AG, se seleccionan un número determinado de individuos, pudiendo utilizar diferentes métodos para hacer la selección, siendo los individuos seleccionados los progenitores. Estos progenitores se cruzan con otros y son sometidos a un proceso de mutación de los que surge la descendencia,

que debe recombinarse con la población inicial. Este proceso se puede repetir las veces que se crea conveniente.

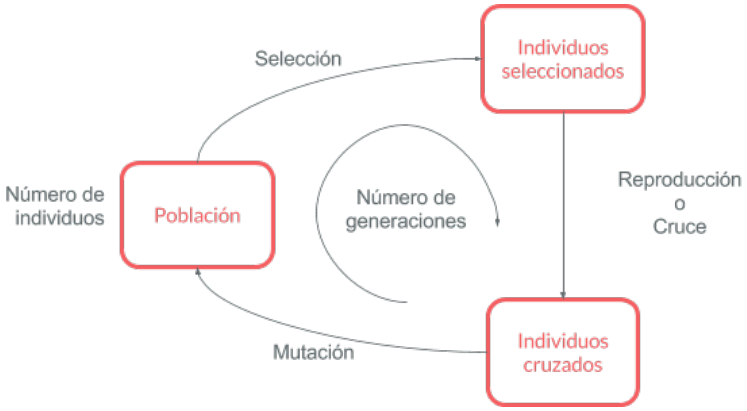

Figura 3.1: Pasos en un algoritmo evolutivo

Un Algoritmo Evolutivo (AE) tiene la misma estructura básica del AG, pero también consta de algunas diferencias:

- La representación es natural, en lugar de binaria, y está más próxima al dominio real del problema.
- Pueden ser utilizadas estructuras de datos más complejas, como vectores, listas o árboles.
- Operadores genéticos específicos para la representación, además de los propios de los AGs.

3.1.3. Operadores genéticos

Los operadores genéticos son los métodos que emplean los algoritmos genéticos y los algoritmos evolutivos para que se pueda mantener la diversidad de la población. Los principales operadores genéticos existentes son:

- Operadores de selección: Son los encargados de elegir cuáles individuos son los que tendrán oportunidad de reproducirse, cruzarse con otros o ser mutados para pasar a la siguiente generación.

- Operadores de cruce o recombinación: Se trata de una recombinación de genes entre los individuos seleccionados, para producir la descendencia que pasará a la siguiente generación.
- Operadores de copia: Se trata de una estrategia de reproducción a partir de la generación anterior similar a la de cruce. La diferencia es que en ésta sólo se copia el individuo padre, pasando tal cual a la generación siguiente.
- Operadores de mutación: La mutación consiste en variar el valor de alguno de los genes (generalmente sólo uno) de forma aleatoria.

En este Trabajo de Final de Grado se utilizarán los operadores genéticos de selección, cruce y mutación.

3.1.3.1. Operadores de selección

Los operadores de selección principales que se utilizan son:

- *Selección por Ruleta:* Para realizar la selección, se debe ajustar la adaptación de cada individuo para que la suma de todos ellos sea 1. Una vez ajustada, se reparte el segmento de forma proporcional. Un número aleatorio entre 0 y 1 determinará qué individuo será seleccionado. Para saber qué individuo es el seleccionado, cada uno debe tener no solo su adaptación, sino también su adaptación acumulada, de forma que si el número aleatorio se encuentra entre la puntuación acumulada del individuo 'i' y la del 'i+1', entonces el individuo 'i' será seleccionado. Con este método, cabe la posibilidad de que un individuo sea seleccionado más de una vez. De la misma forma, estadísticamente, los individuos con menos adaptación tienen mayor probabilidad de quedar fuera de la población seleccionada. Este proceso se repite con tantos números aleatorios como el tamaño de la población.
- *Selección Estocástica universal:* Se realiza una distribución idéntica al método de selección por ruleta, pero esta vez sólo se genera un número aleatorio. Un número 'i' será el punto inicial a partir del cual se irán seleccionando los individuos, este número estará comprendido entre 0 y 1/N, siendo N el tamaño de la población. El ratio 'r' por el cual serán seleccionados los individuos será 1/N.

- *Selección por Torneo:* un tipo de selección más elitista. Se van seleccionando pequeños grupos de individuos, y se escoge el que mejor adaptación tenga. El proceso repite hasta completar la población. Existen otras variantes, en las que se puede elegir de forma probabilística al menos adaptado.
- *Selección por Ranking:* Se otorga un valor a los individuos que hace que éstos sean ordenados decrecientemente. Este valor asignado sólo depende de su ranking, que puede no depender directamente de su valor objetivo.

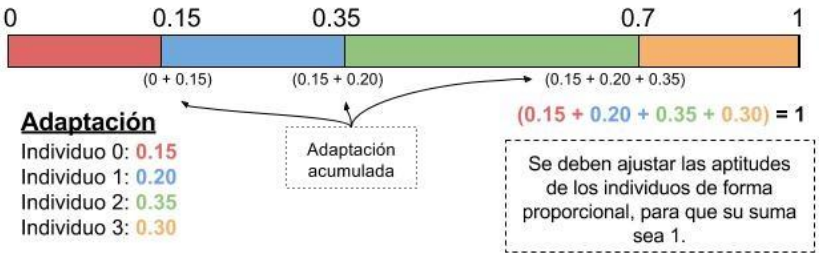

Figura 3.2: Ejemplo de selección por ruleta

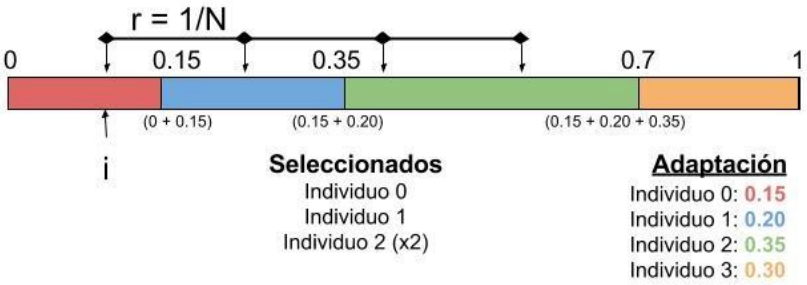

Figura 3.3: Ejemplo de selección estocástica

3.1.3.2. Operadores de cruce

Los principales operadores de cruce existentes son:

- *Cruce monopunto:* Se divide en dos partes a los individuos que se van a cruzar y luego se combinan, generando dos nuevos individuos con una parte de cada progenitor.3.4.
- *Cruce multipunto:* Similar al corte monopunto, pero los individuos se dividen en más de dos partes. Una vez cortadas, se combinan generando los nuevos individuos con al menos una parte de cada progenitor.
- *Cruce uniforme:* En este caso se decide con una probabilidad Pi (normalmente del 50
- *Cruces específicos con representación no binaria:* Los siguientes operadores de cruce no pueden utilizarse con representación binaria, pues requieren de operaciones que darían un resultado real:

  - *Cruce aritmético:* Se realiza una combinación lineal de los genes de los padres. Una forma de hacer es que el gen de los hijos se genere haciendo una media aritmética de los genes de los padres. Dado que, de los 2 padres, sólo se genera un gen para un hijo; se puede perder población. Por esa razón, tras haber obtenido un hijo, éste se suele cruzar con uno de los padres para obtener al segundo.
  - *Cruce geométrico:* Se realiza la raíz cuadrada del producto de los genes de los padres. Presenta el mismo problema de pérdida de población que el anterior.
  - *Cruce SBX:* El cruce SBX favorece que se generen individuos cercanos a los padres, si la diferencia entre ellos es pequeña. Está basado en el cruce monopunto binario.

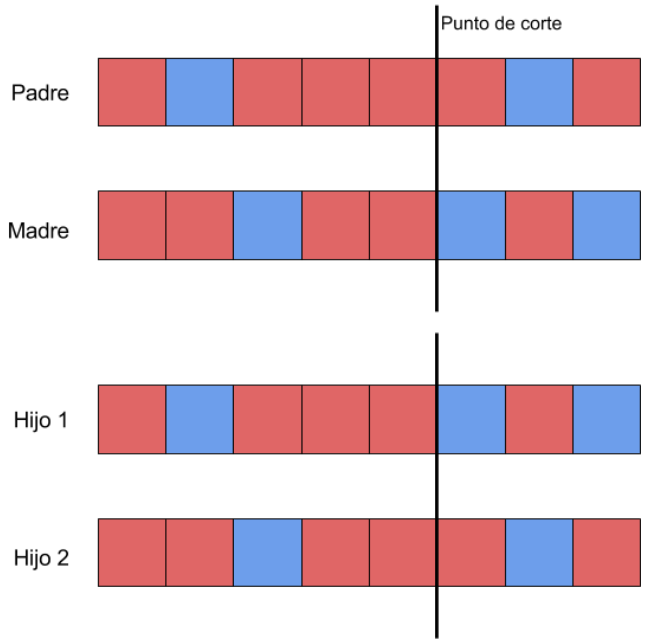

Figura 3.4: Ejemplo de cruce monopunto

3.1.3.3. Operadores de mutación

Los operadores de mutación que se utilizarán en este proyecto serán explicados en cada una de las secciones siguientes, porque serán diferentes según se trate de la parte de generación de mazmorras o de la IA 3.5.

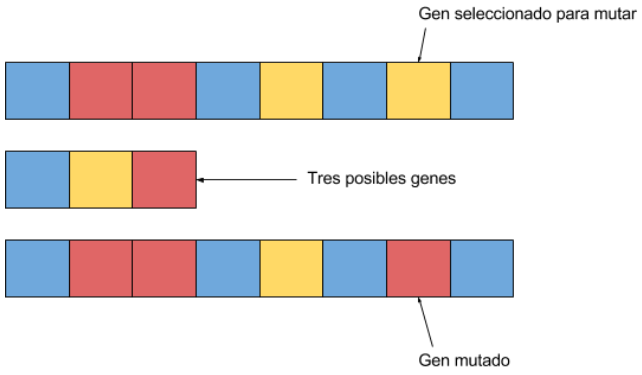

Figura 3.5: Ejemplo de mutación

3.1.4. Función de Fitness

En computación evolutiva, el concepto de Fitness se asocia a la capacidad que tiene un individuo de sobrevivir y reproducirse. Para evaluar esa capacidad se asigna un valor a cada individuo que determinará su aptitud. Lo que se pretende con esto es que, con el paso de las generaciones, sean los individuos con mejor Fitness los que sobrevivan.

3.1.5. Mejoras en el algoritmo evolutivo

3.1.5.1. Elitismo

Se selecciona, en un pequeño porcentaje, a los mejores individuos de cada generación, permitiéndoles pasar intactos a la generación siguiente.

3.1.5.2. Contractividad

Proceso utilizado en los AGs que consiste en, si en una generación los individuos no mejoran lo suficiente, no se tiene en cuenta. Este proceso se hace un número finito de veces y, si en ese tiempo no se ha mejorado, se detiene el algoritmo, ya que podemos suponer que la mejora ha alcanzado su punto de convergencia.

3.2. Programación genética

La programación genética es una técnica evolutiva en la cual los individuos representan expresiones codificadas en forma de árbol.

John Koza es considerado el padre de la programación genética (Koza, 1992). Él definió un AG de la siguiente manera:

"Es un algoritmo matemático altamente paralelo que transforma un conjunto de objetos matemáticos individuales con respecto al tiempo usando operaciones modeladas de acuerdo al principio Darwiniano de reproducción y supervivencia del más apto, y tras haberse presentado de forma natural

una serie de operaciones genéticas de entre las que destaca la recombinación sexual. Cada uno de estos objetos matemáticos suele ser una cadena de caracteres (letras o números) de longitud fija que se ajusta al modelo de las cadenas de cromosomas, y se les asocia con una cierta función matemática que refleja su aptitud."

3.2.1. Representación

Cada individuo de la población se representa en forma de árbol con las siguientes características:

- Los nodos internos del árbol representan las funciones, que son las operaciones que se llevan a cabo. Estas operaciones pueden ser condicionales o la ejecución de otras operaciones en secuencia.
- Las hojas del árbol representan símbolos terminales, que son las acciones que se pueden ejecutar cuando indiquen las operaciones. Estos elementos terminales pueden ser del tipo avanzar, girar o disparar.

Estos árboles representan expresiones que combinan elementos terminales y no terminales

Para llevar a cabo un problema de programación genética se deben seguir estos pasos:

- Identificar los elementos terminales, es decir, los elementos por los cuales el árbol no se seguirá expandiendo.
- Identificar bien las funciones que vamos a necesitar y el número de elementos en los que aplicará (aridad).
- Reconocer la función de adaptación que vamos a necesitar.
- Establecer los parámetros del algoritmo.
- Determinar bajo qué criterios consideraremos que deberá terminar la ejecución del algoritmo.

3.2.2. Métodos de inicialización

Existen varios métodos para inicializar los individuos de la población cuando el genotipo de los individuos es un árbol.

- *Creciente:* Puede haber cualquier tipo de nodo en cualquier profundidad sin superar la profundidad máxima.
- *Completa:* Se fija una profundidad máxima del árbol y se van generando nodos no terminales hasta que se alcanza la profundidad máxima momento en el que solo se seleccionan nodos terminales.
- *Ramped & half:* Se genera la mitad de la población con el método completo, es decir, profundidad máxima y la otra mitad con el método creciente. En la mitad de la población que se genera con el método de inicialización creciente constan de una profundidad máxima y mínima.
- *L & L (método propio):* Para este método se escoge una profundidad mínima y una máxima. Hasta la mínima se usa el método de inicialización completa y una vez se ha alcanzado la profundidad mínima, desde esa hasta la máxima, el método creciente. Decidimos implementar este método porque es un método híbrido y favorece una población variada en tamaños.

3.2.3. Operadores

Al igual que en la computación evolutiva general, los operadores que se usan en programación genética son los de selección, cruce y mutación. Particularmente, hemos utilizado los siguientes:

3.2.3.1. Operadores de selección

- *Selección por ruleta:* a cada individuo se le asigna una parte proporcional en base a su aptitud y a un número aleatorio calculado para cada individuo. La suma de todas las aptitudes debe ser 1.
- *Selección estocástica:* similar a la selección por ruleta, con la diferencia de que se genera un sólo número aleatorio para toda la población.

- *Selección por ranking:* se ordena a los individuos en orden decreciente según su Fitness. En base a esa ordenación, se le otorga un valor a cada uno que será lo que determine su elección o no.
- *Selección por torneo:* se hace competir a los individuos, según su Fitness y un factor aleatorio. En base al resultado obtenido, los mejores tendrán más posibilidades de ser elegidos.

3.2.3.2. Operadores de cruce

1. *Cruce simple*: se seleccionan dos nodos distintos de la raíz en dos árboles. Se cortan por esos puntos y se intercambian los subárboles como se indica en 3.6. endenumerate

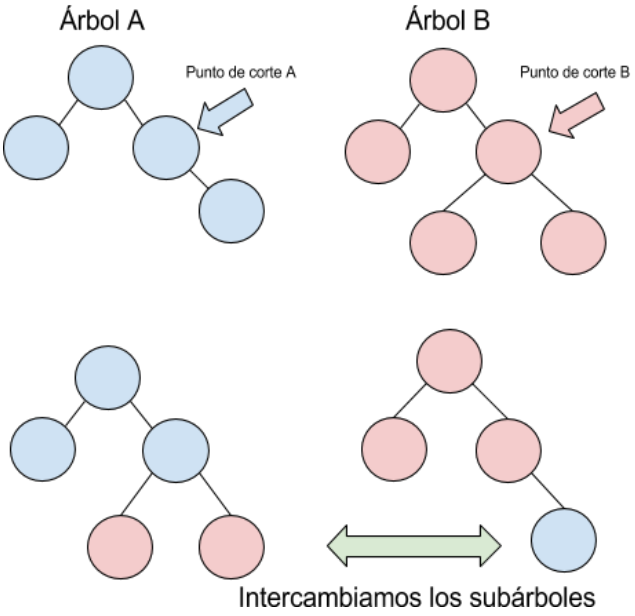

Figura 3.6: Cruce monopunto

3.2.3.3. Operadores de mutación

a) *Mutación de árbol*: se selecciona un nodo y se sustituye por un subárbol generado aleatoriamente. Ver 3.7.

b) *Mutación de función*: se selecciona un nodo que represente un operador de función y se sustituye por otro que corresponda a

otra función con la misma aridad, es decir, que tenga el mismo número de hijos. Ver 3.8.

c) *Mutación de terminal*: se selecciona un nodo que represente un terminal y se sustituye por otro. Ver 3.9.

d) *Mutación combinada*: se elige aleatoriamente qué tipo de mutación de las tres anteriores se va a utilizar y se aplica.

e) *Mutación por permutación*: consiste en intercambiar los hijos de un padre. Esta mutación decidimos no introducirla ya que, después de varias pruebas con las mencionadas anteriormente, constatamos que no iba a suponer ninguna mejora. Ver 3.10.

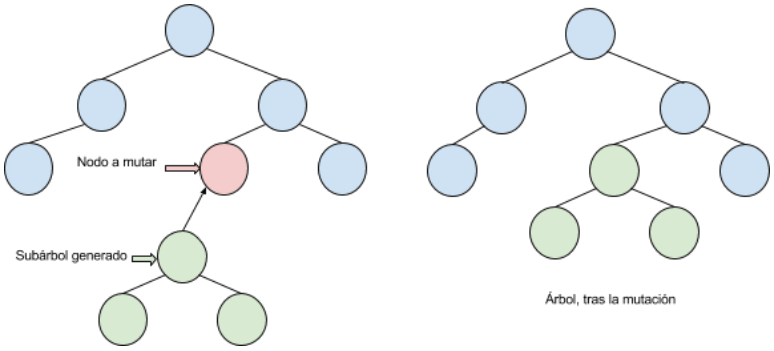

Figura 3.7: Mutación de árbol

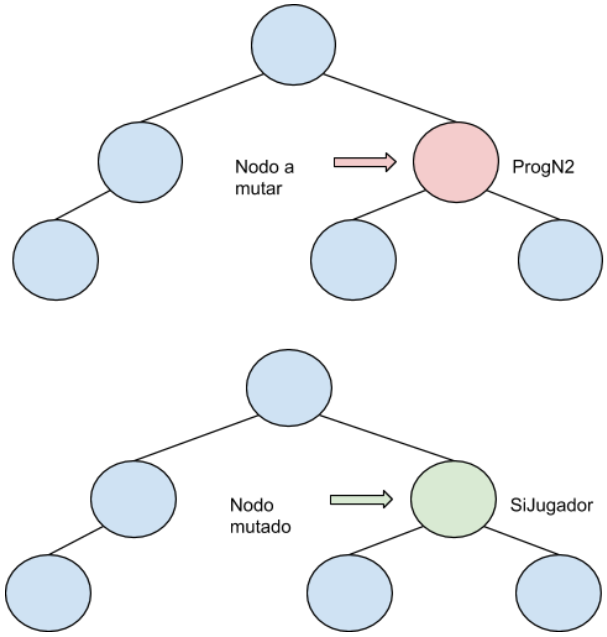

Figura 3.8: Mutación de funcion

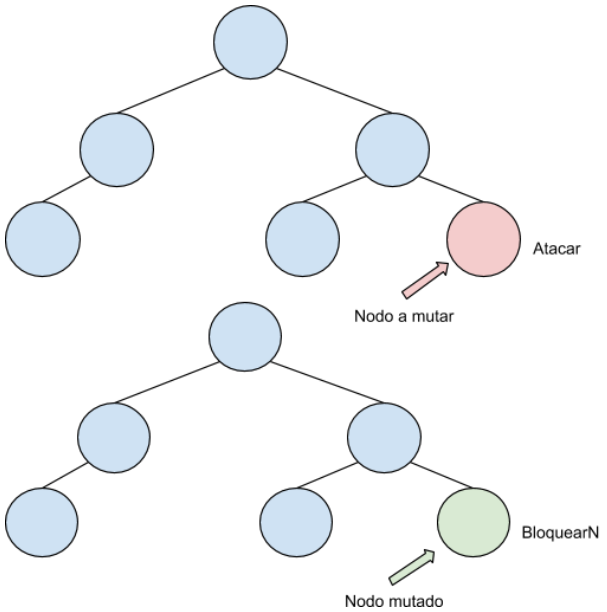

Figura 3.9: Mutación terminal

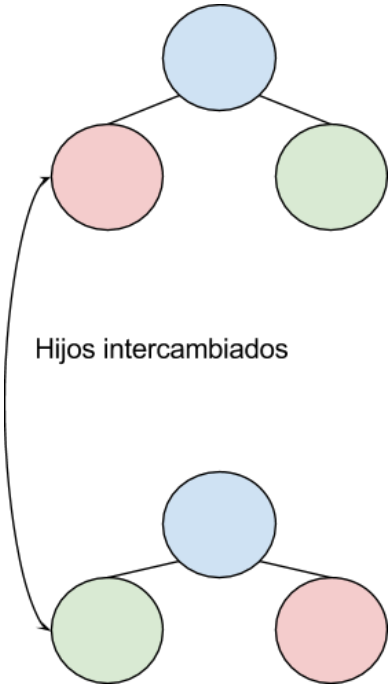

Figura 3.10: Mutación por permutación

3.2.3.4. Función de evaluación -Fitness-

La función de fitness calcula la aptitud de un individuo en base al resultado con que éste resuelve el problema. Se tienen que evaluar las operaciones que figuran en su árbol y compararlas con los valores óptimos. Se deberá calcular su aptitud dependiendo de si el óptimo es un valor concreto, o si el problema debe maximizar o minimizar.

3.2.4. Mejoras en el algoritmo evolutivo

3.2.4.1. Control del bloating

Cuando el genotipo de los individuos se basa en una estructura de datos que puede expandirse de forma infinita, como por ejemplo un árbol, se limita su crecimiento para evitar individuos excesivamente grandes. Cuando los individuos crecen en exceso, sin que ello suponga una mejora significativa del fitness, se utilizan muchas técnicas para controlar ese crecimiento:

a) Penalizar a los individuos muy grandes en la función de fitness.

b) Establecer límites al tamaño de los individuos.

c) Evaluar a varios individuos y, en caso de aptitud similar, escoger a los más pequeños.

d) Evitar cruces que produzcan hijos peores que los padres.

Algunos de los métodos de control de bloating son Penalización bien fundamentada o Tarpeian.

3.2.4.2. Intrones

Debido a la representación en forma de árbol de los individuos, en ocasiones pueden aparecer ramas con expresiones redundantes. Estas expresiones provocan que los individuos contengan información irrelevante que puede ser eliminada o sustituida por expresiones más simples. Un caso típico en programación genética son aquellos nodos de decisión -equivalente a una sentencia if then else- en la que ambas ramas conducen a ejecutar las mismas acciones. Este tipo de expresiones redundantes se denominan intrones.

La detección de intrones depende siempre del problema concreto que se pretende resolver, ya que es un problema asociado directamente a la semántica y el contexto de los individuos. Las mismas expresiones pueden ser consideradas intrones para un problema concreto y no serlo para otro.

3.2.5. Ejemplos

Un ejemplo para ilustrar lo que es la programación genética es el problema de "la hormiga artificial sobre el rastro de Santa Fe" (Araujo y Cervigón, 2009) (Koza, 1992). Hay que diseñar una hormiga artificial que sea capaz de encontrar toda la comida situada en un tablero de 32x32 casillas, empezando desde la casilla situada en la esquina superior izquierda.

Hacen falta operaciones que permitan avanzar, girar en ambas direcciones o comer. Por tanto, se pueden utilizar las siguientes funciones y los siguientes terminales:

- TERMINALES = Avanza, Derecha, Izquierda; donde Avanza hace que la hormiga avance una casilla en la dirección en que esté mirando en ese momento, y Derecha e Izquierda giran 90º en la dirección correspondiente.

- FUNCIONES = SIC(a, b), PROGN2(a, b), PROGN3(a, b, c); donde *SIC(a, b)* ejecuta la orden *a* si detecta comida delante y *b* en otro caso, y ambos *PROGN* evalúan sus argumentos en orden devolviendo el último de ellos.

Un posible individuo y el correspondiente árbol que lo representa sería:

$PROGN3(Derecha, PROGN2(Avanza, Avanza), PROGN2(Izquierda, Avanza))$

En la figura 3.11 puede verse el árbol de este individuo.

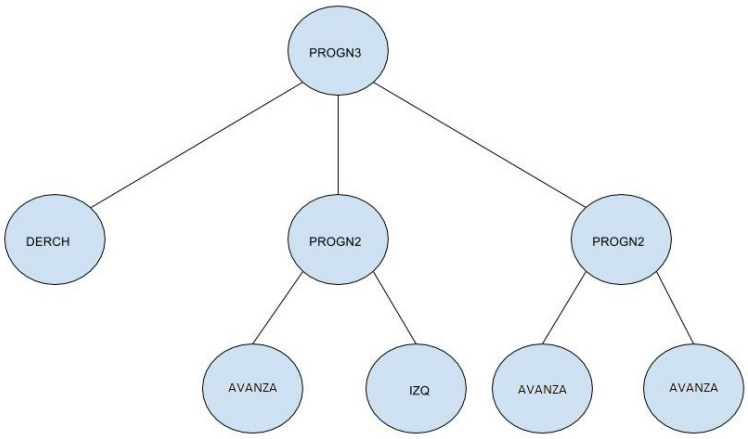

Figura 3.11: Ejemplo de árbol de un individuo

Para medir la aptitud de los individuos en este problema se puede utilizar la cantidad de comida consumida por la hormiga en un intervalo de tiempo determinado. Se considera que cada operación de avance o giro consume una unidad de tiempo. Para probar a los distintos individuos se propuso un modelo de rastro llamado "rastro de Santa Fe" y tiene un rastro irregular, porque hay algunos huecos de una o dos posiciones que también pueden estar en las esquinas.

En la figura 3.12 se muestra un ejemplo de ejecución (en negro) sobre dicho rastro en el cual se ve que la hormiga ha seguido una estrategia de tipo zig zag.

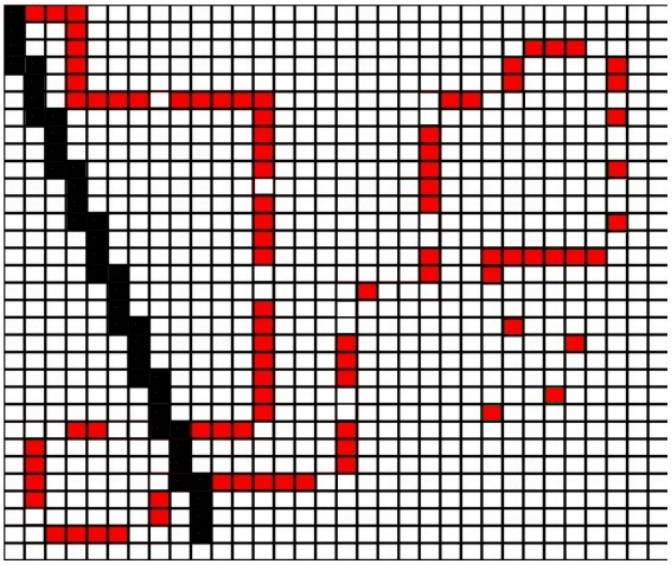

Figura 3.12: Ejemplo hormiga usando estrategia en zig zag

Ejemplo de ejecución de (PROGN3 Derecha (PROGN2 (Avanza Avanza)) (PROGN2 (Izquierda) (Avanza))

Capítulo 4

Técnicas para la generación de mazmorras

Infinito: Mayor que la cosa más grande que haya existido nunca, y más. Mucho mayor que eso, en realidad; verdadera y asombrosamente enorme, de un tamaño absolutamente pasmoso, algo para decir: "vaya, qué cosa tan inmensa".

Douglas Adams Novelista y humorista inglés.

4.1. Idea inicial

La idea inicial de este trabajo era investigar y desarrollar técnicas para automatizar diferentes partes o elementos de un videojuego mediante AGs, y, si fuese posible, utilizar una red neuronal como elemento que otorgue una nota o medida de calidad -fitness- a los individuos. La intención es que, mediante técnicas de programación evolutiva, se logre automatizar los procesos de desarrollo del mayor número de elementos posibles de un juego.

4.2. Generación de mazmorras mediante algoritmos genéticos

4.2.1. Introducción

Como se ha explicado anteriormente, se quieren generar mazmorras de forma automática utilizando AG. Con la descripción de mazmorra que hemos dado, la tarea principal es encontrar una implementación de cromosoma que describa correctamente toda la información relevante de una mazmorra, pero con la que además sea posible aplicar los operadores genéticos propios del AG.

Un individuo de nuestro AG es entonces una mazmorra completa, es decir, un conjunto de salas interconectadas.

Por lo tanto parece viable que la representación para nuestro individuo sea un grafo que represente los elementos interconectados. Desde el punto de vista del AG, los grafos presentan ciertas ventajas: son fáciles de implementar, se pueden cortar y combinar con facilidad en poco tiempo y las mutaciones pueden consistir en añadir o quitar tanto nodos como aristas aleatorias.

Otras ventajas que tiene la implementación mediante grafos son la gran variedad de implementaciones disponibles y la facilidad para realizar combinaciones y mutaciones entre grafos.

4.2.2. Decisiones de implementación

Ya decidido que la representación de las mazmorras será un grafo, existen múltiples formas de implementarlo. Teniendo en cuenta las operaciones a realizar, las distintas implementaciones tendrán diferentes costes, tanto temporales como en espacio.

En nuestro caso comparamos tres implementaciones típicas, como se puede ver en 4.1:

- Matriz de adyacencia: Una matriz bidimensional de dimensiones NxN, donde N es el número total de nodos, en la que cada posición Mij tendrá valor true o false según exista o no arista entre los nodos i y j respectivamente.
- Lista de aristas: Se mantiene un array de Aristas, indicando los nodos Origen y Destino (simple nomenclatura, dado que el grafo no será dirigido).

- Lista de adyacencia: Similar a la matriz de adyacencia, pero en lugar de ser una matriz bidimensional, de cada nodo se guarda un conjunto de nodos adyacentes.

En primer lugar es necesario mencionar que todas estas implementaciones deben ir acompañadas de una lista con la información que guardan los nodos, en nuestro caso particular la información propia de cada sala (dimensiones, enemigos, etc).

Para justificar la toma de decisiones, es necesario entender cómo es cada implementación internamente, y cuáles son sus costes.

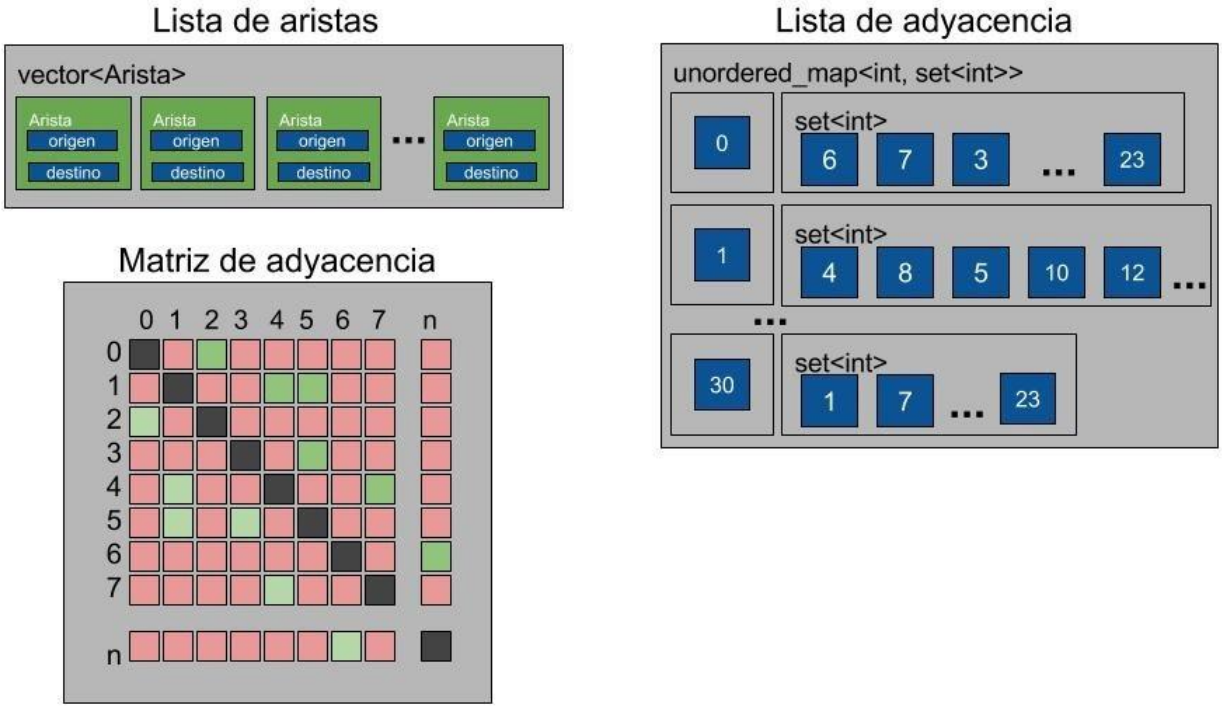

Figura 4.1: Diferentes implementaciones del Grafo

Los costes de las operaciones más utilizadas, en el caso peor, son:

OperacionesImplementación	L. Aristas	L. Adyacencia	M. Adyacencia
Hay arista en (i, j)	O(A)	O(log(G))	O(1)
Grado(i)	O(A)	O(1)	O(N)
Insertar nodo	NA	O(1)*	$O(N^2)$**
Insertar arista	O(1)*	O(log(G))	O(1)
Coste en espacio	O(A)	O(2A)***	$O(N^2)$

Tabla 4.1: Costes implementaciones Grafo

Leyenda: NA: No Aplicable; A = nº Aristas; N = nº Nodos; G = Grado del nodo 'i'
**El coste de inserción es constante (de promedio)
***Conlleva crear otra matriz mayor (puede optimizarse)
****Cada arista aparece dos veces, una en 'i' y otra en 'j'

Descartamos de forma casi inmediata la implementación de Lista de aristas por varias razones. En primer lugar, esa implementación conlleva que, o bien un nodo ejerce de origen y otro de destino para una arista en concreto, lo que dificulta la operación de saber si dos nodos son vecinos, ya que es necesario buscar en ambos extremos de la arista cada uno de los nodos, necesariamente recorriendo la lista (coste lineal en el nº de aristas), o bien se duplica la información, poniendo cada arista dos veces, pero intercambiando los nodos origen y destino. Otro de los inconvenientes de dicha implementación es que, dado un nodo en particular, no se puede saber de forma inmediata con cuántos nodos tiene arista, esto es, el grado del nodo.

Entre las implementaciones de Matriz de adyacencia y Lista de adyacencia debíamos decidir cual nos convenía más.

El principal problema de la Matriz de adyacencia es que implica crear una nueva matriz si se amplía el número de nodos del grafo; además, es lógico pensar que en operaciones que impliquen mezclar grafos será necesario copiar trozos de cada matriz, lo que tendría un coste promedio en tiempo de N^2. La única ventaja de la Matriz de adyacencia es que el coste de saber si dos nodos están conectados es constante, pero tiene muchas desventajas en operaciones más complejas.

Por su parte, la Lista de adyacencia tiene costes logarítmicos en el grado de cada nodo en particular en los peores casos, lo cual no es demasiado, ya que los grafos no deberían ser demasiado densos en general. Además, por su propia estructura, mantiene la adyacencia de cada nodo de forma separada, lo que facilita dividir un grafo en dos,

quitando de cada mitad los nodos vecinos que estuviesen en la otra.

Es importante mencionar que la operación Grado(i) tiene un peso especial, ya que, como se explicará más adelante, será una de las bases de la función de fitness del AG; esta función se utilizará con mucha frecuencia por lo que conviene que tenga un coste constante.

Por estos motivos decidimos usar la Lista de adyacencia como nuestra implementación de grafo. Para muestra, la figura 4.2 y la siguiente tabla:

Nodo considerado	Lista de adyacencia
1	2 y 3
2	1, 3 y 4
3	1, 2 y 4
4	2 y 3

Tabla 4.2: Lista de adyacencia nodos

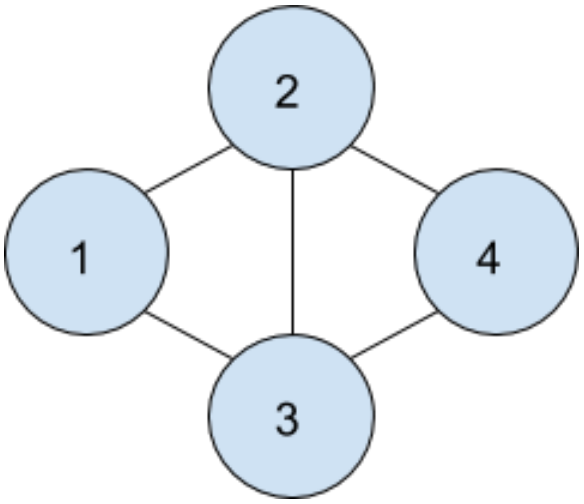

Figura 4.2: Adyacencia nodos

4.2.3. Evaluación de las mazmorras: Visión general

El grafo será entonces el esqueleto que determina cómo están unidas las salas. El segundo paso consiste en rellenar ese esqueleto con los elementos propios del juego como puertas que unen las salas, enemigos,

cofres y demás elementos decorativos. Esto se llevaría a cabo tras haber generado un grafo suficientemente bueno.

Sin embargo, nos presentaban un problema fundamental para el correcto funcionamiento del AG, la función de evaluación, esto es, ¿cuán bueno es un grafo?

Debido a la complejidad de la implementación, tuvimos que descartar la idea inicial de que la función de evaluación la realizara una red neuronal previamente entrenada con ejemplos a mano. Optamos entonces por crear una función de fitness que tuviese un funcionamiento similar, evaluando cada grafo en función de diversos aspectos clave, cada uno de ellos con un peso que podríamos variar hasta dar con una ponderación adecuada. Ahora bien, ¿cuáles son estos aspectos clave?

Ésta era la siguiente pregunta a responder. En primer lugar, nos dimos cuenta de que debido a la aleatoriedad de los grafos, éstos podrían tener varias CCs diferentes, e igual una de ellas era suficiente para nosotros como mazmorra completa. Así pues, la primera decisión que tomamos fue que la nota o "medida de calidad" de un grafo completo sería la mejor nota de cualquiera de sus CCs. Quedaba entonces por determinar los aspectos clave para evaluar las CCs.

El objetivo era generar una mazmorra que, sin ser excesivamente grande, resultase divertida. Para este punto, decidimos que la mazmorra tuviese ciertas salas clave, en concreto, una sala inicial, de la que parte el jugador, una sala final, a la que debe llegar, y una sala con una llave, que permitirá al jugador pasar al siguiente nivel desde la sala final. De esta forma, se proporciona un objetivo al jugador para explorar la mazmorra.

Durante dicha exploración, el jugador debería encontrarse obstáculos que superar (enemigos) y recompensas por haber explorado (cofres). Conseguimos esto sencillamente generando para cada sala un número aleatorio de estos dos elementos. Sin embargo, este enfoque presentaba unos problemas que tuvimos que resolver y se explicarán más adelante.

Tras un profundo análisis, consideramos que había 5 puntos importantes que eran interesantes para nuestros mapas o mazmorras.

- **La distancia entre las salas clave:** si las tres salas clave están demasiado juntas, el jugador no tendrá la necesidad de explorar la mazmorra.
- **El tamaño de la mazmorra:** si la mazmorra es muy pequeña resultaría aburrida; por el contrario, si es demasiado grande, podría resultar injugable.

- **La conexión entre salas:** si las salas están demasiado interconectadas, al jugador le cuesta mucho más crear un mapa mental de la mazmorra.
- **La dispersión de enemigos y cofres:** al ser situados de forma aleatoria, podría resultar que en una zona de la mazmorra se acumulasen muchos enemigos o cofres.
- **El tamaño de las salas:** se favorece que las salas tengan una forma rectangular (apaisada), de manera que representarlas en una pantalla sea más estético. Este parámetro es el menos relevante.

4.2.4. Evaluación de las mazmorras: Detalle

A continuación explicamos cómo valoramos cada uno de esos aspectos.

4.2.4.1. Distancia entre salas clave

El recorrido natural del jugador debería ser ir desde la sala inicial hasta la que contenga la llave, y luego encontrar la sala final para pasar al siguiente nivel. Por este motivo, decidimos **maximizar** la distancia (en salas o nodos) desde el origen hasta la sala de la llave y desde la sala de la llave hasta la sala final. Para el cálculo de la distancia, realizamos un recorrido en anchura (BFS) sobre el grafo. Para que una CC pueda puntuar en este aspecto, es necesario que contenga las tres salas clave. Favorecer mazmorras que solo tuviesen dos de ellas podría desembocar en individuos a los que les faltase una de las salas clave, lo cual no es permisible.

$$DistanciaSalasClave = distancia(SalaInicio, SalaLlave) +$$

$$+distancia(SalaLlave, SalaFin)$$

4.2.4.2. Tamaño de la mazmorra

El tamaño de la mazmorra no es más que el número de salas que tiene la CC. Determinar si este era un aspecto que debíamos maximizar o minimizar no fue sencillo. Si maximizamos, corríamos el riesgo de generar mazmorras demasiado grandes y si minimizamos, sucedería lo contrario. Decidimos que lo mejor sería **minimizar** este aspecto, y en combinación con la distancia entre salas clave, se produciría un "tira y afloja" en el que un aspecto favorecería mazmorras con muchas salas,

mientras que el otro potenciaría aquellas más pequeñas. Si estos dos parámetros se ajustaban correctamente, podríamos generar mazmorras con un número adecuado de salas (entre 15 y 30, según nuestro criterio).

4.2.4.3. Conexión entre salas

Para evitar mazmorras demasiado complejas debíamos **minimizar** la interconexión entre salas. A *priori* puede parecer que minimizar este aspecto conllevaría favorecer aquellas mazmorras que tuviesen pocas salas, pero al valorar la distancia entre salas clave como algo positivo lo que este factor favorece es la generación de mazmorras con varias salas interconectadas, pero de forma mínima 4.3.

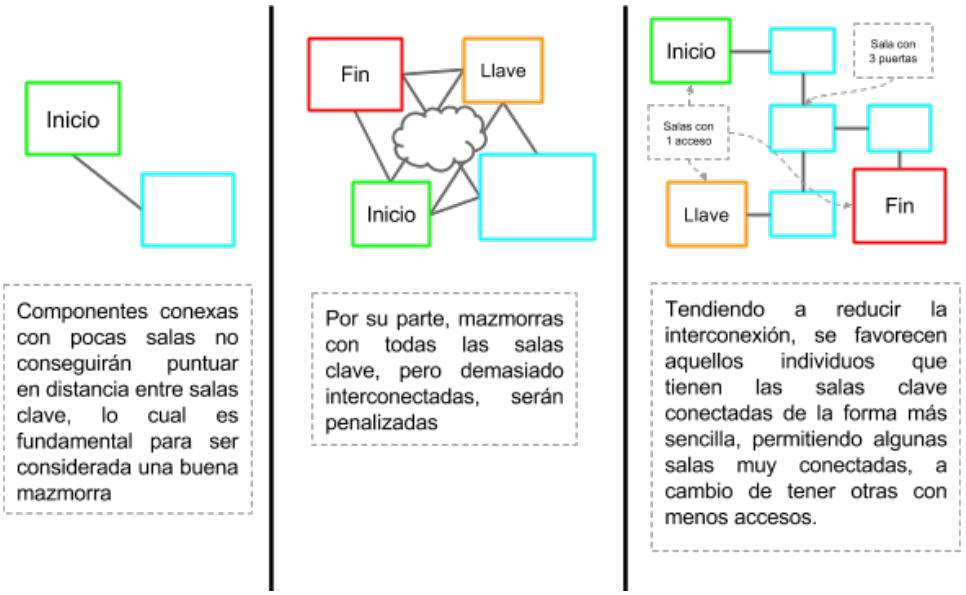

Figura 4.3: Conexiones entre salas

La forma de minimizar la interconexión es penalizar aquellas CCs en donde las salas tengan demasiadas entradas. En otras palabras, se pretende minimizar el grado de los nodos del grafo.

$$MediaGrado = \frac{\sum_{i=1}^{n} grado(i)}{n}$$

Donde n es el número de salas que tiene la CC y *grado(i)* es el número de aristas que tiene la sala i.

4.2.4.4. Dispersión de enemigos y cofres

Este parámetro sin duda debía ser maximizado ya que lo que se buscaba era que estos elementos del juego estuviesen muy repartidos. Para un elemento del juego, por ejemplo cofres, el parámetro de dispersión se establece como la distancia media (en salas) que separa a cada cofre del resto. Esta distancia se calcula para cada cofre de la CC y luego se realiza la media total. Más formalmente:

$$Dispersion(e) = \frac{\sum_{i=1}^{n_e} \frac{\sum_{k=1}^{n_e} distancia(i,k)}{n-1}}{n}$$

Donde e es un elemento del juego (enemigos o cofres) y n_e es la cantidad de dicho elemento en la CC. La función *distancia(i, k)* devuelve la distancia en salas entre el elemento i y el elemento k, si ambos están dentro de la misma sala la función vale 0.

4.2.4.5. Tamańo de las salas

Para favorecer las salas de forma apaisada, se debía **maximizar** el ratio entre ancho y alto de las salas. Para ello, se calcula la media de ancho y alto de todas las salas de la CC. Luego se calcula el *ratio*, simplemente dividiendo estos dos valores. Por último, se le resta 1. Si el *ratio* entre ancho y alto de una CC es menor que 1, implica que, de media, sus salas son más altas que anchas, luego este parámetro será negativo y penalizará en la función de fitness. De forma contraria, si el ratio es mayor que 1, el valor sumará. Si es igual a 1, es decir, salas normalmente cuadradas, el parámetro no penaliza, pero tampoco suma al fitness.

$$RatioSalas = \frac{mediaAncho}{mediaAlto} - 1$$

4.2.4.6. La función de fitness

La función que determinaría entonces cuán buena es una mazmorra sería la que sigue:

$$Fitness = Maximizado - Minimizado$$

$$Maximizado = DistanciaSalasClave + Dispersion(cofres) +$$

$+Dispersion(enemigos) + RatioSalas$

$Minimizado = NumeroSalas * 0{,}4 + MediaGrado$

Multiplicamos por 0.4 el número de salas de la CC para que evitar que se penalicen de más las mazmorras grandes, ya que tras algunas pruebas, restar el número directamente provocaba que las mazmorras resultantes fueran demasiado pequeñas.

4.2.5. Operadores del Algoritmo Genético

Ya hemos determinado cuál es la estructura de los individuos dentro del AG, así como la forma de evaluar cada uno de ellos. Falta entonces explicar otro de los aspectos fundamentales de la programación evolutiva: los operadores de selección, cruce y mutación.

Como ya se ha explicado anteriormente, variando entre diferentes métodos se modifica la forma en la que una población de individuos evoluciona. En concreto, nosotros decidimos implementar y probar las posibles combinaciones de los siguientes métodos.

4.2.5.1. Métodos de selección

Con distintos operadores de selección, se determina qué individuos pasarán a la fase de cruce, es decir, se eligen los individuos que podrían ser progenitores de nuevos individuos que conformarán la siguiente generación. Puede parecer lógico optar por un método de selección que siempre elige a los mejores individuos de cada generación, sin embargo, al hacer esto se puede estar reduciendo significativamente el espacio de soluciones posibles que el AE pueda alcanzar. En ocasiones puede suceder que la combinación entre un buen individuo y otro no tan bueno ocasione la creación de un nuevo individuo que supere al mejor individuo obtenido.

Distintos métodos de selección afectan de forma diferente a las poblaciones que se obtienen. Un método de selección que favorezca mucho a los mejores individuos provoca un aumento en la presión selectiva y a su vez se puede producir una evolución en avalancha o una convergencia prematura a causa de la falta de diversidad.

La presión selectiva se define como la aptitud máxima entre la aptitud media de una población.

Si la presión selectiva es muy alta, se pueden producir superindividuos que reducen el espacio de búsqueda del problema, si es muy baja, es

indicativo de una falta de diversidad en la población. Lo ideal, es que la presión selectiva sea baja durante las primeras generaciones, y vaya aumentando conforme los individuos evolucionan.

Dicho esto, los operadores probados en la generación de mapas han sido:

- Selección por Ruleta: los individuos son seleccionados de forma proporcional a su aptitud. En un segmento de longitud 1, se divide de forma que a los individuos mejor adaptados les corresponde mayor porcentaje.
- Selección Estocástica: Mediante este método un individuo puede ser seleccionado múltiples veces, así como no ser seleccionado ninguna.
- Selección por Torneo: Se seleccionan individuos en grupos de 3 y se van escogiendo los que mejor adaptación tengan, hasta completar la población.
- Selección por Ranking: Se ordena en orden decreciente a los individuos según su fitness o adaptación.

4.2.5.2. Métodos de cruce

En el caso particular de los grafos, es difícil aplicar operadores de cruce complejos que realmente aporten algo nuevo en la generación de nuevos individuos. Por el momento, hemos probado con:

- Cruce monopunto: se elige un número aleatorio entre 1 y N-2, siendo N el número total de nodos del grafo. Esto genera dos subgrafos diferentes que serán combinados con los subgrafos del otro progenitor.

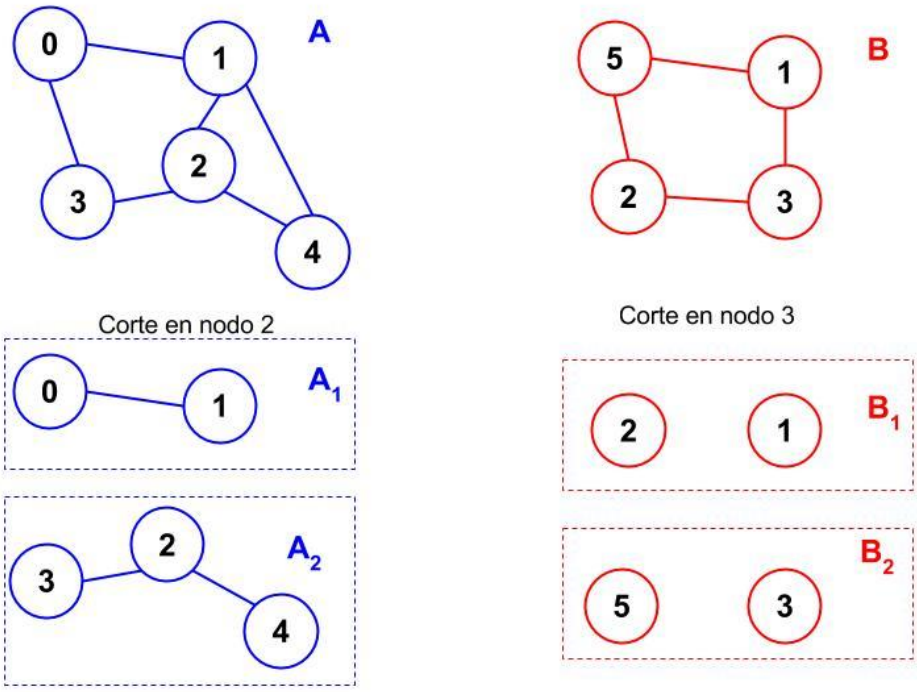

Figura 4.4: Cortes subgrafos

Como se ve en la figura 4.4, los puntos de corte de los progenitores pueden ser distintos. Al cortar, se generan dos subgrafos, que serán idénticos al grafo original, pero del cual se ha eliminado una parte de los nodos, y las aristas que conectan con dichos nodos.

Estos subgrafos se generan siguiendo el orden de los índices. Los mayores del índice de corte serán un subgrafo y los menores, otro.

Una vez obtenidos estos subgrafos, se combinan los subgrafos obtenidos de A, con los obtenidos de B, consiguiendo de esta forma nuevos individuos.

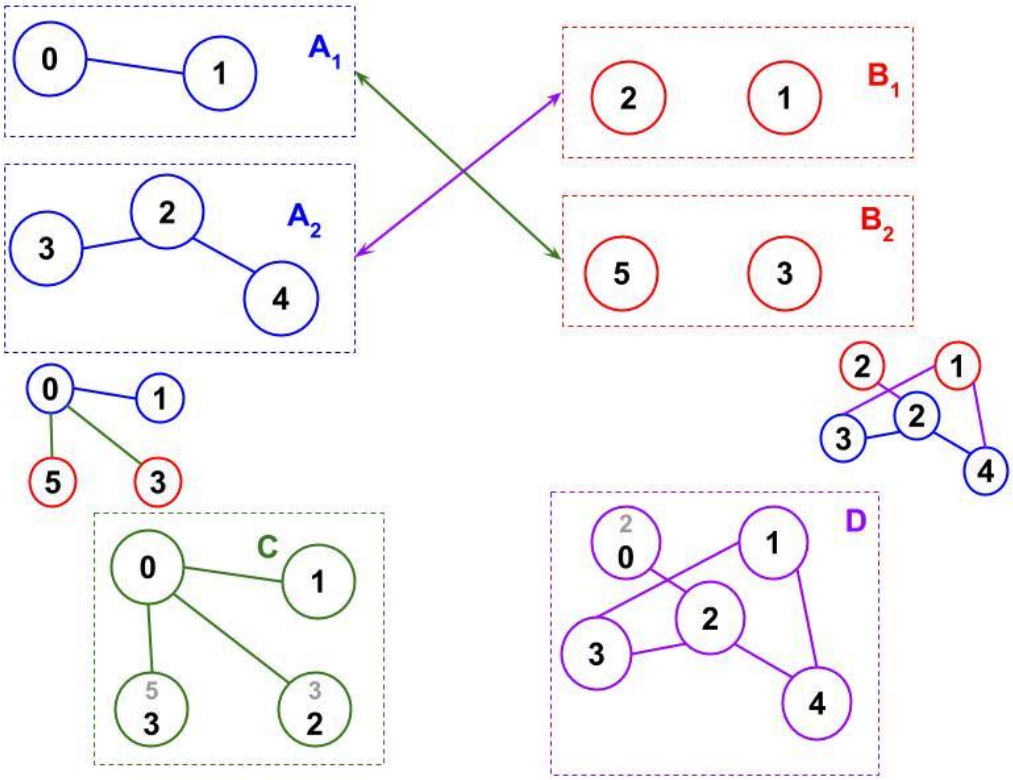

Figura 4.5: Unión subgrafos

Sin embargo, no es suficiente con unir los subgrafos de A y B; es necesario que entre estos subgrafos se hagan nuevas conexiones, sino se perderían aristas en cada cruce, consiguiendo eventualmente grafos totalmente inconexos.

Para evitarlo, una vez unidos los subgrafos en uno solo, se elige un conjunto pequeño de las posibles aristas entre los nodos del subgrafo A y los de B y se añaden al nuevo grafo. Una vez se han añadido estas aristas aleatorias, se reescriben los índices para que el rango sea de nuevo de 0 a N-1 4.5.

- Cruce multipunto: el cruce multipunto se basa en varios cortes monopunto. Se eligen varios puntos de corte, y se combinan los diferentes subgrafos de A y B para conseguir los nuevos individuos.

4.2.5.3. Métodos de mutación

De manera eventual e intentando imitar el fenómeno de la mutación aleatoria que sucede de forma natural, algunos individuos pueden modificar su estructura interna sin necesidad de combinarlo con otros. Éste es el operador de mutación.

Hasta el momento, tenemos cuatro implementaciones de mutación para los grafos:

- Mutación de arista: se añade o se borra (50 % - 50 %) una arista aleatoria 4.6.

Figura 4.6: Mutación de arista

- Mutación de nodo 4.7: se añade o se borra (50 % - 50 %) un nodo aleatorio.

Figura 4.7: Mutación de nodo

- Mutación de sala: Se mueve de sitio una de las salas clave (inicio, llave o fin) de un nodo a otro 4.8.

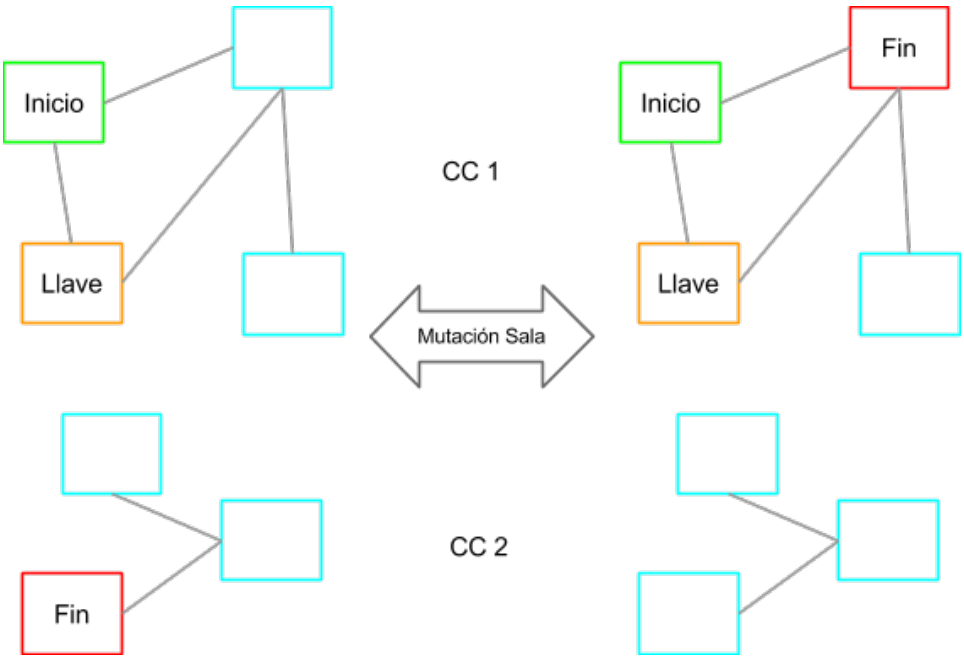

Figura 4.8: Mutación de sala

- Mutación combinada: una mutación que combina las 3 anteriores como una sola.

4.2.6. La implementación en detalle

Una vez vistos los conceptos teóricos del cromosoma, es necesario explicar cómo será internamente un individuo del AG en lenguaje C++.

Para empezar, hemos decidido abstraer el concepto de Grafo del concepto Cromosoma. En lugar de que el Cromosoma en sí contenga los elementos propios de un grafo, hemos optado por encapsular el Grafo dentro del Cromosoma.

Aun así, el Cromosoma mantiene las operaciones propias de un individuo: función de evaluación, control del bloating, guarda parámetros de adaptación, acumulada, etc.

La función de evaluación se basa en realidad en pedir toda la información necesaria al Grafo que compone su genotipo, y realizar los cálculos en base a ella.

Por su parte, el Grafo está construido según lo explicado anteriormente, con algunos pequeños detalles. Se mantiene una tabla hash donde se asocia un identificador de nodo unsigned int con su información asociada (tamaño de sala, nº de enemigos) Por otro lado, otra tabla hash

asocia un identificador de nodo i con un conjunto que indica qué nodos son adyacentes a i.

Los atributos de un Grafo son entonces:

std::unordered_map<unsigned int, std::setint>>_ady;

std::unordered_map<unsigned int N >_nodos;

Los Grafos son construidos de forma aleatoria en base a 3 parámetros clave:

- Número mínimo de nodos
- Número máximo de nodos
- Densidad de aristas del grafo

Grafo <N >(unsigned int minNodos, unsigned int maxNodos, double densidad);

Cabe destacar que hemos implementado la clase Grafo de forma genérica, pudiendo abstraer la implementación del Grafo de la de Nodo, dado que el Grafo no necesita conocer ninguna información asociada a los nodos más allá de la adyacencia. Los Grafos son una estructura de datos muy común en el desarrollo de videojuegos, y en nuestro caso representan una mazmorra, pero pueden ser utilizados como representación de otros elementos de juego, pudiendo ser combinados y mutados de forma igual o similar.

Algunas operaciones son necesarias para cualquier grafo, como la posibilidad de añadir nuevos nodos o crear una arista entre dos nodos.

void anadeNodo(N nodo, int id = -1);

bool anadeArista(unsigned int v, unsigned int w);

El método *anadeNodo* puede opcionalmente recibir el identificador del nodo a añadir (útil si se quieren cruzar dos Grafos con identificadores repetidos), aunque el identificador por defecto será el número de nodos del Grafo en el momento de la llamada.

Más importantes son los métodos específicos de nuestra implementación y de los que se ha hablado anteriormente. Entre ellos se encuentra el método:

unsigned int getGradoNodo(unsigned int v); Este método recibe un identificador de nodo y, si dicho nodo se encuentra en el Grafo, devuelve con cuántos nodos está conectado (grado del nodo). Dentro de este método, se busca el nodo en la tabla de adyacencia y se devuelve el tamaño del conjunto asociado, ambas operaciones de coste constante (en promedio).

```
std::vector <Grafo <N > > divideGrafo(unsigned int v);
static Grafo<N > unirGrafo(Grafo <N > a, Grafo <N > b);
std::vector<Grafo <N > > divideEnGrafos(unsigned int n);
static Grafo<N > unirGrafos(std::vector<Grafo <N > > subs);
```

El método *divideGrafo* recibe un identificador de nodo, que será el punto por el cual se dividirá el Grafo. De está forma, todos los nodos anteriores a él (inclusive) formarán el primer subgrafo, y los posteriores el segundo. Las aristas que hubiese entre ambos subgrafos son eliminadas. Por su parte, el método *unirGrafo* realiza la operación contraria, crea un Grafo en base a dos subgrafos, y añade aristas arbitrariamente entre ambos. Los métodos *divideEnGrafos* y *unirGrafos* no son más que múltiples llamadas a los dos métodos anteriores respectivamente. Por último, implementamos un método que permitiese conocer las CCs de un grafo, necesarias en la función de fitness, pues cada una de ellas se evalúa de forma independiente pese a formar parte del mismo Grafo.

```
std::vector<ComponenteConexa <N > > getComponentesConexas()
const;
```

En un primer momento, decidimos que la clase *ComponenteConexa* $<N>$ (la cual hereda de Grafo) nos podía permitir extender las operaciones que tenían los Grafos, como por ejemplo saber cuántos ciclos presentaba, sabiendo que en ellas no existen nodos inconexos unos de otros. Sin embargo, como se explicó anteriormente, está es una operación compleja y muy costosa en tiempo, así que a efectos prácticos, las CCs son iguales que los Grafos, pero con la garantía de que todos los nodos están conectados. Dentro del repositorio del proyecto se pueden encontrar más métodos, pero estos son los más relevantes.

4.2.7. Individuos obtenidos

A continuación se muestran ejemplos de mazmorras obtenidas con estos parámetros:

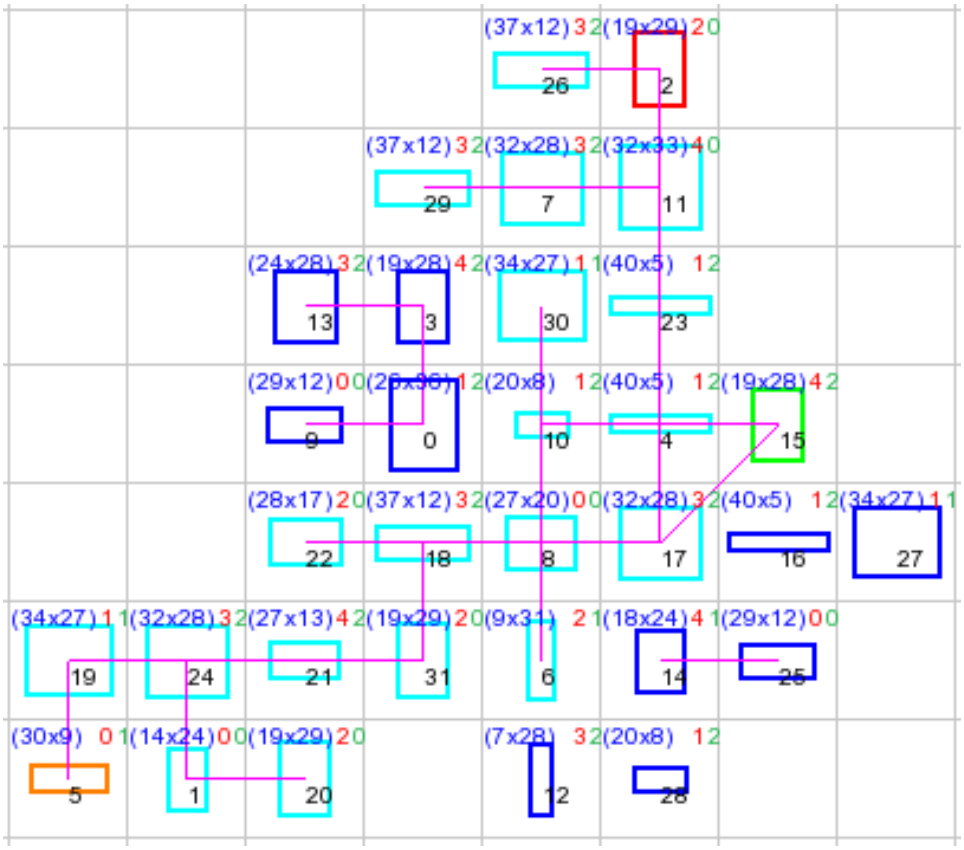

Figura 4.9: Ejemplo de mazmorra 1

- Las salas en color turquesa son las que forman parte de la CC seleccionada como mejor.
- Las salas coloreadas de un azul más oscuro son aquellas que no pertenecen a la CC seleccionada.
- La sala verde es la Inicial, la naranja la de la Llave y la roja la Final.
- Las aristas que unen los nodos (salas) están representadas mediante una línea morada.
- Los números en color negro representan la enumeración de la sala.
- Los números en rojo representan los enemigos dentro de esa sala.
- Los números verdes identifican los cofres.
- Los números azules identifican las dimensiones de la sala.

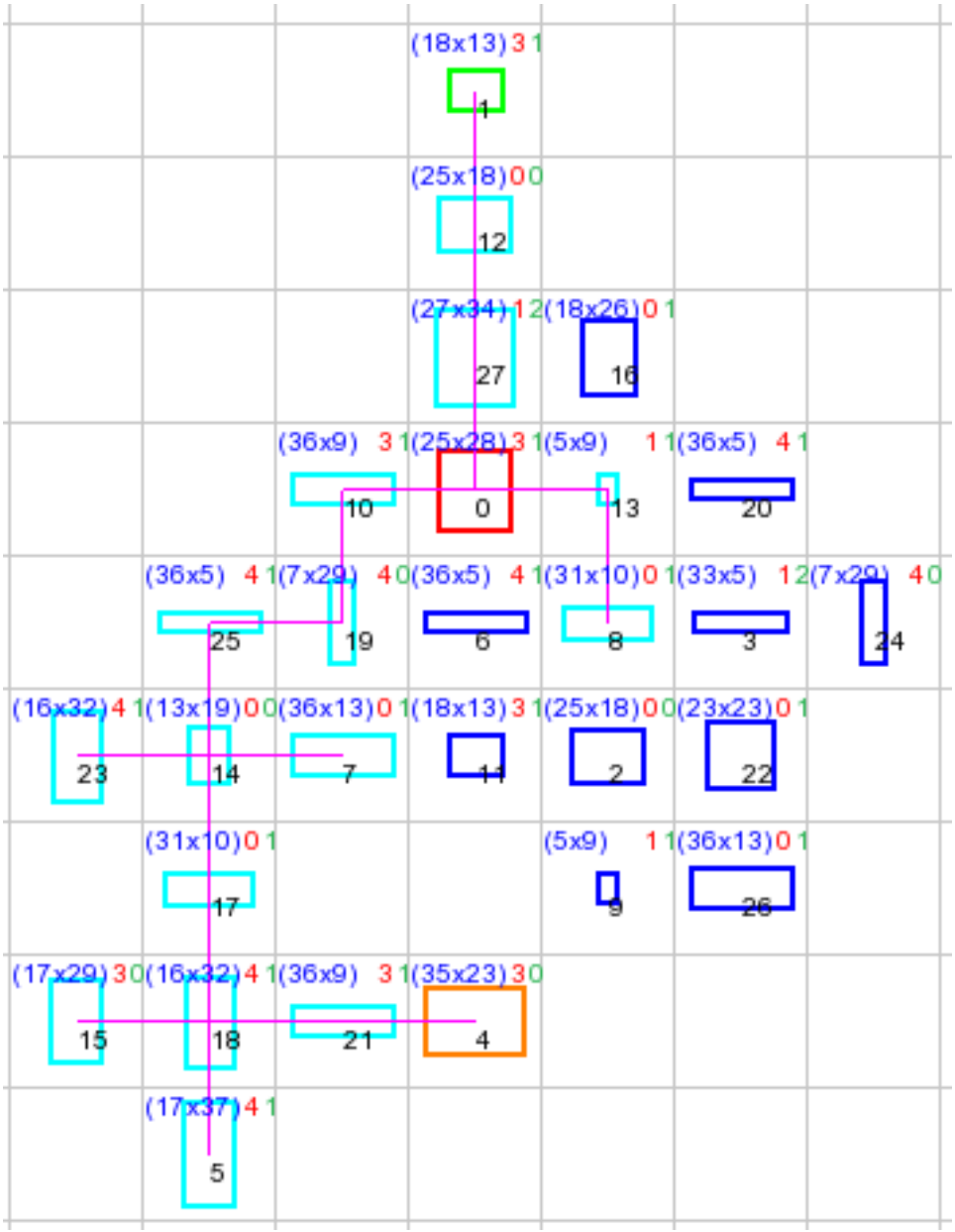

Figura 4.10: Ejemplo de mazmorra 2

Como se observa, las mazmorras son bastante buenas, ya que tienen el número de salas adecuado (entre 15 y 30) y no están demasiado interconectadas. Por tanto, en vista a los resultados empíricos llevados a cabo, podemos suponer que la función de fitness es lo suficientemente adecuada. Estos individuos han sido obtenidos tras varias horas de pruebas de parámetros del AG. El tiempo de generación de individuos

con un fitness aceptable (como los mostrados), está entre 1 y 3 minutos, un tiempo que consideramos aceptable como tiempo de carga de contenido en la ejecución del juego.

4.3. Interfaz de usuario

Al elegir como lenguaje de desarrollo C++, tuvimos que superar el problema de la representación gráfica. Este problema nos surgió en el desarrollo de las mazmorras. Decidimos utilizar la misma librería gráfica con la que posteriormente desarrollaríamos el juego: SFML. Para poder extraer conclusiones de la ejecución de los distintos AGs era necesario crear algo similar a un visor de gráficas de función. Para visualizar las gráficas de evolución y analizar la convergencia, se ha implementado un visor de funciones 4.11 -similar a la librería de java jMathPlot- con algunas de las operaciones que necesitábamos. A saber:

- Poder representar AGs con diversas generaciones. (Eje X de tamaño variable)
- Poder representar más de una línea de valores en el Eje Y (poder ver el fitness del mejor individuo global a la vez que la media o el mejor por cada generación).
- Poder asignar un nombre a cada línea de valores (Leyenda).

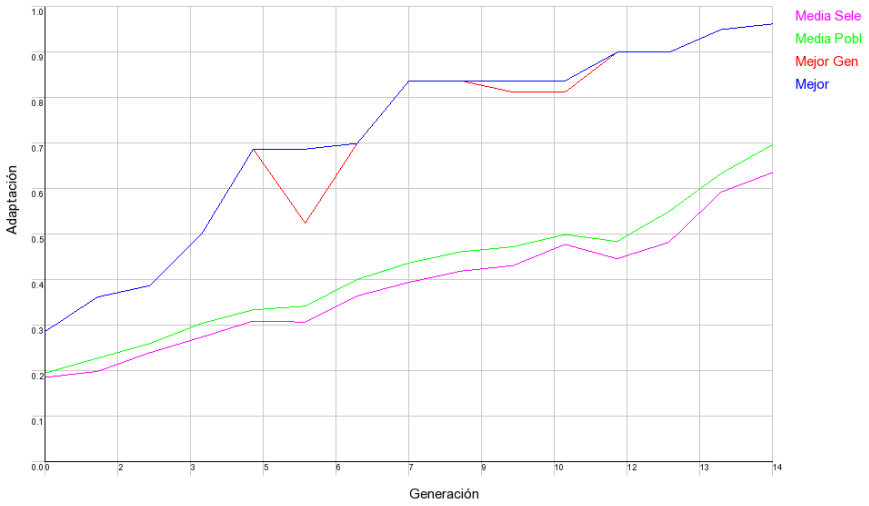

Figura 4.11: Visor de funciones

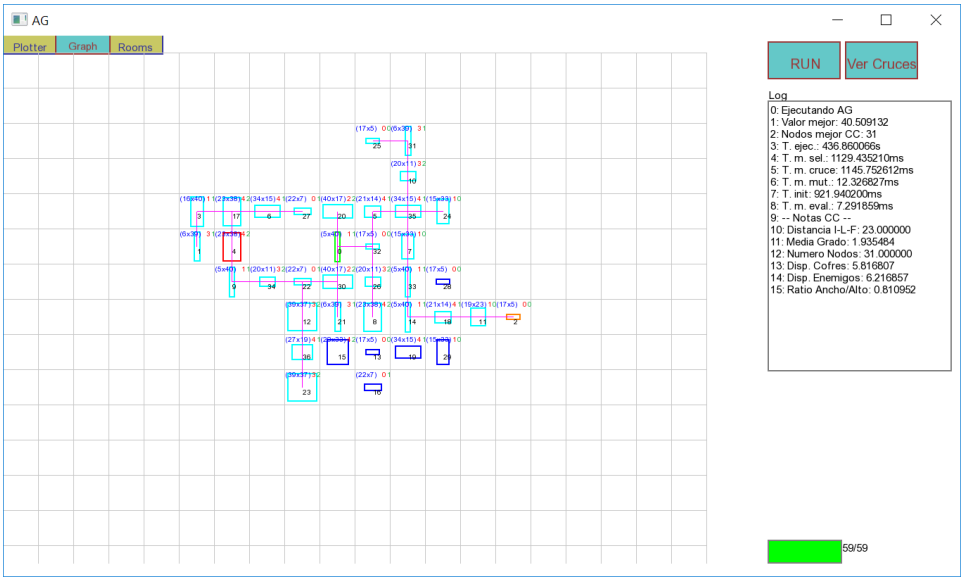

Figura 4.12: Visor de mazmorras

Este visor de funciones 4.11 nos ayudó mucho en lo respectivo a los AGs, sin embargo, aún resultaba complicado ver qué tipo de mazmorras se generaban. En las gráficas se puede apreciar como va evolucionando el fitness de la mazmorra con el paso de las iteraciones. Este valor se obtiene de la función de fitness mencionada anteriormente.

Por tanto, decidimos implementar un visor de grafos 4.12 que incluyese de forma clara la información de las salas. Por último, queríamos poder ver cómo se verían las salas definitivamente en el juego, así que decidimos crear también un visor de salas 4.13.

Figura 4.13: Visor de salas

Con algunos sprites provisionales, podíamos hacernos a la idea de la forma de las salas y su dificultad. Tanto los enemigos como los cofres

son colocados de forma aleatoria, pues su posición no está indicada ni es relevante en está fase del desarrollo.

De esta forma, con una GUI sencilla podíamos ver toda la información de una ejecución completa del AG.

Capítulo 5

Técnicas para la generación de la inteligencia artificial

Las tres leyes de la robótica son:
1. Un robot no puede dañar a un ser humano ni, por inacción, permitir que un ser humano sufra daño.
2. Un robot debe obedecer las órdenes dadas por los seres humanos excepto cuando tales órdenes entren en conflicto con la Primera Ley.
3. Un robot debe proteger su propia existencia hasta donde esta protección no entre en conflicto con la Primera o Segunda Ley.

Isaac Asimov, Bioquímico y escritor estadounidense.

5.1. Idea inicial

Para la parte relativa a la inteligencia artificial se pensó inicialmente en utilizar gramáticas evolutivas. Después de un profundo estudio y un análisis de las ventajas y desventajas, decidimos decantarnos por la **programación genética**. Las razón principal es que al utilizar programación genética trabajamos directamente con una codificación de los individuos en forma de árbol. Esto es importante ya que facilita

mucho la adaptación al videojuego y separa lo máximo posible la implementación y uso del AE con la inclusión de la IA en el videojuego. Por tanto, nuestro algoritmo para la IA se basa en la evolución de **árboles** que representan las distintas decisiones que toman nuestros PNJs.

Una vez habíamos decidido la tecnología, planteamos cómo íbamos a generar estas IAs para el videojuego. Decidimos que para cada PNJ íbamos a tener dos árboles: uno representa el árbol correspondiente al estado *Patrulla*, que consiste en que los PNJs se mueven por la sala intentando encontrar al jugador y el otro representa el árbol correspondiente al estado *Ataque*, en el cual los PNJs, una vez han encontrado al jugador, intentan acabar con él. De esta forma el enfoque utiliza programación genética con árboles diferentes dentro de cada estado de una máquina de estados.

Es importante destacar que las IAs, a diferencia de las mazmorras, no se crean en tiempo de ejecución. Estas IAs se generan previamente y se añaden al videojuego. En nuestro caso concreto, se genera un número de IAs y en la ejecución del juego se asignan de forma aleatoria a cada PNJ. Esto nos facilitó el proceso ya que no debíamos preocuparnos por la eficiencia en tiempo. Lo que nos preocupaba únicamente era que las IAs fueran lo más realistas y divertidas posibles dentro de las limitaciones que teníamos.

5.2. Decisiones de implementación

Una vez elegida la técnica y representación para soportar la IA, lo primero que tuvimos que decidir eran las operaciones o acciones a realizar por nuestros PNJs.

Las IAs debían ser capaces de realizar acciones básicas, como moverse, girar o atacar, en función de decisiones que dependerían de su estado dentro del mapa.

Con estas directrices, las acciones serían nodos hoja en el árbol, mientras que las decisiones serían nodos intermedios, que, en base a la condición que representen, determinarían cuál o cuáles de sus hijos deben ser ejecutados.

La gramática del árbol se divide entonces entre una serie de operaciones terminales y otras operaciones de función. Las reglas son sencillas. Como forman un árbol, las operaciones de función hacen las veces de nodos no hoja y las operaciones de terminales hacen de nodos hoja. Las operaciones de función pueden tener dos o más hijos. Por el contrario, las operaciones terminales no pueden tener ningún hijo, ya que

son hoja. A continuación enumeramos todas las operaciones originales distinguiendo su tipo y explicando lo que implican en el juego.

Las operaciones representadas con elementos terminales (nodos hoja) son:

- Avanzar: el PNJ avanza una casilla en la dirección en la que mira.
- Girar Izquierda: el PNJ gira 90º a la izquierda.
- Girar Derecha: el PNJ gira 90º a la derecha.
- Cambiar Estado: el PNJ cambia del árbol de *Patrulla* al de *Ataque*. Esta operación solo puede estar en el árbol de *Patrulla*.
- Bloquear N: el PNJ bloquea ataques en la dirección que mira durante N ticks del juego. Esta operación solo puede estar en el árbol de *Ataque*.
- Atacar: el PNJ ataca en la dirección que mira. Esta operación solo puede estar en el árbol de *Ataque*.
- Retroceder: el PNJ retrocede en la dirección contraria a la que mira manteniendo su orientación fija. Esta operación solo puede estar en el árbol de *Ataque*.

Las operaciones correspondientes a funciones son:

- ProgN2: encadena dos operaciones.
- ProgN3: encadena tres operaciones.
- Si jugador: si hay un jugador está en la casilla adyacente al PNJ y en la dirección que mira, ejecuta la primera acción y en caso contrario ejecuta la segunda. Este terminal solo puede estar en el árbol de *Patrulla*.
- Si bloqueado: si delante hay una casilla bloqueada (un borde, un muro,) realiza una acción, si no otra.
- Si jugador en rango: si el jugador está en rango de ataque, realiza una acción, si no, otra.
- Si jugador detectado: si el jugador está en rango de detección, realiza una acción, si no, otra.

Estas funciones toman la forma de un *if-then-else*, de tal forma que si se cumple la condición realizamos la operación del nodo derecho y si no, la del izquierdo.

Después de muchas pruebas, como se detalla más adelante en el punto **5.3.5**, decidimos que necesitábamos cambiar algunos de los nodos del árbol de ataque. Los nuevos nodos función del árbol de ataque quedaron en esto:

- ProgN2: encadena dos operaciones.
- ProgN3: encadena tres operaciones.
- Si bloqueado: si delante hay una casilla bloqueada (un borde, un muro) realiza una acción, si no otra.
- Si jugador en rango: si el jugador está en rango de ataque, realiza una acción, si no, otra.
- Vida IA: en función de la cantidad de vida del PNJ, realiza una acción u otra.
- Vida Jugador: en función de la cantidad de vida del jugador, realiza una acción u otra.

Los nuevos nodos terminales quedaron así:

- Bloquear N: el PNJ bloquea ataques en la dirección que mira durante N ticks del juego.
- Atacar: el PNJ ataca en la dirección que mira.
- Acercar: el PNJ se acerca al jugador utilizando el algoritmo A*. Esto lo permitimos ya que decidimos que una vez el árbol de patrulla detecta al jugador, se conoce siempre la posición de este.
- Alejar: el PNJ retrocede en la dirección contraria a la que mira manteniendo su orientación fija
- Curar: durante cuatro turnos el PNJ no realizará ninguna acción y, una vez se hayan terminado, gana un punto de vida hasta el máximo.

Un ejemplo de programa en formato de árbol sería el indicado en la figura 5.1.

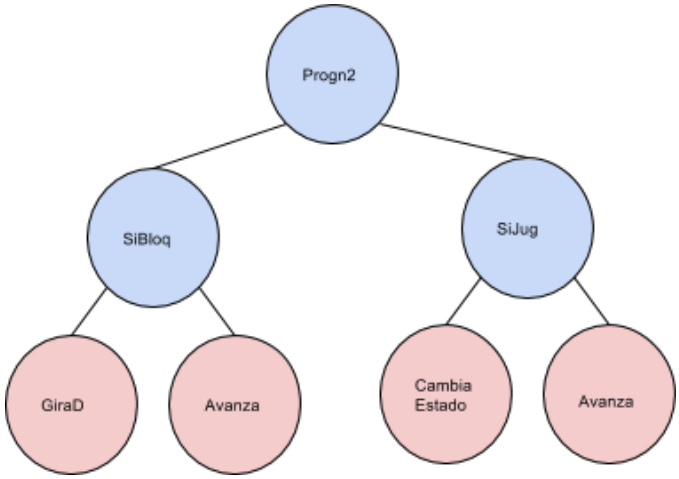

Figura 5.1: Ejemplo árbol

Y así sería el ejemplo de este mismo árbol como instrucciones:

$$PROGN2(SIBLOQ(GiraD, Avanza), SIJUGADOR(CambiaEstado, Avanza))$$

Estas operaciones las decidimos mantener en un Enumerado.

Posteriormente, elaboramos una clase Nodo que representa una operación dentro del árbol de decisión de la IA. Esta clase Nodo contiene a su vez un *hash map* para poder acceder de forma constante al número de nodos hijos que puede tener una operación en el árbol.

Para evitar que los árboles se volvieran demasiado largos, implementamos una función de control del bloating. En este caso concreto el control del bloating consiste en, una vez el árbol ha superado una profundidad máxima, podarlo y transformar todos los nodos situados en el nivel inferior en nodos terminales u hojas.

5.3. Función de evaluación

La función de evaluación de las IAs ha sido, sin duda, la parte más desafiante de este proyecto. Nos hemos frustrado en ocasiones y realizado multitud de pruebas, como se destaca en las siguientes páginas. Comenzó siendo una suma de valores ponderados y ha acabado siendo algo más parecido a lo que utilizamos en las mazmorras, una suma de elementos a maximizar, sustrayendo de estos elementos que consideramos malos y que por tanto deberían ser minimizados. El principal problema ha sido el tiempo de ejecución que rondaba unas doce horas.

Eso nos ha dificultado mucho la posibilidad de realizar más pruebas y nos frenó bastante durante las primeras aproximaciones hasta que encontramos la solución.

5.3.1. Introducción

La función de evaluación nos planteó una serie de preguntas a contestar que no eran sencillas. Teníamos varias dificultades añadidas de las cuales no hemos encontrado mucha información sobre el tema. En el tema de los videojuegos, como hemos podido leer en el estado del arte, sí que hay muchos estudios y técnicas desarrolladas para generar IAs que jueguen a videojuegos, desde juegos simples como el *Space Invaders* hasta más complejos como el *Street Fighter* (Capcom, 1987). La ventaja que tienen esos juegos sobre el nuestro es que son IAs que se van a enfrentar a otras IAs y los mapas son siempre conocidos o irrelevantes. Sin embargo, nuestra IA teóricamente debe enfrentarse a un humano que no tiene por qué seguir una estrategia siquiera similar. Además los mapas son aleatorios puesto que se generan en cada ejecución del juego.

Nuestro objetivo es intentar conseguir unas IAs buenas para un videojuego de forma automática. Esto es importante ya que las IAs no deben ganar siempre. Deben parecer inteligentes - podría ser más óptimo avanzar pegando espadazos a diestro y siniestro, pero no parecería muy natural - y tienen que ser divertidas, ya que, si te ganan siempre, el juego perdería su componente principal que es divertir al jugador.

5.3.2. Primera aproximación

Para conseguir una IA versátil decidimos crear una serie de mapas de prueba (6) y lanzar varias pruebas sobre cada uno de los mapas. Para cada prueba variamos la posición tanto del jugador como del PNJ. Para simular un jugador humano decidimos que era mejor que realizara actos aleatorios y no crearle nosotros un patrón de conducta o un árbol de decisión con el objetivo de evitar que nuestro algoritmo nos genere PNJs que sólo sepan jugar contra un tipo concreto de estrategia. De esta forma intentamos introducir el mayor grado de aleatoriedad a nuestro jugador con el objetivo de obtener un árbol de decisión capaz de enfrentarse a diferentes situaciones. Es por tanto una estrategia reactiva con la que se decide el movimiento o acción posible a partir del estado de juego actual, pero no tienen en cuenta las decisiones tomadas previamente.

También pensamos que lo mejor sería dividir este AE en dos árboles simulando dos estados de una máquina de estados. Estos estados serían *Patrulla* y *Ataque*. En el primero, el algoritmo se centra en premiar el mayor número de superficie observada y el tiempo que tarda el PNJ en encontrar al jugador. En el árbol de Ataque, se parte de la premisa de que sabemos la posición del jugador -o la casilla inmediatamente anterior-. Esta función de evaluación premia el atacar el mayor número de veces al jugador y recibir el menor número de golpes, siendo este último menos importante; el enemigo siempre preferirá morir él y el jugador a que escapen ambos.

Para evaluar un individuo, calculamos la media de puntuaciones obtenida en cada uno de los mapas de prueba. Para cada mapa de prueba el individuo tiene un número de turnos para demostrar su nivel adaptación. En esta primera aproximación el individuo disponía de 100 turnos por mapa, siguiendo el diagrama de flujo indicado en la figura 5.2.

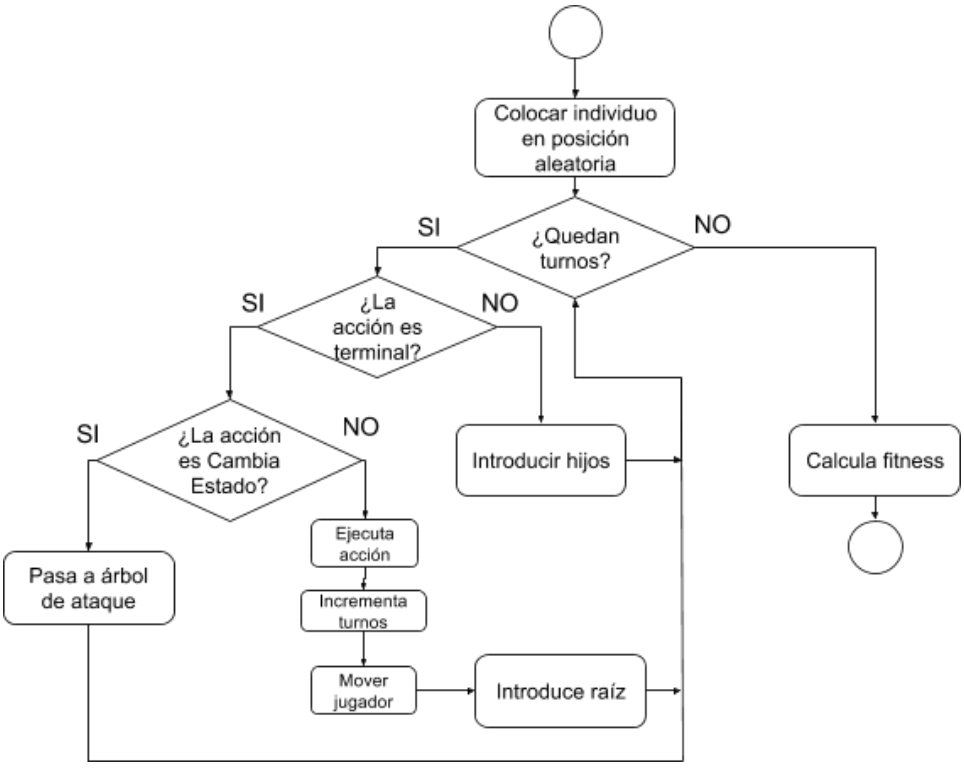

Figura 5.2: Diagrama de flujo

Una vez el individuo ha terminado de evaluarse en este mapa, se guardan unos valores que determinan lo bueno que es.

Estos valores eran:

- El número de casillas exploradas
- Turnos en el árbol de patrulla
- Golpes intentados
- Golpes recibidos
- Daño al jugador

De estos valores, nos interesaba maximizar el número de casillas exploradas, el número de golpes intentados y el daño al jugador. Por el contrario, queríamos minimizar el número de turnos en patrulla y los golpes recibidos por el PNJ.

Como la función de evaluación queríamos que estuviese entre cero y uno, generamos un array con los valores óptimos para cada elemento a considerar. Estos óptimos eran:

[DimensionDelMapa, 0, 20, 0, 3]

Y una vez teníamos un valor entre cero y uno para cada elemento, pasamos a ponderarlo para conseguir un único valor que nos indicaría el fitness. Los pesos eran los siguientes:

[0.3, 0.2, 0.05, 0.1, 0.35]

Una vez se ha realizado el proceso como se muestra en la figura 5.3 se calcula la media de todos los mapas, la cual nos daría la evaluación final del individuo.

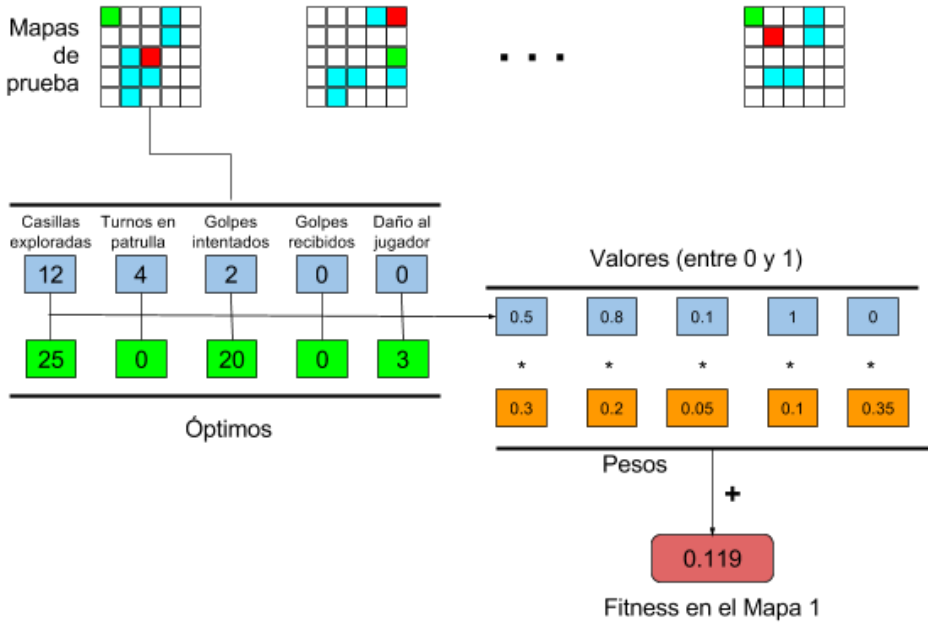

Figura 5.3: Evaluación mapa

5.3.3. Segunda aproximación

Después de esta nueva aproximación, decidimos modificar la función de evaluación para evitar el minimizar valores ya que siempre puntuaban al máximo si el individuo no hacía nada. Por ejemplo, si te quedas quieto, el jugador probablemente no irá a por ti y no recibirás heridas, o, si tu árbol de patrulla es simplemente cambiar al de ataque, los turnos eran mínimos, lo cual no nos parecía coherente. Finalmente decidimos que el árbol de patrulla iba a estar determinado por el tamaño del área explorada y el árbol de ataque por el número de golpes efectuados con éxito (aunque el jugador bloquee el golpe), el número de golpes bloqueados y el número de heridas infligidas al jugador.

La simulación se lanza sobre seis mapas. En el primero sólo se coloca al jugador en una posición y es vacío. En los demás, están rellenos como se muestran en las figuras 5.4, 5.5, 5.6, 5.7 y 5.8 y se lanzan cinco veces distintas, colocando cada vez al jugador en una de las casillas marcadas.

Como se puede apreciar en las imágenes, el objetivo era crear una serie de salas diferentes para que el individuo, al ser colocado de forma aleatoria en cada una de las evaluaciones, se enfrente a diferentes situaciones y ver cómo de bueno es ese árbol de decisión con salas dispares.

El flujo de la función de evaluación es el mismo que en la figura 5.2 y los elementos que valoramos pasaron a ser los siguientes:

- Número de casillas exploradas.
- Golpes intentados.
- Golpes bloqueados.
- Daño al jugador.

Eliminamos los óptimos por lo que la función de evaluación podía tomar cualquier valor y cuanto más grande fueran estos valores, más adaptado estaba ese individuo.

Sin embargo, estos valores sí que decidimos ponderarlos de la siguiente manera:

[0.5, 0.1, 0.05, 0.35]

Estos valores surgieron de la discusión de intentar valorar, sobre uno, la mitad en el árbol de ataque y la mitad en el árbol de patrulla. Estos valores los fuimos refinando con pruebas y aproximaciones.

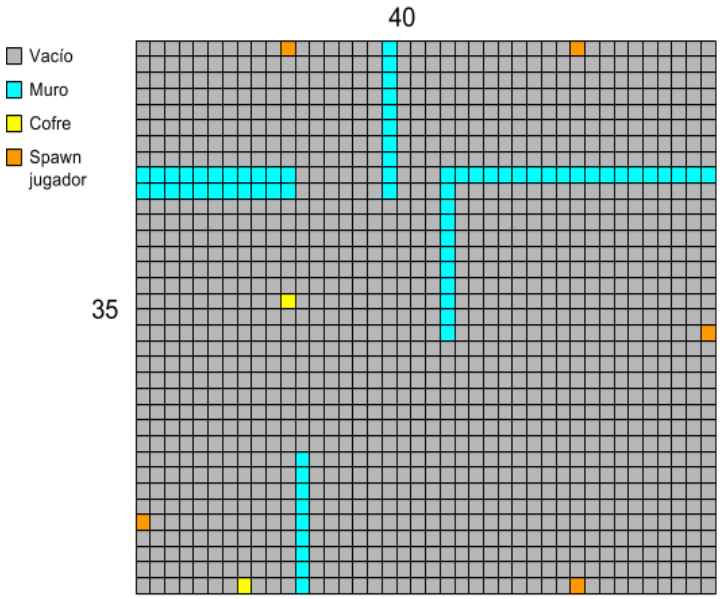

Figura 5.4: Mapa 1

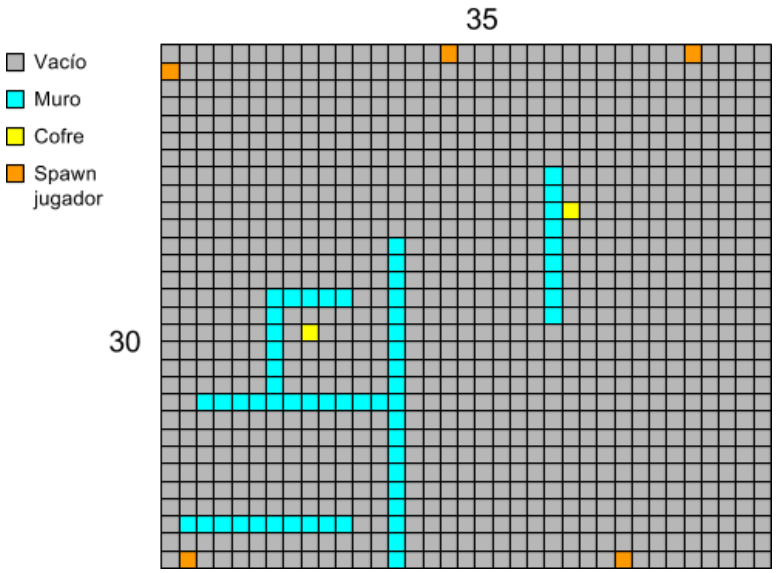

Figura 5.5: Mapa 2

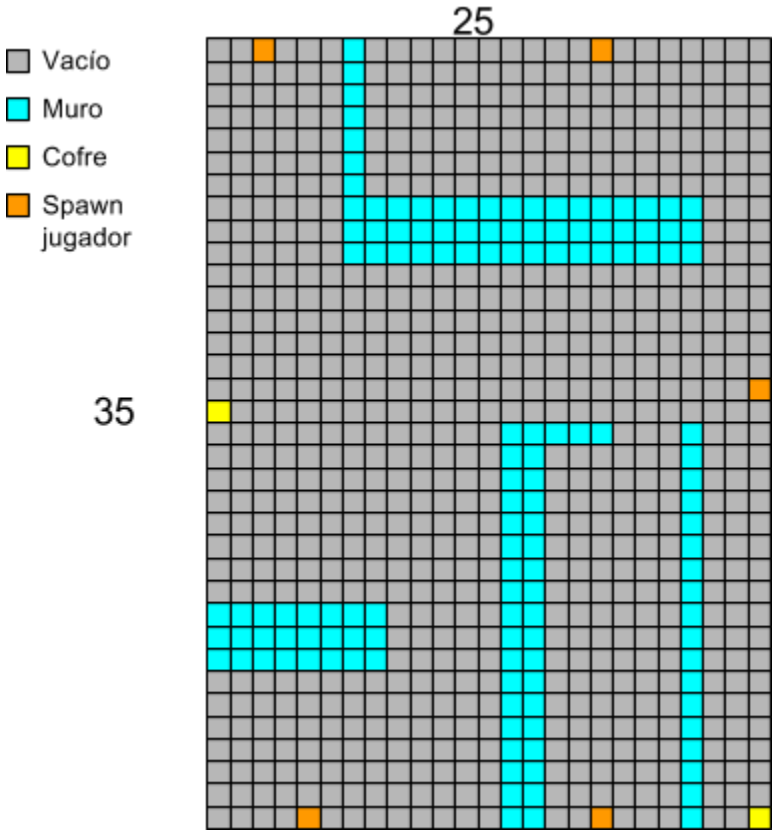

Figura 5.6: Mapa 3

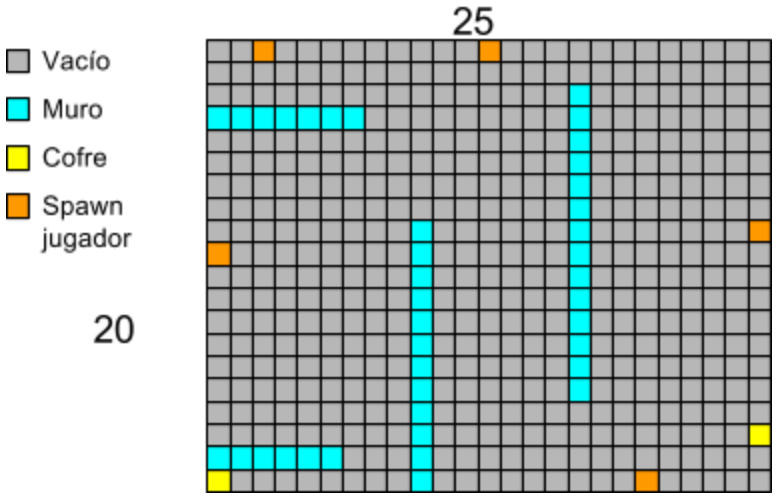

Figura 5.7: Mapa 4

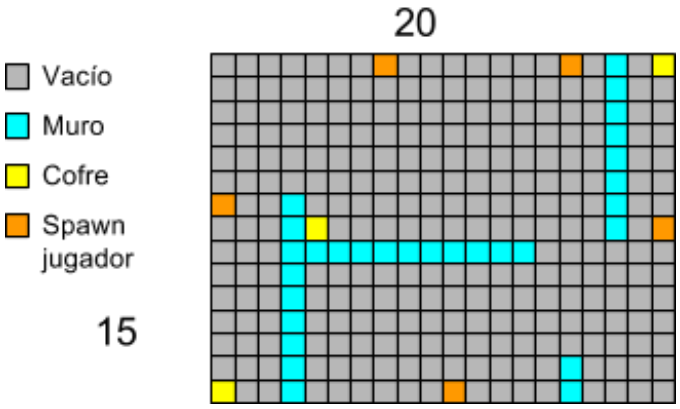

Figura 5.8: Mapa 5

5.3.4. Tercera aproximación

Tras varias ejecuciones, nos dimos cuenta de que estábamos infravalorando el árbol de patrulla de la mayoría de individuos, ya que sólo valorábamos las casillas exploradas, lo que descartaba a individuos que se movían por la sala de forma más o menos "inteligente".

Para está tercera aproximación, además de los elementos considerados anteriormente, incluimos en los cálculos cuántas casillas cubría el individuo.

Otro factor importante para valorar a los individuos era que debían tener la oportunidad de detectar al jugador, algo que hasta el momento podía resultar difícil si el jugador y el individuo aparecían en lugares muy separados. Por está razón, decidimos que la posición aleatoria del individuo debía estar cerca de la del jugador pero siendo aún así aleatoria.

Con tantos mapas de prueba, la función de evaluación demoraba mucho tiempo, así que decidimos optimizar en la medida de lo posible este paso. En primer lugar, en vez de re-evaluar cada individuo tras una modificación (cruce, mutación, bloating, introns) es más eficiente marcar a estos individuos como modificados y evaluarlos a todos tras haber pasado todas las posibles modificaciones.

En lo respectivo a la función de fitness, consideramos una buena idea descartar a un individuo si no puntuaba en alguno de los mapas. Esto es, si tras ser probado en un mapa, el individuo no consigue puntuar en ningún aspecto, se detiene su evaluación y se le puntúa con 0.

5.3.5. Aproximación final

Entre la anterior aproximación y está implementación final tuvieron lugar varias decenas de combinaciones de los parámetros importantes de los individuos. Antes de hallar una función de fitness definitiva, pudimos comprobar que, dados los nodos originales utilizados en el árbol de ataque, era prácticamente imposible para las IAs matar al jugador. La mayoría de ejecuciones terminaban con individuos con árboles de patrulla prácticamente perfectos, pero que, o bien no pasaban a ataque, o pasaban pero no conseguían atacar al jugador.

Por esto, decidimos cambiar los nodos utilizados en el árbol de ataque como se mencionó en el punto 5.2.

La intención de estos nuevos nodos es proporcionar a las IAs algo más de información. En nuestro primer enfoque, existía un problema fundamental. Durante la patrulla, la IA debe encontrar al jugador y para ello no requiere -ni se le proporciona- ninguna información más allá de la que pueda conseguir por sí misma. Sin embargo, cometimos el error de pensar que, una vez encontrado el jugador, el árbol de ataque se encargaría de perseguir y matar al mismo, pero no fue así. Entre los factores que consideramos importantes en la fase de ataque no estaba el hecho de perseguir al jugador, ya que al incluirlo, los árboles de ataque tendían a hacer lo mismo que los de patrulla.

Desde el comienzo, nuestra intención ha sido la de crear IAs que sean buenas y que no requieran de más información de la que puedan disponer. Sin embargo, nos parecía lógico que, una vez la IA hubiese patrullado hasta encontrar al jugador, pudiese saber donde se encuentra. Esto dio lugar a reemplazar el terminal de "Avanza" por un terminal algo más complejo, pero con una semántica similar: *Acercar*. Este terminal, como una única acción, trazaría un camino hasta el jugador -mediante $A*$ (Intriago, 2014)- y movería al enemigo hasta una posición adyacente que se aproxime a él.

Otro de los grandes obstáculos que obligabamos a superar a los individuos era enfrentarse a un jugador completamente aleatorio. Tras investigar más concretamente sobre evolucionar IAs en videojuegos roguelike, comprobamos que el hecho de que los individuos vivieran hasta consumir todos los turnos que tenían para ser evaluados era contraproducente, ya que en la mayoría de casos, los enemigos no suelen durar más de unos pocos segundos antes de que el jugador acabe con ellos. Si el jugador al que se enfrentaban nuestras IAs era poco realista, la evolución produciría IAs que podrían matar a una entidad que no representa un jugador real. Por tanto, a la hora de la verdad no tendrían un comportamiento como el observado durante su evolución.

Por este motivo, decidimos hacer más inteligente el movimiento del jugador, pero sin ser excesivo. La nueva implementación del jugador hacía que, de forma eventual, se fuese acercando al enemigo -nuestra IA- y una vez cerca, atacase o bloquease ataques de forma aleatoria, dando prioridad a la acción de atacar.

Como la nueva operación de "Acercar", en conjunción a la nueva IA del jugador, podría facilitar demasiado el trabajo de atacar a la IA, incluimos los nuevos nodos de función VidaIA y VidaJugador, que permitirían a los individuos tomar decisiones en función de su nivel de vida como el del jugador respectivamente. Estos, junto con la acción de Curar, darían lugar a estrategias más complejas e interesantes.

De forma análoga a las mazmorras, con la función de evaluación se pretendía favorecer a los individuos que hiciesen lo que tienen que hacer -se explica más adelante- y penalizar acciones inútiles o contraproducentes. La función de evaluación quedó de esta forma:

En primer lugar, determinamos lo que los individuos debían hacer tanto en patrulla como en ataque. Las acciones que cuentan en patrulla son las casillas que la IA cubre, bien sean las andadas o las exploradas.

$$PuntuacionPatrulla = CasillasAndadasPatrulla+$$

$$+CasillasExploradasPatrulla$$

En el ataque podíamos puntuar más acciones. Como es lógico, lo primero son golpes que la IA intenta realizar. Si una IA no realiza la acción de atacar no puede ser buena, ya que nunca mataría al jugador. También nos parecía relevante puntuar aquellas veces que el jugador intentaba atacar a la IA y está conseguía bloquear el ataque, ya que podría dar lugar a estrategias defensivas.

Para favorecer a los individuos que también en ataque se mueven por la sala, decidimos contar también cuántas casillas cubría la IA en la fase de ataque. Con la intención de que el nuevo nodo Curar tuviese su importancia en la función de fitness, más allá del hecho de que podría mantener con vida a la IA durante más turnos, incluimos el número de veces que la IA conseguía recuperar un punto de vida, no simplemente realizar la acción de Curar, sino que además fuese útil.

Por último, para evitar IAs que constantemente atacasen con la esperanza de que el jugador se posicionase justo delante, penalizamos todos los golpes que la IA intentaba donde no estaba el jugador. Si el jugador estaba delante, pero bloquea el ataque, ese golpe no es fallido.

$$PuntuacionAtaque = Golpes+Bloqueos+CasillasAndadasAtaque+$$

$$+Curaciones - GolpesFallados$$

Como se puede observar en ambos bloques, apenas hay acciones que penalizar, por esto, decidimos aplicar un factor que penalizase de forma drástica todo el conjunto de acciones realizadas si no se alcanzaban ciertos hitos.

El primer factor crucial para la patrulla es que la IA encuentre al jugador. Encontrar al jugador con el árbol de patrulla implica que se tiene que dar una secuencia de dos acciones. Primero, la IA debe haber ejecutado la acción SiDetectado y haber tomado la rama de Sí, lo que implica que en ese turno, el jugador está delante de la IA y esta lo ha visto. La segunda acción necesaria es que, tras haber detectado al jugador, las decisiones que tome la IA durante ese turno lleven a ejecutar un nodo CambiaEstado. A está secuencia de acciones lo denominamos el factor de patrulla, que tomará el valor 1 si se ha realizado y 0 en caso contrario. Si una IA no realiza estas dos acciones combinadas, puntuará 0.

El segundo factor es el número de heridas que la IA ha conseguido infringir al jugador. Esto implica que una IA que no consigue herir al jugador también puntúe 0. Sin embargo, una IA que mate al jugador -conseguir herirlo 3 veces- multiplicará por 3 su fitness.

Por último, se tuvieron en cuenta dos factores análogos que evitaban la propagación de individuos demasiado pasivos que no se mueven. Los factores AndarAtaque y AndarPatrulla tomarán valor 0 si la IA no se ha movido en ataque o en patrulla respectivamente, y 1 en caso contrario. Esto provoca que IAs que no se muevan en ambos estados puntuen 0.

$$FactorFinal = FactorAtaque * FactorPatrulla*$$

$$*AndarAtaque * AndarPatrulla$$

La conclusión es que un individuo debe realizar todas estas acciones para poder puntuar. Después de mucha investigación hemos concluido que esta función de fitness debe ser muy estricta para conseguir que los individuos hagan lo que tienen que hacer y una vez lo han hecho, se les puntúe acorde a la calidad de las acciones realizadas. Por ello, consideramos una condición necesaria aplicar elitismo en la evolución. El elitismo evita que se pierdan aquellos individuos con árboles que cumplen todos los hitos.

Sin embargo, el elitismo por sí mismo no es suficiente para conseguir individuos que superen este corte tan estricto. Es condición indispensable que exista variedad en la población, lo que nos obligó a utilizar

grandes tamaños de población, de 50 o 100 individuos mínimo. El hecho de utilizar selección por torneo también ha afectado positivamente al control de la diversidad mediante análisis de la presión selectiva.

Como se mencionó en la introducción de este apartado, la función de evaluación de la IAs obligaba a probar cada individuo sobre varios mapas, lo que consumía la mayor parte de nuestro tiempo a la hora de realizar pruebas. Conseguimos reducir de forma significativa -hasta 120 veces menos- el tiempo que se tardaba en evaluar la población realizando dos simples mejoras.

La primera y más básica es utilizar un conjunto de individuos marcados. Cada vez que un individuo sufre una modificación, bien sea por haber sido cruzado, mutado o se le haya cortado alguna rama debido al control de bloating, en lugar de ser evaluado, se marca para su posterior evaluación. Así, si un individuo pasa por un cruce y luego es mutado, sólo se evalúa una vez. De la misma forma, un individuo que atraviesa una generación sin sufrir cambios, no necesita ser reevaluado.

Las segunda y más importante optimización es la paralelización de la evaluación del fitness. En lugar de probar a un individuo sobre cada uno de los mapas de forma secuencial, se ejecuta cada simulación de cada mapa por separado y se obtiene el fitness en cada uno de ellos. Con esto, la función de fitness demora como mucho el tiempo que tarda en simularse el más lento de los mapas, en lugar de la suma de todos ellos.

Finalmente, la función de evaluación queda como sigue:

$$Evaluacion = FactorFinal*(PuntuacionPatrulla+PuntuacionAtaque)$$

Estos parámetros significan lo siguiente:

Para conseguir que la función de evaluación haya obtenido resultados importantes, además del cambio de algunas operaciones fueron fundamentales varias optimizaciones que realizamos.

El principal problema que nos encontramos en la nueva función de evaluación es que debía ser muy exigente para evitar evolucionar individuos con malas estrategias. Debido a esto, consideramos una condición necesaria aplicar elitismo en la evolución.

Debido a esta función restrictiva, también nos dimos cuenta que era necesario que la población fuera suficientemente grande ya que, sin variedad, no funcionaba bien.

La necesidad de tener poblaciones grandes acrecentaba aún más nuestro problema de tiempos, que estaba del orden de los quince o veinte minutos por generación para poblaciones de 100 individuos. Por esta

Variable	**Significado**
FactorAtaque	Número de heridas infligidas
FactorPatrulla	1 si ocurre siDetectado - CambiaEstado, 0 si no
AndarAtaque	1 si anda en ataque, 0 en caso contrario
AndarPatrulla	1 si anda en patrulla, 0 en caso contrario
CasillasAndadasPatrulla	Número de casillas andadas en patrulla
CasillasExploradasPatrulla	Número de casillas exploradas en patrulla
Golpes	Número de impactos sobre el jugador
Bloqueos	Número de ataques bloqueados
Curaciones	Número de curaciones con éxito
GolpesFallados	Número de ataques no impactados

Tabla 5.1: Lista de parámetros IA

razón decidimos paralelizar la función de evaluación de manera que para un individuo se evalúan simultáneamente todos los mapas.

5.4. Operadores

5.4.1. Inicializador de la población

Para inicializar la población utilizamos el método LL. Nos decantamos por este ya que es el más completo de los tres. Con este método obtenemos una mezcla de árboles irregulares de diferentes profundidades creadas por el método creciente y árboles más regulares creados por el método completo.

5.4.2. Métodos de selección

De los métodos de selección explicados anteriormente, hemos decidido utilizar la selección por Torneo ya que favorece la supervivencia de los mejores individuos. Otra de las razones de utilizar el método de torneo es que es el único de los implementados que tiene en cuenta el fitness de manera directa. todo esto es necesario porque, como hemos explicado anteriormente, la función de fitness es muy exigente.

5.4.3. Operador de cruce

El operador de cruce que hemos utilizado aquí es el monopunto o simple, ya que no consideramos que uno más complejo fuera a favorecer la mejoría de los individuos.

5.4.4. Operadores de mutación

De los operadores de mutación y después de múltiples pruebas, decidimos decantarnos por la combinada. Observamos que al ser la que más variedad permitía, más favorecía a conseguir las acciones obligatorias que nuestros individuos deben llegar a realizar.

5.4.5. Eliminación de intrones

La eliminación de intrones pretende corregir aquellos árboles en los que se encuentran expresiones redundantes. En programación genética, la existencia de intrones depende de la semántica que tenga el árbol y su contexto. En nuestro caso, muchos de los intrones se pueden detectar y simplificar surgen del hecho de que los árboles deben ser ejecutados durante un turno. Esto implica, por ejemplo, que si un nodo SiBloqueado devuelve NO, mientras la IA no ejecute una acción de giro o retroceso, seguirá bloqueada y el resto de nodos SiBloqueado también seguirán por la rama del NO. A continuación se explican cuáles han sido los intrones que hemos considerado:

a) Nodo PROGN2

1) Acciones contrarias: Si un nodo PROGN2 tiene como hijos dos acciones que realizan acciones contrarias, como por ejemplo GiraIzquierda y GiraDerecha, se sustituye el nodo PROGN2 por un terminal aleatorio.
2) Primer hijo CambiarEstado: Dada la naturaleza del nodo CambiarEstado, que provoca el cambio instantáneo al árbol de ataque, el resto de hijos de PROGN2 no se ejecutarán, así que sustituimos el nodo PROGN2 directamente por un nodo CambiarEstado.

b) Nodo PROGN3

1) Primer hijo CambiarEstado: Caso análogo al de PROGN2, sustituimos el nodo PROGN3 directamente por un nodo CambiarEstado.

c) Nodo tipo condicional

1) Ambos hijos terminales iguales: Si ambas opciones de un nodo de decisión son la misma, siendo estas acciones terminales, semánticamente dicho nodo es equivalente a cualquiera de sus hijos.
2) Hijo equivalente: Si un hijo de un nodo condicional es a su vez el mismo condicional, dicho hijo puede ser sustituido por la rama condicional en la que él mismo se encuentra.

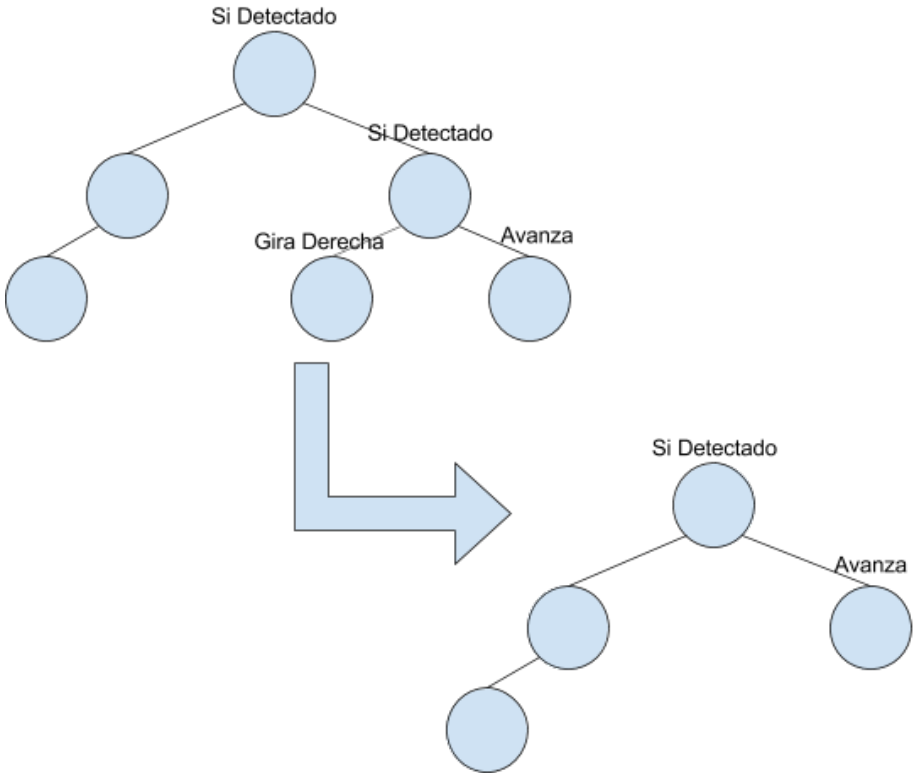

Figura 5.9: Ejemplo de intrones

5.5. Paralelización del fitness

Para conseguir las mejoras que hemos mencionado y alcanzar la función de fitness final, fue fundamental paralelizar la evaluación de los individuos. Para poder hacer esto, decidimos crear un thread por cada mapa de evaluación, de tal forma que se podían evaluar todos los mapas y se tardaba en evaluar todos lo que tardase el mapa más largo.

5.6. Interfaz de usuario

La GUI para la sección de IA es similar a la utilizada en la generación de mazmorras. Mantenemos el visor de gráficas para poder ver la evolución de los individuos. Para poder visualizar de forma clara cómo se comportan los individuos, decidimos crear un visor de la evaluación de los individuos. Con él podemos ver en tiempo real qué movimientos está llevando a cabo el individuo, así como las casillas que ha cubierto o la dirección hacia la que mira 5.10.

Con el objetivo de facilitar también la visualización de los árboles de patrulla y ataque, creamos una ventana adicional 5.11 en la que se muestran los árboles del individuo que está siendo evaluado. Todas estas ventanas decidimos complementarlas con la posibilidad de manipular los atributos del AG, creando un framework usando SFML. Esto se detalla en el siguiente capítulo.

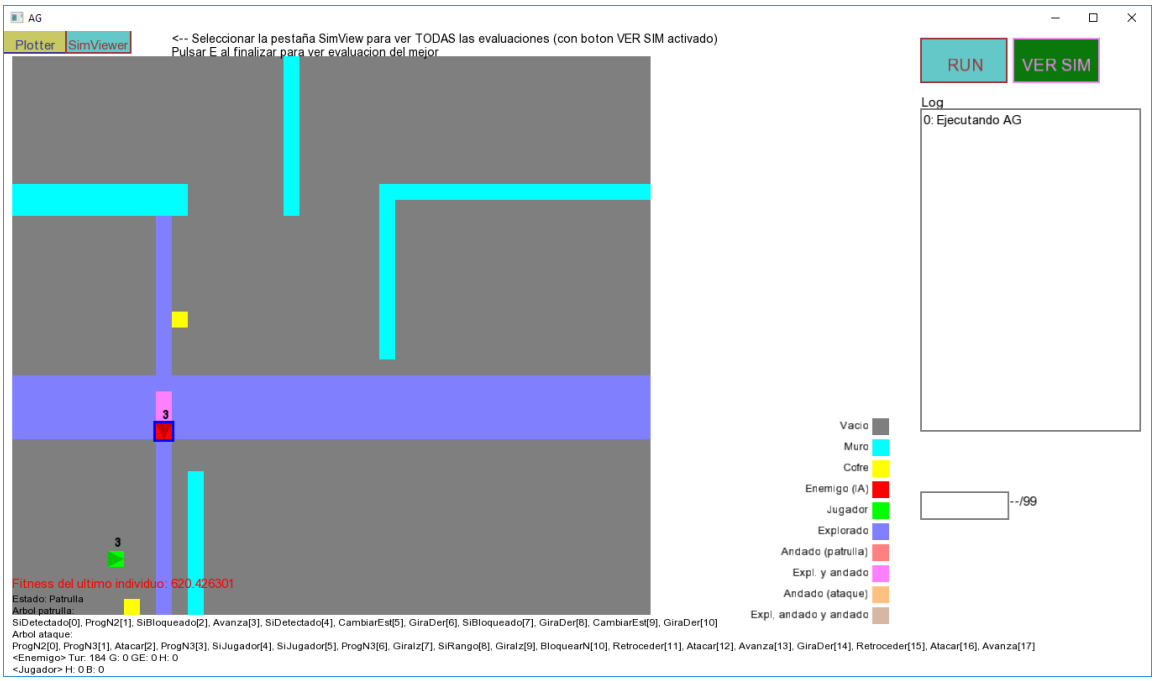

Figura 5.10: Visualización GUI de la parte de la IA

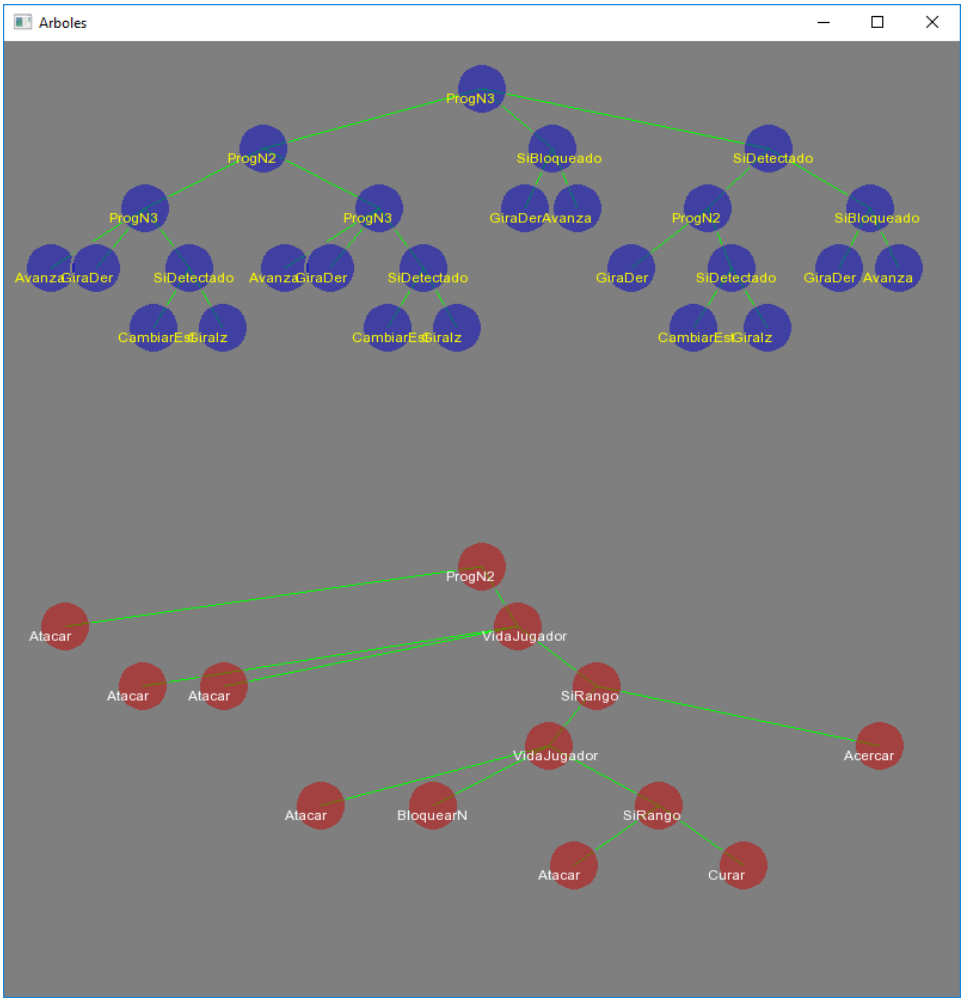

Figura 5.11: Árboles de patrulla y ataque

5.7. Resultados

Como se puede apreciar en la imagen 5.12, la gráfica presenta una mejora clara generación tras generación, tanto en la tendencia de la media como en el mejor individuo. El mejor individuo de la generación -Mejor gen en la leyenda- y el mejor individuo absoluto coinciden perfectamente ya que para las IAs siempre activamos elitismo, por lo tanto el mejor nunca se descarta.

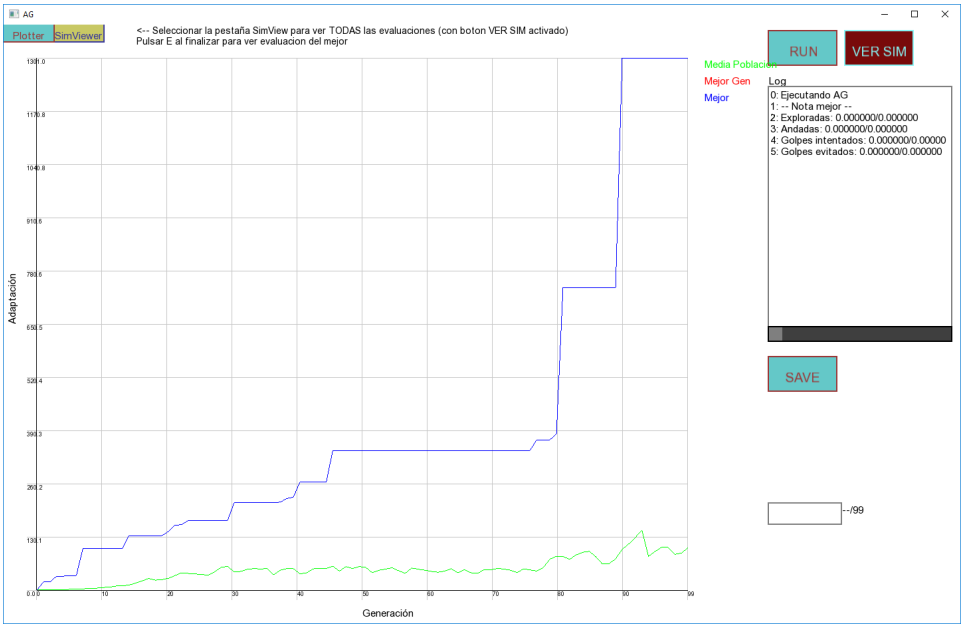

Figura 5.12: Gráfica de evaluación

Sin embargo, si no ponemos elitismo, se aprecia en la figura 5.13 que van variando el mejor y el de la mejor generación. Se obtienen individuos con menor fitness, tanto el mejor como el resto de la población. Por estas razones podemos concluir que el elitismo es un parámetro fundamental en la generación de las IAs.

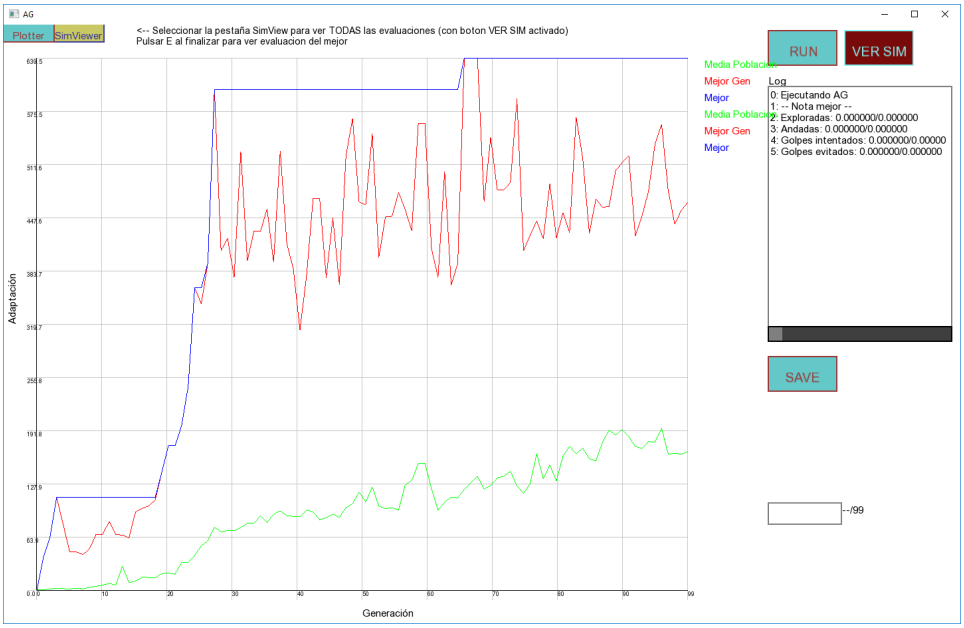

Figura 5.13: Gráfica de evaluación sin elitismo

En la imagen 5.14 podemos apreciar una estrategia algo defensiva ya que tiende a bloquear y curarse bastante. Sin embargo, la que vimos anteriormente, en la figura 5.11 es una mucho más ofensiva.

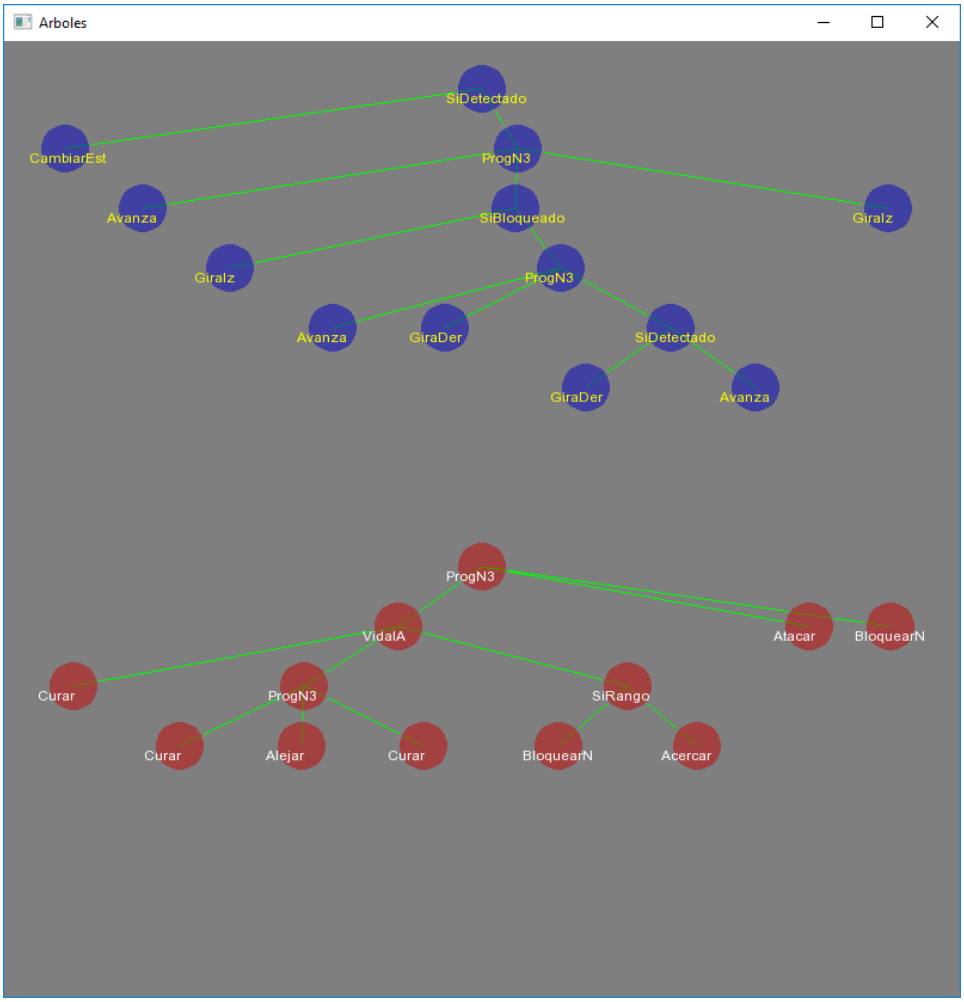

Figura 5.14: Ejemplo de estrategia defensiva

```
Patrulla:
(SIDETECTADO)
    (CambiaEstado)
    (PROGN3
                (Avanza
                (SIBLOQUEADO)
                        (GiraIzquierda)
                        (PROGN3
                                (Avanza
                                GiraDerecha
                                (SIDETECTADO)
                                        (GiraDerecha)
                                        (Avanza))
                GiraIzquierda)

Ataque:
(PROGN3
    ((VIDAIA)
        (Curar)
        (PROGN3 (Curar Alejar Curar))
        ((SIRANGO)
                        (Bloquear)
                        (Acercar))
    Atacar
    Bloquear))
```

En esta otra figura 5.15 podemos apreciar otra estrategia algo más simple, como suelen ser en los videojuegos rogue-like.

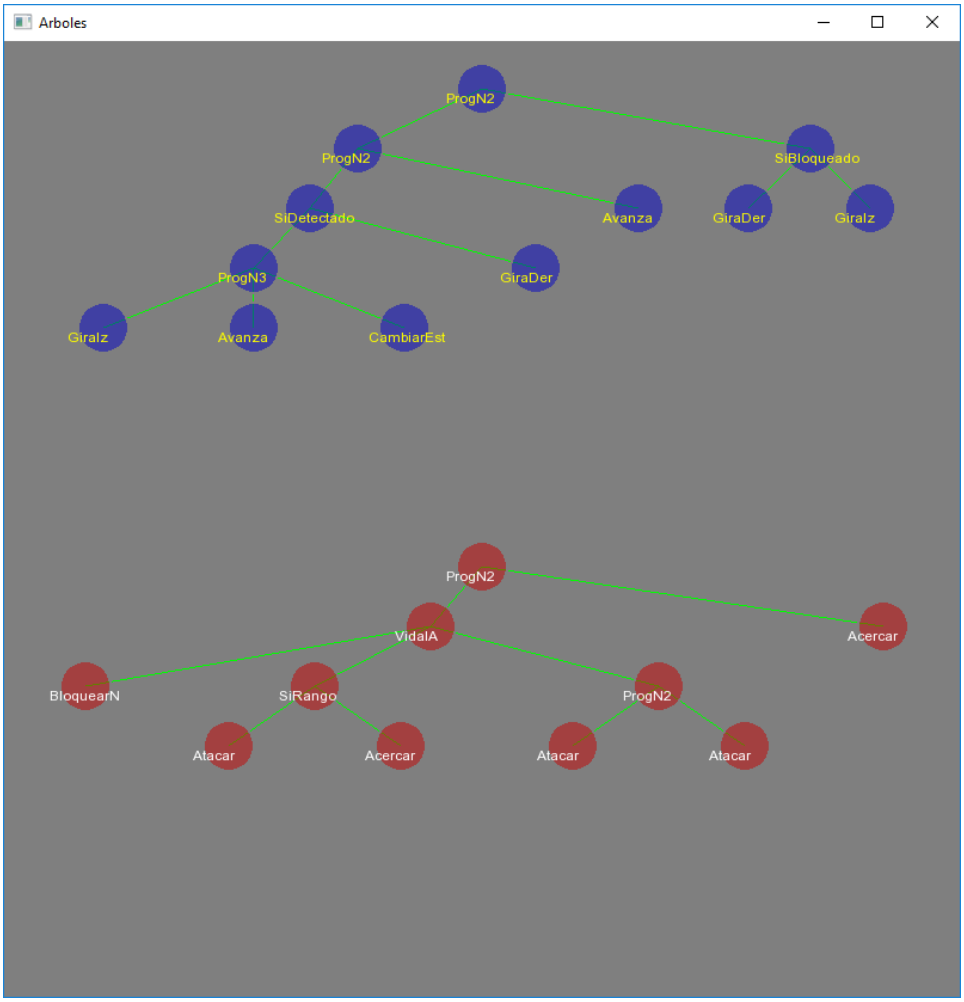

Figura 5.15: Ejemplo de estrategia típica rogue-like

```
Patrulla:
(PROGN2
    ((PROGN2
        ((SIDETECTADO)
            (PROGN3 (GiraIzquierda Avanza CambiaEstado ))
            (GiraDerecha)
        Avanza))
    (SIBLOQUEADO)
        (GiraDerecha)
        (GiraIzquierda)))
```

```
Ataque:
(PROGN2
    ((VIDAIA)
        (Bloquear)
        ((SIRANGO)
                (Atacar)
                (Acercar))
        (PROGN2 (Atacar Atacar))
    Acercar))
```

El algoritmo ha acabado siendo muy robusto ya que no hay ninguna ejecución en la que no se pueda aprovechar ningún individuo. Sin embargo, siempre es necesario retocarlos un poco ya que es muy complicado el conseguir árboles sintácticamente perfectos cuando las funciones y terminales, como es nuestro caso, son bastante complicadas.

Capítulo 6

Framework de pruebas

El principio de la ciencia, casi la definición, es el siguiente: "La prueba de todo conocimiento es el experimento". El experimento es el único juez de la verdad científica.

Richard Feynman, Físico estadounidense y premio Nobel de Física.

Con la intención de que la estructura que ya teníamos fuese más versátil, decidimos ir un paso más allá permitiendo parametrizar el AE de una forma más fácil mediante una ventana en la interfaz desde la que poder establecer todos los parámetros.

Además de los parámetros propios de evolución - población, operadores, probabilidades - permitimos seleccionar desde la interfaz que operaciones pueden ser utilizadas dentro de los árboles.

Otra de las opciones seleccionables era la paralelización de la función de evaluación, lo que reduce de forma drástica el tiempo que tarda en evaluarse cada individuo, a costa eso sí, de no poder ver la simulación sobre los mapas. De está forma, si se pretende analizar la influencia de un nuevo operador sobre la evolución, se puede optar por no paralelizar y ver de forma fácil cuales son las estrategias de los individuos conforme evolucionan. Por otro lado, si ya se han realizado las pruebas necesarias y lo que se busca la obtención de resultados, seleccionando la opción de paralelizar se pueden reducir hasta 120 veces el tiempo total de ejecución, especialmente en poblaciones grandes.

Aun habiendo seleccionado está opción, siempre se permite ver la simulación del mejor individuo obtenido.

- *Tamaño Pob:* Permite seleccionar el número de individuos.
- *Generaciones:* Permite seleccionar el número de generaciones.
- *Prof. Mínima:* Permite seleccionar la profundidad mínima con la que serán construidos los árboles.
- *Prof. Máxima:* Permite seleccionar la profundidad máxima con la que serán construidos los árboles.
- *Elitismo:* Habilitar o no el elitismo.
- *Bloating:* Habilitar o no el control de bloating.
- *Contractividad:* Habilitar o no la contractividad.
- *Prob. Cruce:* Probabilidad de cruzar un individuo en tanto por ciento.
- *Prob. Mutación:* Probabilidad de mutar un individuo en tanto por ciento.
- *Met. Selección:* Permite seleccionar el método de selección.
- *Met. Cruce:* Permite seleccionar el método de cruce.
- *Met. Mutación:* Permite seleccionar el método de mutación.
- *Eval. Paralela:* Habilita o no la evaluación paralela
- *Operaciones:* Permite elegir las operaciones que se incluiran y las que no.

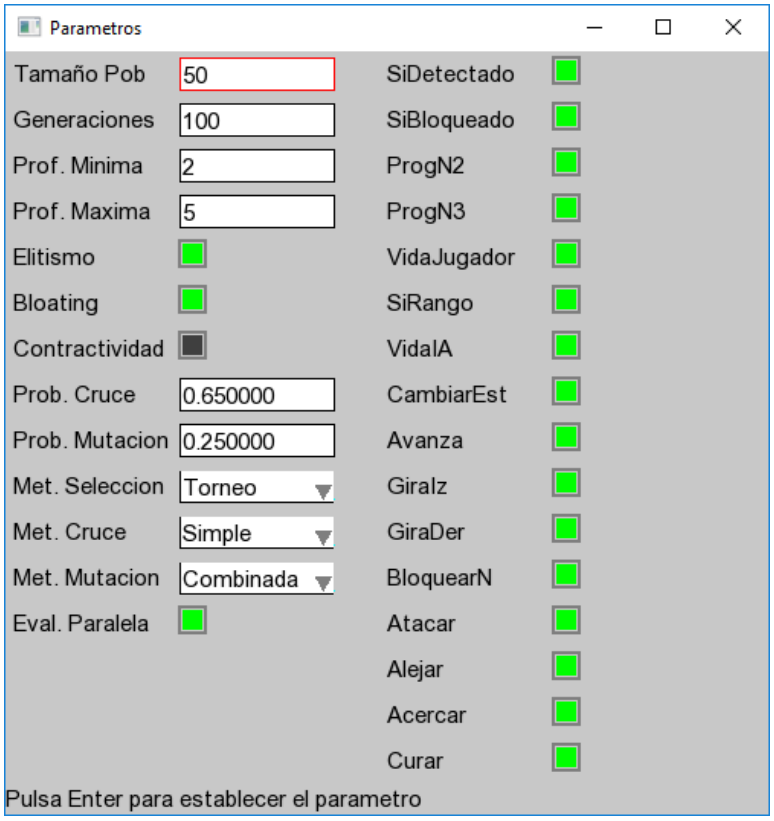

Figura 6.1: Ventana de parametrización del framework

Capítulo 7

Videojuego

Mi trabajo es un juego, un juego muy serio.

Maurits Cornelis Escher, Artista neerlandés.

Aquí explicamos el desarrollo del videojuego como prueba de concepto. Hemos desarrollado un juego estilo rogue-like muy simple e introducido tanto la generación de mazmorras como las inteligencias articiales generadas. Este tipo de videojuegos consiste en explorar una mazmorra, eliminando enemigos y consiguiendo mejoras que te ayuden en el viaje. El juego que hemos implementado carece de historia y es estilo arcade. El jugador debe intentar superar el máximo número de niveles posibles en un intento.

Los controles son muy sencillos. Con las teclas W-A-S-D nos movemos y con las teclas K y L atacamos y bloqueamos. Para abrir los cofres basta con atacarlos. Para moverse entre salas o recoger la llave, es suficiente con pasar por encima. Las mazmorras se generan cada vez que el jugador juega una partida o avanza una fase. Por el contrario las IAs son ?jas ya que los tiempos de ejecución no las hacían viables para que se generen en cada partida. Además, aunque los tiempos hubiesen sido viables, no son tan consistentes como las mazmorras y es arriesgado introducir IAs sin revisarlas previamente.

En nuestro videojuego, el jugador debe explorar infinitos niveles de una mazmorra. Para pasar de un nivel al siguiente, debe llegar al portal que simboliza el fin del nivel. Para poder atravesarlo deberá haber encontrado antes una llave situada en algún lugar del nivel. Durante sus aventuras, el jugador irá explorando salas en

las que habrá portales para moverse, enemigos que le dificultarán su viaje y cofres para ayudarle.

Conforme elimina a sus enemigos, consigue bonus de los cofres, o avanza niveles, el jugador recibe puntos.

No hemos implementado ningún tipo de historia por lo que el objetivo del juego es puramente arcade.

De esta forma, hemos conseguido incluir las técnicas evolutivas para la generación de mapas y de inteligencias artificiales en un juego completo.

7.1. Imágenes del videojuego

Figura 7.1: Pantalla de inicio

Figura 7.2: Ejemplo de una sala de la mazmorra

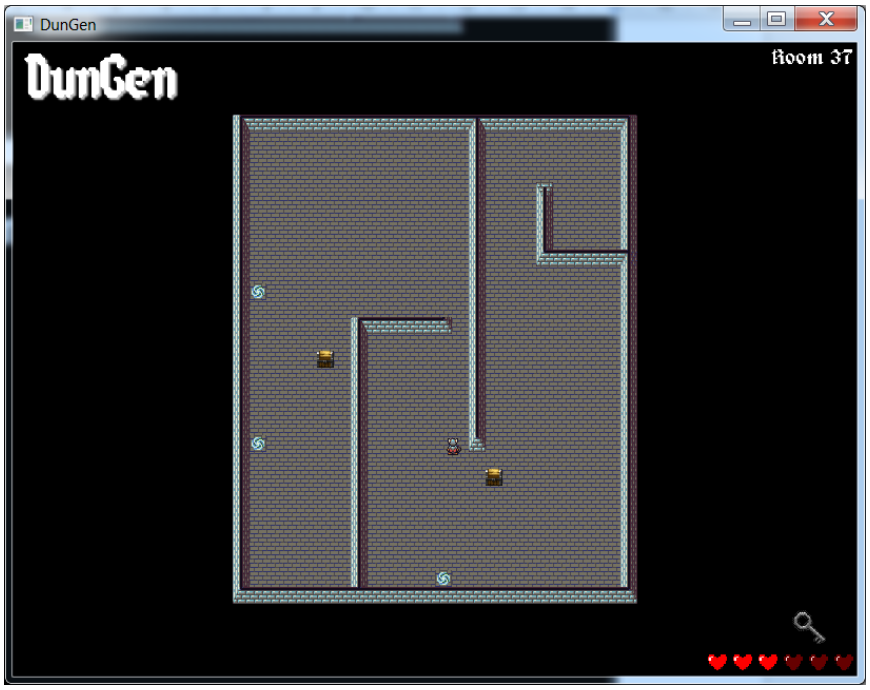

Figura 7.3: Ejemplo de otra sala de la mazmorra

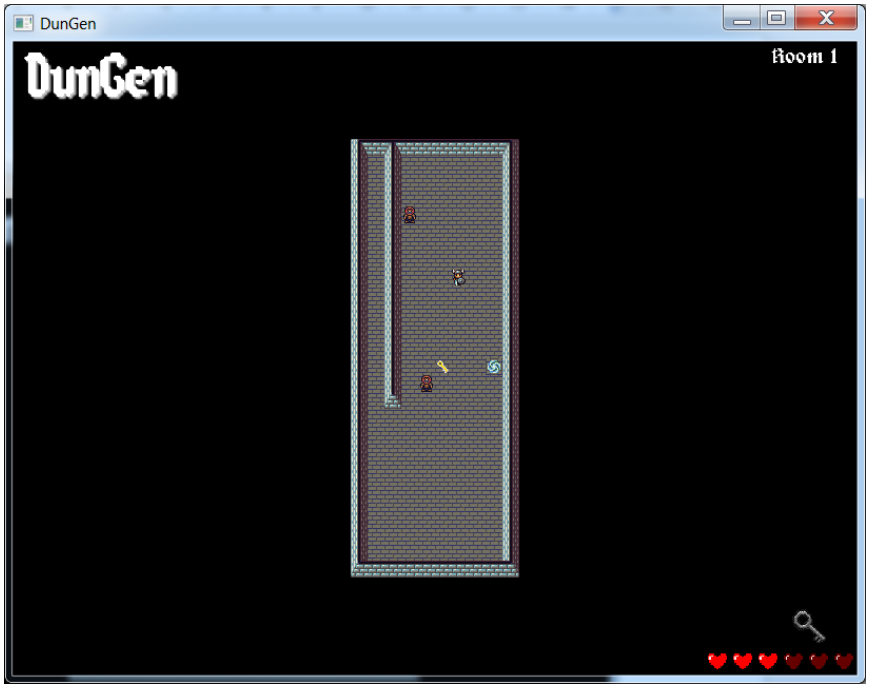

Figura 7.4: Ejemplo de sala con la llave

Figura 7.5: Ejemplo de sala con el portal de fin

Figura 7.6: Ejemplo de pausa

Capítulo 8

Trabajo individual

Una sola flecha puede ser destruida fácilmente pero un haz de flechas es indestructible.

Gengis Kan, Khaqan mongol.

Aquí nos gustaría destacar que todos hemos estado implicados en el proyecto y consideramos que sin el trabajo de todos nosotros y nuestro director, esto no habría sido posible. La propuesta del TFG y los detalles de cómo hemos ido avanzando cuando han salido problemas ha sido un trabajo común del equipo.

Queremos destacar también que nos hemos ayudado mutuamente en la corrección de la memoria. Por tanto, aunque tengamos que especificar el trabajo individual, nos gustaría destacar que todo este proyecto es un trabajo común.

Al ser un proyecto de tamaño considerable, intentamos dividirnos lo mejor posible el trabajo acorde a nuestros fuertes.

Samuel Lapuente:

Al tener más conocimientos y base sobre los mecanismos utilizados en videojuegos para simular inteligencia, fui el encargado principal del desarrollo del bloque de inteligencias artificiales.

Fue tarea mía la realización de las ideas y decisiones tomadas previamente -y durante- así como el principal encargado de programarlo. Para ello utilicé el la ventana que creó mi compañero

Álvaro y decidimos posteriormente convertirla en un framework para algoritmos genéticos.

Como se ha podido apreciar anteriormente, el proceso para alcanzar la función de fitness definitiva no fue rápido y llevó no sólo bastante trabajo programando sino pensando y probando. Fue, sin ninguna duda, la parte más complicada de todo el TFG. Realizamos muchas más pruebas de las que hemos citado para evitar que toda la memoria se hablara de esto. Hasta llegar a alcanzar la propuesta final, estuve investigando ya no sólo en inteligencias artificiales para los videojuegos sino también en el supuesto concreto de los rogue-like.

Como se ha podido leer en la segunda aproximación de la función de fitness del capítulo de la IA, generamos unos mapas de prueba sobre los que simular el comportamiento. Yo me encargué de inventar esos mapas.

La parte más complicada y crítica del trabajo han sido ambas funciones de evaluación. La función del generador de mapas o mazmorras fue complicada pero la conseguimos solventar más rápidamente. Estuve presente y aporté ideas para conseguir esta función. Como he mencionado anteriormente, la de la inteligencia artificial no fue menos. De esta fuí yo el encargado principal pero recibí mucha ayuda de Álvaro para conseguir alcanzar la función definitiva.

Al ser un TFG propuesto, la idea inicial la propusimos a nuestro director Álvaro y yo. El proceso para pulir esa idea fue un trabajo del grupo hasta lograr los resultados aquí expuestos. Comenzamos sin tener una idea definida de videojuego y con la intención de intentar generar también música e historia.

Posteriormente, para la *proof of concept* del videojuego *rogue-like*, aporté ideas de diseño del mismo, como por ejemplo la vista de la cámara o el diseño del mismo. Esta prueba de concepto es un videojuego muy sencillo, sin historia ni grandes personalizaciones. Me encargué de buscar las texturas del videojuego ya que no somos expertos en ello y las que creé de prueba eran bastante malas. De buscar la música del juego así como los efectos de sonido me encargué yo.

De la parte de la memoria me he encargado de redactar la sección de las inteligencias artificiales y, al igual que mis compañeros, de corregir y aportar en todas las demás secciones.

En cuanto a la organización del grupo, mis compañeros decidieron que ejerciera las funciones de una especie de jefe de proyecto. Me encargué de crear el repositorio de github y gestionar algún

problema con commits y merges que surgieron. En la parte de la organización de la memoria intenté ir asignando alguna tarea por orden de prioridad; pero no fueron ni determinantes ni muy importantes ya que mis compañeros se supieron gestionar a la perfección el tiempo. Para la parte del código no necesitamos que nadie ejerciera las funciones de un jefe de proyecto ya que intentamos que la mayor parte del proceso fuera un trabajo común, ya fuera en la puesta de ideas o en el proceso de creación.

Finalmente, me encargué de acabar la maquetación en que empezó Iván antes de dejar el trabajo.

Álvaro Lázaro:

Dada mi experiencia previa con el lenguaje C++, fui encargado de crear el esqueleto que utilizamos para los algoritmos genéticos, así como todo lo relacionado con el apartado de generación de mazmorras, sin embargo, he de destacar que el proceso creativo ha sido fruto de una puesta en común de ideas y muchas pruebas, habitualmente fallidas, hasta dar con una implementación y resultados acordes a nuestras expectativas.

Debido a que fui el miembro del grupo que más uso había hecho de la librería SFML, me encargué de realizar las GUIs tanto en la parte de generación de mapas como en la de IA, con el objetivo de generar un framework que nos permitiera realizar pruebas y observar resultados fácilmente.

Por la naturaleza de los algoritmos evolutivos, son necesarias muchas pruebas antes de encontrar una combinación de parámetros óptima. Por ello, he estado involucrado en todo el proceso de pruebas de ambos bloques principales, desde el testeo de las partes más simples hasta el de las funciones de fitness.

Como se ha podido leer en los apartados previos, las funciones de fitness han sido partes fundamentales en ambos bloques. He investigado sobre ellas y aportado ideas en el proceso de desarrollo de las mismas y, en ocasiones, soluciones.

Al haber implementado la parte de grafos, también he sido el responsable de redactar los detalles de la implementación y el capítulo sobre la generación de mapas.

En lo referente a la implementación final del juego, he realizado una breve investigación sobre el desarrollo de videojuegos en SFML, además de aportar ideas sobre la estructura general del juego.

Para facilitar el uso de la programación genética en lo referente a las inteligencias artificiales, tuve que desarrollar algunos componentes visuales que tuviesen la funcionalidad necesaria para poder parametrizar el algoritmo evolutivo. Esto era necesario para realizar la conversión de este bloque a un framework de trabajo con el que poder investigar, ahora y en el futuro, diferentes combinaciones de operadores genéticos.

He organizado el repositorio en el cual hemos desarrollado todos los proyectos que se han mencionado, especialmente lo referente a limpieza de proyectos obsoletos.

En menor medida, de forma previa a la realización de este proyecto, he recopilado información sobre generación procedural y algunas aplicaciones de algoritmos genéticos sobre videojuegos.

He tomado parte en la revisión de la presente memoria, si bien me gustaría destacar que ésta ha sido una tarea compartida por todos.

Iván Quiros:

A lo largo de todo el desarrollo del proyecto, he sido el encargado principal de buscar trabajos previos que pudieran ayudarnos en la realización de este proyecto. La búsqueda de videojuegos en los cuales alguno de sus contenidos hayan sido generados automáticamente (generación procedural, técnicas evolutivas), la recopilación de información acerca de cómo se generan las IAs en otros videojuegos, cómo se han combinado en anteriores trabajos el uso de Grafos con programación evolutiva o la búsqueda de información previa acerca del uso de Grafos para generar los mapas de un videojuego.

He sido el encargado de redactar parte de la introducción, el capítulo sobre computación evolutiva y programación genética, parte del resumen de este trabajo y lo referente al trabajo relacionado, así como -al igual que mis compañeros- de la corrección de las partes que creímos necesario.

También me encargué de la parte referida a la bibliografía de esta memoria, gestionando que se presentase en el formato correcto, utilizando para ello la aplicación Mendeley.

Por otra parte, me encargué de la parte de la maquetación de la memoria del proyecto utilizando LaTeX. El formato fue algo que consensuamos mis compañeros, nuestro director y yo para que la presentación quedase lo mejor posible.

Iván comenzó con nosotros este proyecto pero antes de acabar el curso decidió que no era para él y acabo dejándolo. Nos parecía importante mencionar su trabajo antes de dejarlo y no atribuirnos un mérito que no nos corresponde.

Capítulo 9

Conclusiones y trabajo futuro

A fin de cuentas, todo es un chiste.

Charles Chaplin, Actor y director británico.

9.1. Conclusiones en la generación de mazmorras

Tras el análisis requerido para conseguir un algoritmo genético que fuese capaz de generar mazmorras acordes a nuestras expectativas, nos parece importante concluir si el trabajo realizado puede ser considerado útil para su implementación real en un videojuego.

En primer lugar, podemos decir que el resultado obtenido en este apartado ha satisfecho con creces nuestras expectativas iniciales, no sólo desde el punto de vista de la calidad de las mazmorras, sino también en lo referente a tiempos de ejecución.

Por otro lado, es cierto que hallar una configuración adecuada del AG, en especial la función de fitness, requiere de mucho tiempo, dedicación y pruebas que deben ser consideradas antes de optar por el uso de técnicas evolutivas.

Dicho esto, creemos que este proceso creativo y de desarrollo depende en gran medida de la experiencia previa y en nuestro caso particular, la configuración óptima de los parámetros para

afinar la evaluación de los mapas ha supuesto un gran esfuerzo y ha requerido una cantidad de tiempo considerable.

Consideramos que la técnica evolutiva utilizada para la generación de mapas nos ha servido para experimentar con una una función de fitness calibrada y ajustada en función de los parámetros seleccionados como relevantes, para definir una representación novedosa de los individuos o mapas, y para experimentar con todos los operadores genéticos adecuados a esta representación.

9.2. Conclusiones en la generación de inteligencias artificiales

En la generación de las IAs ha sido donde más nos ha costado obtener resultados convincentes. Al principio cometimos el error de ser demasiado ambiciosos, como se ha visto en las múltiples aproximaciones para la función de fitness. Sin embargo, después de todas las pruebas y resultados, podemos concluir que es viable obtener inteligencias artificiales lo suficientemente buenas como para incluirlas directamente en un videojuego. Por supuesto, también hay que tener en cuenta que cuanto más complejo sea el videojuego, más complicado es conseguir una función de evaluación adecuada, ya sea por la cantidad de operaciones que habría que incluír así como la complejidad de la simulación de la IA del jugador.

Sin embargo, somos muy optimistas con los resultados que hemos obtenido. Hemos conseguido obtener unas IAs lo suficientemente buenas como para incluirlas en el videojuego. Por tanto, podemos concluir que este trabajo sería tanto viable para generar un videojuego de forma real y susceptible de incorporar muchas mejoras para mejorar el comportamiento general.

9.3. Conclusiones generales

Nuestra apreciación final es que trabajar con algoritmos evolutivos requiere muchas pruebas, conocerlos exhaustivamente e implementar operaciones y estructuras en ocasiones complejas.

Teniendo esto en cuenta, cabe destacar que hemos observado muy buenos resultados en ambas partes y con un tiempo limitado. Tenemos confianza en que, si tuviéramos que volver a realizar algo

similar, nos resultaría más sencillo y conseguiríamos resultados buenos en menos tiempo.

Creemos que las técnicas evolutivas, en comparación con las técnicas procedurales, presentan ventajas en cuanto a la versatilidad y posibilidad de reutilización. Por otra parte, esta facilidad en el método se opone al hecho de definir bien la estructura de los individuos, así como encontrar una forma adecuada de evaluarlos.

Por tanto, podemos concluir que la utilización de algoritmos evolutivos para la generación automática de contenidos para videojuegos es viable y útil. Esto se ha podido apreciar con la prueba de concepto creada que, si bien es simple, sirve para demostrar nuestro enfoque.

9.4. Trabajo futuro

Como se ha insinuado en capítulos anteriores, para nosotros ha sido un gran desafío tener que desarrollar todo nuestro proyecto desde cero, empezando por el proceso creativo, con sus reuniones de *brainstorming* hasta el diseño completo de una interfaz que, si bien es mejorable, cumple nuestras necesidades. Por este motivo y dada la naturaleza del proyecto, aún existen opciones y mejoras sobre nuestro trabajo. Consideramos que el bloque de generación de mazmorras es definitivo. Podría ser la base para crear nuevos tipos de mazmorras o incluso un generador parametrizado para distintos tipos de mapas y mazmorras.

Dentro de está aplicación, podría estudiarse la inclusión de técnicas de generación procedural combinadas con los propios algoritmos evolutivos, con la idea de encontrar soluciones aún mejores. Sin ser un desarrollo completo, hemos podido aplicar este enfoque híbrido en la implementación del videojuego, combinando la generación de mazmorras mediante AG con un método procedural encargado de rellenar cada sala con los elementos necesarios.

Como hemos querido hacer ver con la implementación del framework de pruebas para las IAs, nuestra intención es que puedan ampliarse las opciones para ser utilizadas en distintos tipos de videojuegos, manteniendo la base que proporcionamos.

Analizar el rendimiento de generación de IAs mediante gramáticas evolutivas, en lugar de programación genética, sería otro de los puntos que nos hubiese gustado analizar. Si bien el uso de gramáticas evolutivas es más simple a efectos de implementación, su uso conlleva de una u otra forma traducir las expresiones

obtenidas al formato que requiera la aplicación sobre la que se quieran usar. Por otro lado, con un mismo algoritmo se pueden hacer evolucionar diferentes gramáticas, pudiendo optar por gramáticas de grano fino, con terminales simples, similares a los que hemos utilizado, u otras de grano más grueso, en las que los terminales representan acciones más complejas, como estrategias completas. Estas estrategias pueden ser ofensivas, de evasión, etc.

9.5. Conclusions in dungeon generation:

After the required analysis to achieve a genetic algorithm capable of generating dungeon as we expected them, we think it is important to conclude if the work done here can be considered useful for a real video game implementation.

In the first place, we can say that the results obtained in this part of the project have been greatly satisfactory. The dungeons generated are as we expected not only in the quality but in the time expended to generate them.

On the other hand, it is true that finding the proper configuration for a GA, specially the fitness function, can be hard and it requires a lot of dedication, work and tests. All of this should be kept in mind before opting for the use of evolutionary techniques.

All said, we believe that this creative and development process depends on previous experience highly, and, in our particular case, the consecution of optimal parameters was a big deal and we had to expend a lot of time and resources to get it.

Finally, We consider that this final fitness function allowed us to gain experience in genetic programming and genetic algorithms as well as give a new representation to maps and dungeons in video games.

9.6. Conclusions in artificial intelligence generation:

It is in this part where we had the biggest issues. In the first place we made the mistake of being too ambitious, as it has been seen in the multiple approximations to the fitness function we wrote before. However, after all the tests and results, we can conclude that it is viable to obtain good enough artificial intelligences to be

included in a game. Of course, we all have to keep in mind that, the harder the game is, the harder to achieve a fitness function is.

Nevertheless, we are really optimistic with the results obtained. We finally got AIs good enough to be included in the video game. So, we can say that this work could be also viable to generate AIs for a real video game, and it is still susceptible of improvements to upgrade the general behaviour of the AIs.

9.7. General conclusions:

Our final appreciation is the following: Working with genetic algorithms requires a lot of previous testing, knowing them perfectly and implementing operations and data structures which are, sometimes, complex.

Keeping this in mind, it is important to point out that we have appreciated a lot of good results in both main objectives and with little time. We are confident in, with the proper amount of time, we could get something similar done easily.

We believe that evolutionary techniques, compared to procedural generation techniques, presents several advantages in versatility and reusability. On the contrary, this easy in the method it is opposed to the fact the programmer has to define properly the structure of the individuals, as well as founding a fine way of evaluating them.

So, we can conclude that utilize genetic algorithm for automatic generation of content in video games is viable and useful. This can be better of appreciated with the proof of concept provided in this project. Even though this proof of concept is simple, it is good enough to help us prove our point.

9.8. Future work:

As it has been insinuated in previous chapters, this project was a big deal for us, since we developed everything from scratch. We began with the creative process and ended up with all this. For this reason and the nature of the project, there are still options to develop and upgrades to be made. We consider the dungeon generation block is final. This part could be the foundations to

develop new kinds of dungeons or even a parametric generator for several maps and dungeons.

Inside this app, it could be a matter of study the procedural generation techniques combined with GA. In this section we have started this hybrid approach in the video game, combining the dungeon generation with a simple procedural method which fills the rooms with the proper elements.

As we tried to put the focus on with the framework implementation for the AIs, our intention has always been to give a bunch of tools for game developers.

To analyze the AI generation using grammars instead of genetic programming is another work that can be done, since we did not have the time to do it. However, you have to keep in mind that, even the grammars are easier to implement, their use implies to develop a method to translate this grammars into the app they are meant to be used. On the other side, with the same algorithm and using grammars, this algorithm could potentially evolve several of them. Ones could be similar as the functions and nodes we used, and other a little bit more complex, having terminals representing high level actions and complex strategies. This strategies could be offensive, defensive or whatever the developer wants.

Herramientas utilizadas

Si la única herramienta que tiene es un martillo, pensará que cada problema que surge es un clavo.

Mark Twain, Escritor y humorista estadounidense.

Visual Studio

Desde el comienzo del proyecto, tuvimos claro que realizaríamos el proyecto en lenguaje C++. Pese a ser un lenguaje más complejo, tiene la ventaja de ser más eficiente y muy utilizado para la implementación de videojuegos. Además, existen gran cantidad de librerías que pueden ser incluidas de forma fácil en un proyecto. Por este motivo, decidimos utilizar Visual Studio 2013 como IDE con la que desarrollar todo el proyecto.

SFML

SFML es una librería desarrollada para C++, especialmente utilizada en la creación de videojuegos 2D. Permite el uso de elementos gráficos de forma sencilla y cuenta con pequeños módulos para el manejo de sonidos y conexiones de red.
Con ella hemos podido realizar tanto las interfaces gráficas (pese a no estar directamente diseñada para ello), como el juego final.

Github

Para realizar el trabajo de forma colaborativa, utilizamos un repositorio privado en la plataforma GitHub, lo que nos permite llevar un control de versiones del proyecto, dividirlo en ramas en función de la fase del desarrollo y mantenernos informados de los cambios que ha introducido cada miembro del grupo.

Latex

LaTeX es un sistema de preparación de documentos. Está orientado a la presentación de escritos que requieran de calidad profesional. Se compone de una serie de macros que ayudan a usar el lenguaje TeX (Wikipedia, TeX). Permite,a su vez, separar el contenido del formato del documento. En este trabajo, hemos utilizado LaTeX para la maquetación de la memoria.

Mendeley

Mendeley es una aplicación utilizada para llevar la gestión de las referencias bibliográficas. Permite añadir documentos gestionando la propia aplicación los datos de la referencia o añadir las referencias manualmente, teniendo que meter todos los datos de la referencia. Puede utilizarse para toda clase de artículos, además de para páginas web. También permite el uso de varios formatos de bibliografía diferentes, dependiendo de lo que se quiera mostrar en la referencia.

Bibliografía

Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio.

El Ingenioso Hidalgo de Don Quijote de la Mancha

AB, M. Minecraft. 2011. Disponible en `https://minecraft.net/` (último acceso, Mayo, 2017).

Alcalá, J. Inteligencia artificial en videojuegos. *Laboratorio de Investigación y Desarrollo en Inteligencia Artificial, Departamento de Ciencias e Ingeniería de la Computación*, ????

Algorithms y more. árbol de expansión mínima: Algoritmo de kruskal. 2012. Disponible en `https://jariasf.wordpress.com/2012/04/19/arbol-de-expansion-minima-algoritmo-de-kruskal/` (último acceso, Mayo, 2017).

Araujo, L. y Cervigón, C. *Algoritmos evolutivos: un enfoque práctico*. RA-MA S.A, 2009.

Bullen, T. y Katchabaw, M. Using genetic algorithms to evolve character behaviours in modern video games problem encoding population initialization evaluation selection evolution population replacement. 2004.

Capcom. Street fighter. 1987. Disponible en `http://www.streetfighter.com/` (último acceso, Mayo, 2017).

Corporation, T. Space invaders. 1978. Disponible en `http://www.spaceinvaders.net/` (último acceso, Mayo, 2017).

EcuRed, P. Lisp. 2016. Disponible en https://www.ecured.cu/Lisp (último acceso, Mayo, 2017).

Matemática Aplicada y Estadística, U. D. d. Test de turing. 2004. Disponible en http://matap.dmae.upm.es/cienciaficcion/DIVULGACION/3/TestTuring.htm (último acceso, Mayo, 2017).

Font, J. M., Izquierdo, R., Manrique, D. y Togelius, J. Constrained level generation through grammar-based evolutionary algorithms. 2016.

Gamasutra. Procedural dungeon generation algorithm. 2014. Disponible en http://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php (último acceso, Mayo, 2017).

Games, E. Unreal tournament. 1999. Disponible en https://www.epicgames.com/unrealtournament/ (último acceso, Mayo, 2017).

Games, H. No man's sky. 2016. Disponible en https://www.nomanssky.com/ (último acceso, Mayo, 2017).

García-Ortega, R. y García-Sanchez, P. My life as a sim: evolving unique and engaging life stories using virtual worlds. *ALIFE 14*, 2014.

Guillén Torres, B. El verdadero padre de la inteligencia artificial. 2016. Disponible en https://www.bbvaopenmind.com/el-verdadero-padre-de-la-inteligencia-artificial/ (último acceso, Mayo, 2017).

Intriago, J. Algoritmo a estrella. 2014. Disponible en https://advanceintelligence.wordpress.com/2014/10/07/algoritmo-a-estrella/ (último acceso, Mayo, 2017).

Jackson, D. Evolving defence strategies by genetic programming. *EuroGP*, 2005.

Koza, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

Departamento de Matemática Aplicada, U. P. d. M. Triangulación de delaunay. 2015. Disponible en http://www.dma.fi.upm.es/recursos/aplicaciones/geometria_computacional_y_grafos/web/

triangulaciones/delaunay.html (último acceso, Mayo, 2017).

MUÑOZ, M. Juegos roguelike: Historia y actualidad. 2014. Disponible en http://www.fsgamer.com/juegos-roguelike-historia-y-actualidad-20140414.html (último acceso, Mayo, 2017).

NAMCO. Pacman. 1980. Disponible en http://pacman.com/ (último acceso, Mayo, 2017).

UNIVERSIDAD DE OVIEDO, C. D. I. A. Problema del coloreamiento de un grafo. 1997. Disponible en http://www.aic.uniovi.es/ssii/Tutorial/Grafos.html (último acceso, Mayo, 2017).

PÉREZ, D., TOGELIUS, J., SAMOTHRAKIS, S., ROHLFSHAGEN, P. y LUCAS, S. Automated map generation for the physical travelling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 2013.

SOFTWARE, G. Borderlands. 2009. Disponible en https://borderlandsthegame.com/ (último acceso, Mayo, 2017).

TOGELIUS, J., PREUSS, M., BEUME, N., WESSING, S., HAGELBÄCK, J., YANNAKAKIS, G. N. y GRAPPIOLO, C. Controllable procedural map generation via multiobjective evolution. *Genetic Programming and Evolvable Machines*, 2013.

TOY, M., WICHMAN, G. y ARNOLD, K. Rogue. 1980.

UNITY. Angry bots. 2011.

WIKIPEDIA (TeX). Entrada: "TeX". Disponible en https://es.wikipedia.org/wiki/TeX (último acceso, Mayo, 2017).