TRABAJO FIN DE GRADO

INGENIERÍA INFORMÁTICA

Generador de niveles para videojuegos del género plataforma

Computación evolutiva y Super Mario Bros

Autor

David Téllez Rodríguez

Directores

Alejandro José León Salas

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Julio de 2016

Generador de niveles para videojuegos del género plataforma

Computación evolutiva y Super Mario Bros

Autor
David Téllez Rodríguez

Directores
Alejandro José León Salas

Generador de niveles para videojuegos del género plataforma: Computación evolutiva y Super Mario Bros

David Téllez Rodríguez

Palabras clave: Algoritmo genético, computación evolutiva, generación automática de niveles, optimización, Mario, Infinite Mario Bros, Super Mario Bros, videojuego, género plataformas.

Resumen

En este trabajo se estudia y dan soluciones al problema de generar niveles de forma automática para videojuegos de género plataforma. Las soluciones desarrolladas tendrán en cuenta principalmente al jugador, buscando conseguir una experiencia de juego lo más satisfactoria posible, para lo que se emplean algoritmos del campo de la computación evolutiva que permiten obtener muy buenos resultados en problemas de optimización como éste.

Platform games level generator: Evolutionary computing and Super Mario Bros

David Téllez Rodríguez

Keywords: Genetic algorithm, evolutionary computing, automatic level generation, optimization, Mario, Infinite Mario Bros, Super Mario Bros, videogame, platform genre.

Abstract

In this document we study the problem of automatic level generation for platform games and some solutions are proposed. These solutions will be focused on the player seeking the best game experience possible, for which evolutionary computing algorithms are used, because of their great performance in optimization problems like this one.

Yo, **David Téllez Rodríguez**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela T´ecnica Superior de IIngenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 75907697S, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: David Téllez Rodríguez

Granada a 2 de Julio de 2016.

D. **Nombre Apellido1 Apellido2 (tutor1)**, Profesor del Área XXXX del Departamento YYYY de la Universidad de Granada.

D. **Nombre Apellido1 Apellido2 (tutor2)**, Profesor del Área XXXX del Departamento YYYY de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Título del proyecto, Subtítulo del proyecto***, ha sido realizado bajo su supervisión por **Nombre Apellido1 Apellido2 (alumno)**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de mes de 201 .

Los directores:

Nombre Apellido1 Apellido2 (tutor1) **Nombre Apellido1 Apellido2 (tutor2)**

Agradecimientos

Poner aquí agradecimientos…

<ÌNDICE>

1. Introducción

El mundo de los videojuegos ha experimentado un vertiginoso crecimiento en los últimos años: El número de jugadores ha aumentado y con ello la cantidad de contenido necesaria para satisfacer sus necesidades.

El trabajo de un diseñador de niveles de videojuegos no es sencillo: Debe asegurar unos mínimos de diversión y desafío sin caer en la monotonía y la frustración. Es por ello que la generación automática de contenido por medio de algoritmos es más útil que nunca, ya que permite crear contenido que satisfaga las necesidades de los jugadores sin implicar un gran esfuerzo por parte de los diseñadores de niveles.

Actualmente existen varias soluciones comerciales que están gozando de un gran éxito en géneros como el RPG, exploración o acción-aventura: Videojuegos como *Minecraft*, *No Man's Sky*, *Starbound* o la saga *Diablo* entre otros, disfrutan de una gran popularidad y utilizan técnicas de generación procedural para poblar sus niveles automáticamente. Sin embargo, y aunque existen algunas propuestas realmente interesantes (*Spelunky* [1] o *XO-Planets* [2], por ejemplo), no encontramos videojuegos ampliamente conocidos en el género de plataformas que exploten la generación automática de niveles como sistema de creación de contenido.

El objetivo de este proyecto será explorar uno de los métodos de generación automática de contenido para el género de plataformas, basado en computación evolutiva [3]. Dentro del gran número de técnicas que podemos encontrar en la computación evolutiva nos centraremos en algoritmos genéticos, ya que combinan un alto grado de exploración con una excelente capacidad de optimización. Enfrentaremos distintas variaciones del algoritmo implementado para medir la influencia de dichos cambios en los resultados obtenidos, con el fin de comprobar cómo se desenvuelve esta técnica en el campo de la generación procedural de niveles para videojuegos plataformeros.

2. Análisis de estudios previos

En el campo de la teoría de juegos, el jugador es la pieza más importante. Si se quiere desarrollar contenido para un videojuego, hay que centrarse en que el jugador se divierta sin que llegue a notar monotonía o que lo encuentre adictivo sin llegar a sentir frustración. Para conseguir que los niveles generados cumplan con estos difíciles requisitos, es necesario encontrar la combinación perfecta entre una gran multitud de variables que parametrizan dicho escenario.

En los siguientes documentos se exponen varios estudios previos sobre generación automática de niveles para videojuegos de plataformas, en los cuales se establecen cuáles son los factores que más influyen en la bondad del contenido resultante y utilizando técnicas de computación

evolutiva para encontrar el equilibrio óptimo entre éstos.

2.1. Modeling Player Experience in Super Mario Bros (2009)

En esta publicación [4] se busca conocer qué hace a un videojuego divertido, qué factores dentro del mismo influyen y cómo el jugador reacciona durante la partida.

Las características más importantes que se identifican en el estudio son la cantidad y longitud de huecos (que el jugador tendrá que saltar si no quiere caer al vacío y perder una vida), la existencia de cambios de dirección (otorgan una componente novedosa al nivel) y la posibilidad de desencadenar una sucesión de eventos complejas con una acción simple.

Según el estudio realizado, la frustración, el desafío y la diversión se pueden predecir con una alta eficacia usando las características antes presentadas y prestando atención a las reacciones del jugador.

Con los datos obtenidos se propone una estrategia simple pero eficaz para generar niveles automáticamente: El nivel se compone por bloques estructurales (plataformas, huecos, colinas…) y se va construyendo secuencialmente, empezando por el principio y añadiendo un nuevo elemento estructural en cada paso. De esta forma se puede controlar exhaustivamente la organización de estos elementos.

De esta publicación, se tomará la representación del nivel por bloque estructural para este trabajo, pues se adecúa perfectamente a las representaciones de solución empleadas por las técnicas evolutivas que se explorarán más adelante.

2.2. Towards Automatic Personalized Content Generation for Platform Games (2010)

En esta publicación [5] se sigue con el trabajo realizado en (2.1), aunque en esta ocasión, los autores se centran mucho más en mejorar todo lo posible la experiencia del jugador.

Para ello y teniendo en cuenta los resultados obtenidos en el paper anterior, se usan, además, modelos de aprendizaje que ayudan a medir las reacciones del jugador durante la partida y se construyen en base a cuestionarios, que servirán para obtener información sobre las sensaciones experimentadas por los jugadores al acabar a varios tipos de niveles, con el fin de obtener una selección óptima de parámetros del nivel que influyen en las emociones de los mismos.

Con las características extraídas usando las herramientas descritas en el párrafo anterior, se construyen nuevos modelos que permiten crear niveles personalizados utilizando técnicas de

aprendizaje estadístico (perceptrones simples y multicapa). Estos modelos permiten a los niveles generados gozar de una mayor adaptabilidad de cara a los jugadores, haciendo que se adecue mucho más a lo que cada usuario considera divertido o desafiante.

Este documento ha servido para considerar al usuario como eje central del proyecto: Haremos que el algoritmo de generación de niveles tenga en cuenta el conjunto de características que hacen a un nivel divertido y desafiante para un perfil de jugador ocasional, que busque pasar un buen rato en cada partida sin encontrar un desafío demasiado grande que sólo los más experimentados puedan salvar, pero aportando elementos que le sorprendan y obstáculos sobre los que pueda pasar sin gran dificultad.

2.3. Procedural Level Design for Platform Games (2006)

Los autores de esta publicación [6] proponen un método de generación procedural de niveles bastante novedoso para la época, donde no existía casi desarrollo en este campo para videojuegos del género plataforma.

El sistema de generación procedural que se propone en este documento se basa en componentes, que se forman a su vez a partir de elementos estructurales básicos definidos en el videojuego. Así, identificando estos bloques y combinándolos entre sí de forma inteligente, se pueden lograr unos niveles de muy buena calidad de forma sencilla.

Además de crear la estructura de un nivel, el modelo aquí propuesto permite resolver la problemática de si un nivel es o no finalizable, con un sistema de físicas que controla en todo momento todas las posibles trayectorias de salto, asegurando que el avatar (personaje dentro del juego) puede llegar al final de cada nivel generado.

Como último apunte, se propone una forma de representación de niveles basada en celdas y conexiones entre ellas, estableciendo la forma que tendrá el jugador de pasar de una a otra. De esta manera se pueden definir cambios de dirección o de altitud, elementos que aportan variedad a los niveles generados.

Para este trabajo este paper es muy importante, ya que el diseño de la representación de la solución es de vital importancia para los algoritmos evolutivos. Basándonos en el trabajo que han desarrollado estos autores, se elige una representación de solución basada en bloques estructurales: Plataformas básicas, plataformas con colinas, plataformas con cohetes… que combinando entre sí permitirán definir un nivel de forma muy sencilla. También se toma la idea de asegurar que el jugador pueda acabar cualquier nivel generado, para lo cual se establecerán límites y reglas estructurales (por ejemplo, los huecos en los niveles no podrán superar anchura máxima).

2.4. What is Procedural Content Generation? Mario on the borderline (2011)

Este artículo [7] se centra en clarificar el concepto de generación procedural de contenido

(PCG), para lo cual se presentan ejemplos de lo que no es PCG (contenido generado por jugadores humanos o por algoritmos totalmente aleatorios) y de lo que sí es PCG (algoritmos que usan componentes estocásticos, limitando la aleatoriedad con restricciones).

Pero la parte más importante de esta publicación es la que se centra en la generación de niveles para *Infinite Mario Bros*, plataforma open-source basada en el clásico de Nintendo, Super Mario Bros. Los autores utilizan datos de jugadores con diferentes estilos de juego, creando un modelo determinista sencillo de generación de contenido: El algoritmo recoge la actuación del jugador sobre un primer nivel básico, generado con el generador de niveles de *Infinite Mario Bros*. Conforme el jugador juega, se van registrando sus acciones (botones de salto y sprint/bolas de fuego pulsados) para luego añadir elementos donde éstas ocurrieron (colinas donde el avatar saltó y enemigos donde se pulsó el botón de sprint/bolas de fuego), generando un nivel con estos nuevos elementos. Esto permite generar niveles de forma muy simple, pero no existe una componente de aleatoriedad ni elementos realmente sorprendentes, características clave en el campo de la generación automática de contenido.

Este documento ha servido para definir claramente qué se entiende por PCG y cómo crear algoritmos de este tipo. Además el modelo propuesto permite esclarecer qué no queremos hacer en este trabajo: El algoritmo presentado es demasiado simple en su definición y no incluye elementos aleatorios y sorprendentes, por lo que los niveles generados pueden llegar a ser predecibles y monótonos. Además, a raíz de esta publicación, se ha decidido usar *Infinite Mario Bros* como plataforma software sobre la que desarrollar el algoritmo de generación automática de niveles.

3. Plataforma software

Como se vio en el apartado (2.4), se ha elegido *Infinite Mario Bros* [8] como plataforma software sobre la que desarrollar este proyecto. Consiste en un videojuego para navegadores creado en 2009 por *Markus Persson* [9], ampliamente conocido por ser co-creador del popular sandbox *Minecraft*, posteriormente vendido a *Microsoft* por 2.500 billones de dólares [10]. Además, este software ha sido la base de la competición anual *Mario AI Championship* desde 2010 hasta 2012 [11] y de *Platform AI Competition* (sucesor de éste) en 2013 [12].

En su momento se barajó la opción de desarrollar un videojuego de plataformas propio para este proyecto, aunque después de estudiar los papers consultados y dado que el desarrollo de un videojuego no es el objetivo de este trabajo, me decanté rápidamente por *Infinite Mario Bros*, pues dispone de una serie de características muy deseables:

1. La representación de elementos estructurales está basada en componentes (2.3), perfectos para el trabajo con algoritmos evolutivos.

2. Es un proyecto open-source con licencia *Public Domain* [13] de *Creative Commons*, lo que implica que podemos modificar el proyecto a voluntad.

3. Es un clon de *Super Mario Bros*, videojuego masivamente popular de *Nintendo*© y que cuenta con reglas sencillas, conocidas y bien definidas.

4. Diseño de soluciones técnicas

Para abordar este problema, se empezará desarrollando una solución inicial basada en un algoritmo genético básico con inicialización aleatoria, restricciones sobre parámetros de elementos estructurales y enemigos y con una métrica sencilla de dificultad, que asegura que el jugador pueda terminar el nivel.

Sin embargo, esta primera solución presenta incoherencias estructurales y algunos problemas de monotonía, además de considerar una métrica de dificultad demasiado sencilla. Debido a los problemas anteriores, se desarrolla la segunda solución, corrigiendo incoherencias estructurales y mejorando la forma de medir la dificultad deseada.

Como última iteración del proceso de desarrollo de soluciones técnicas, se propone una tercera y última versión que añade nuevas características al proceso que actúan sobre la monotonía y la diversión de los niveles generados.

4.1 Elementos básicos de nivel

Antes de describir las soluciones desarrolladas pasamos a comentar qué elementos forman un nivel. Dividiremos estos elementos según su función en cuatro grupos [14].

4.1.1. Elementos estructurales

Dentro de este grupo se encuentran los componentes estructurales que se usarán para construir el nivel que recorrerá el avatar:

- **Plataforma:** Consiste en en una simple plataforma rectilínea sin desnivel alguno.

Figura 4.1.1.a: Plataforma

- **Colinas:** Plataforma con una serie de plataformas más pequeñas a distintas alturas

que permiten al personaje subir o descender entre éstas.

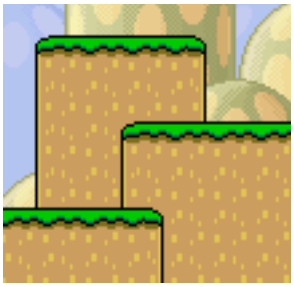


Figura 4.1.1.b: Colina

- **Huecos:** Elementos compuestos por dos plataformas separadas por un vacío que el personaje deberá saltar para seguir adelante.

Figura 4.1.1.c: Hueco

- **Colina con cañón:** Plataforma que puede contener algunos desniveles y que cuenta con un cañón que dispara proyectiles que dañan a Mario si le impactan.

Figura 4.1.1.d: Colina con cañón

Cada uno de estos elementos están formados a su vez por bloques, que se corresponden con los elementos mínimos de construcción que se pueden encontrar en Infinite Mario Bros. Cada bloque se coloca en una matriz que luego se traducirá al nivel jugable final.

4.1.2. Mario

El siguiente elemento básico más importante que podemos encontrar en un nivel es el propio avatar; Mario. Este personaje que lleva jugándose la vida desde 1981 es el protagonista de este videojuego, permitiendo que los jugadores le ayuden a rescatar a la princesa Peach<ref. MarioWiki>.

En lo que a la experiencia de juego se refiere, Mario puede adquirir mejoras repartidas por el nivel que le permitirán derrotar más fácilmente a los enemigos o le impedirán morir en determinadas circunstancias. Durante el desarrollo de la partida, Mario se puede encontrar en una de las siguientes fases:

- **Pequeño**: Es la forma más común de Mario. En esta fase, cualquier daño que sufra hará que pierda una vida.

Figura 4.1.2.a: Mario pequeño.

- **Grande**: En esta forma, Mario puede sufrir daño sin morir, pues si esto ocurre pasará a forma pequeña sin perder una vida en el proceso.

Figura 4.1.2.b: Mario grande.

- **De fuego**: Es la fase más poderosa de todas, ya que Mario disfruta de la mejora anterior, además de poder disparar bolas de fuego que podrán eliminar a algunos enemigos de un golpe y desde la distancia.

Figura 4.1.2.c: Mario de fuego.

Las mejoras repartidas por el nivel permiten a Mario pasar de una fase a otra, lo que hace a estos objetos una ventaja a tener en cuenta durante la partida.

4.1.3. Enemigos

Estos personajes que pueden encontrarse por el nivel intentarán eliminar a Mario por todos los medios a su alcance, presentando un desafío extra para el jugador, que tendrá que lidiar tanto con los obstáculos estructurales como con estos molestos pero peligrosos personajes.

Si Mario toca a un enemigo sin eliminarlo sufrirá una penalización. Esta penalización consistirá en volverse pequeño si Mario se encuentra en tamaño grande

- **Koopa**: Son enemigos de dificultad baja que aparecen con regularidad. Como particularidad, una vez derrotados, Mario puede vuelva a saltar sobre su caparazón para lanzarlo rápidamente hacia adelante, eliminando a cualquier enemigo con el que se cruce. Los koopas pueden aparecer en dos variantes:

Figura 4.1.3.a: Koopa verde.

Figura 4.1.3.b: Koopa rojo.

La única diferencia entre estos dos tipos de koopa reside en que el enemigo verde camina en línea recta hasta que cae o es derrotado, mientras que el rojo hace guardia, evitando precipicios.

- **Goompa**: Son los enemigos más fáciles de eliminar del juego, pero pueden aparecer en grandes números, creando situaciones de peligro. En algunas ocasiones pueden aparecer con alas, lo que les permite saltar mientras caminan por el escenario.

Figura 4.1.3.c: Goompa.

- **Spiny**: Es un enemigo imposible de eliminar por Mario, sólo podrá ser eliminado al caer por un hueco, por lo que es el enemigo de mayor dificultad del juego.

Figura 4.1.3.d: Spiny.

- **Piranha Plant**: Generalmente se encuentran en tuberías, emergiendo de ellas de

forma periódica para atacar a Mario, de modo que si éste toca las fauces de este enemigo sufrirá una *penalización*. Sólo se pueden derrotar con bolas de fuego o lanzándoles caparazones de koopa.

Figura 4.1.3.e: Piranha Plant.

4.1.4. Mejoras (powerups)

Son objetos que aparecen dentro de los ladrillos con interrogación y otorgan mejoras a Mario, haciendo que le sea más sencillo superar el nivel.

- **Hongo rojo**: Hará que Mario crezca de tamaño, de forma que cuando sea golpeado por un enemigo no morirá, sino que volverá a su estado original.

Figura 4.1.4.a: Hongo rojo.

- **Flor**: permite a Mario lanzar bolas de fuego, haciendo más seguro eliminar enemigos. Además, hace que Mario resista un golpe extra de enemigos (ventaja que también otorga el hongo rojo).

Figura 4.1.4.b: Flor.

- **Hongo verde**: Proporciona una vida extra.

Figura 4.1.4.c: Hongo verde.

4.1.5. Ladrillos y monedas

Estos elementos se pueden encontrar en plataformas simples y pueden escon-

der mejoras para Mario. Existen tres tipos de ladrillos:

- **Básico**: Pueden esconder mejoras, monedas o nada. Estos ladrillos pueden ser destruidos por Mario si éste se encuentra en fase grande o de fuego.

Figura 4.1.5.a: Ladrillo básico.

- **Con moneda**: Estos ladrillos esconden una o más monedas que Mario puede recoger al saltar debajo de ellos y los golpea con la cabeza.

Figura 4.1.5.b: Ladrillo con moneda.

- **De piedra**: Cuando Mario recoge todas las monedas de uno de los ladrillos anteriores, pasará a ser un ladrillo de piedra que Mario no podrá romper de ningún modo.

Figura 4.1.5.c: Ladrillo de piedra.

Las monedas que se pueden encontrar por el nivel son la divisa oficial del reino Champiñón, lugar donde transcurre el videojuego *Super Mario Bros* y por tanto en el que se basa la apariencia de *Infinite Mario Bros*. Al recogerlas, se aumenta la puntuación del jugador en la partida.

Figura 4.1.5.d: Moneda.

4.2. Algoritmo base

En el campo de la computación evolutiva se encuentran los algoritmos que basan su estructura en la evolución biológica natural, esto es, la capacidad de un individuo, que se encuentra en una población, de reproducirse y propagar su material genético [1]. El proceso de reproducción de cada individuo está guiado principalmente por su capacidad de adaptación al entorno, aunque también intervienen otros factores como la situación de partida o las mutaciones, que pueden darse de forma aleatoria.

Para crear algoritmos basados en esta idea se deben traducir estos procesos a un entorno de computación, creando el concepto de evolución artificial. Estos algoritmos están formados por componentes evolutivos, que son los equivalentes artificiales de los mecanismos que encontramos en la evolución natural [1]:

- **Función de valoración (*fitness*)**: De suma importancia, pues define cómo de bueno es un miembro de la población. Esta medida es de vital importancia tanto en la fase de selección como en la fase de reemplazamiento.

- **Representación de la solución**: Establece la codificación de cada individuo dentro de la población. Elegir una buena representación permitirá al algoritmo adaptarse al problema y llevar a cabo todo el proceso de forma más natural.

- **Inicialización de la población**: Crea una población inicial de un tamaño determinado. Se pueden utilizar enfoques totalmente aleatorios, dirigidos con alguna heurística o una mezcla entre ambos.

- **Evaluación de la población**: En cada nueva iteración, lo primero que se hace es comprobar cuál es el valor *fitness* de cada miembro de la población.

- **Selección**: Se toman los miembros de la población necesarios para crear uno o más nuevos individuos. Se pueden emplear mecanismos aleatorios, pero lo más recomendable es usar alguna heurística que conduzca la población a contar con individuos de mejor valoración.

- **Reproducción**: Define el mecanismo que se va a usar para obtener nuevos miembros de la población a raíz de los seleccionados en el paso anterior. Esto depende totalmente de la representación de la solución elegida.

- **Reemplazamiento**: Con los nuevos individuos ya generados, se define un mecanismo para incluirlos en la población. Existen varias estrategias que permiten una mayor exploración o una mayor explotación.

Esta categoría de algoritmos disponen de componentes estocásticos para simular la aleatoriedad que se puede encontrar en los procesos de sus equivalentes naturales, lo cual añade a su vez un factor de exploración en el espacio de soluciones del problema muy importante.

En el campo de evolución artificial se pueden encontrar varios grupos de algoritmos [15]:

- **Algoritmos genéticos**: Son los algoritmos pertenecientes a la computación evolutiva más conocidos, pensados para resolver problemas de optimización, contando

con una excelente componente de exploración. Permiten una gran flexibilidad en la representación de la solución y en la codificación de los componentes evolutivos antes descritos.

- **Programación genética**: Caso particular de algoritmos genéticos usados para inducir programas de ordenador de forma automática. La diferencia principal entre estos dos grupos reside en la codificación de la solución: En este caso, la codificación utilizada permite codificar programas.

- **Estrategias evolutivas**: De nuevo una modificación de algoritmos genéticos en el mismo sentido que programación genética. En este caso, las soluciones se codifican como vectores, usando mecanismos de adaptación auto ajustados, por lo que su uso principal es la optimización numérica.

- **Programación evolutiva**: Variante de programación genética en la que se busca inducir programas de inteligencia artificial. Para ello, se considera una población de máquinas de estado finito (FSMs), lo que hace que cada individuo sea único. Esto obliga a cambiar el esquema del algoritmo, eliminando los procesos de selección y reproducción, por lo que sólo existe la mutación como herramienta que permite ejercer cambios en los miembros de la población.

Para este trabajo se ha elegido como técnica a los algoritmos genéticos, ya que son más flexibles, más sencillos de codificar y modificar y se adaptan mucho mejor al problema de generación de niveles de forma automática que el resto.

4.3. Soluciones técnicas

En este apartado se expondrán las implementaciones de las distintas soluciones desarrolladas.

El enfoque que se va a seguir a lo largo del proceso de desarrollo de soluciones es el de *ascensión de colinas*, es decir, una solución será siempre mejor tanto conceptualmente como en calidad de resultados.

4.3.1. Solución inicial

Esta será la solución más sencilla que permite acabar todos los niveles generados y considera una medida simple de dificultad, añadiendo algunas restricciones que influyen en la dificultad y la estructura de los niveles generados.

Para esta solución, los componentes evolutivos son:

- **Función de valoración (*fitness*)**: En esta versión, la más simple de todas, se propone una función *fitness* muy sencilla, que sólo tendrá en cuenta el número de enemigos y su tipo para cada individuo de la población (nivel):

$$enemyDifficulty = \sum_{i=0}^{i=n} d_i \cdot e_i$$

Donde:

- **n** es el total de tipos de enemigos disponible en el nivel (generados en el proceso de inicialización o por mutación).
- $\mathbf{d_i}$ es la dificultad del tipo de enemigo i.
- $\mathbf{e_i}$ es el número de enemigos del tipo i en el nivel.

El valor *fitness* de cada individuo se calcula como la resta entre el resultado anterior y una dificultad deseada en valor absoluto, de modo que conforme más cercano a 0, más se ajustará este valor a la dificultad deseada y por tanto mejor valoración tendrá dentro de la población.

$$individualFitness = \left| enemyDifficulty - desiredDifficulty \right|$$

A partir de estudios empíricos se ha podido concretar que un valor razonable para el índice de dificultad deseada debe variar entre 35 y 100, de modo que si nos acercamos a 35 los niveles generados son más sencillos que cuando rozamos el valor de 100.

- **Representación de la solución**: Se ha optado por una representación de la solución orientada a elementos estructurales, donde ciertos de estos tipos tendrán la posibilidad de incluir enemigos (estos componentes estructurales pasan a denominarse *elementos genéticos*).

Cada individuo codifica sus elementos estructurales como un array de objetos que cuentan con la siguiente estructura:

Componente	Significado
ElementType	Tipo de elemento estructural (entre 1 y 4).
X	Posición X del elemento en el mapa 2D.
Y	Posición Y del elemento en el mapa 2D.
Param1	Anchura del elemento o longitud si se tiene un elemento de tipo “hueco”.
Param2	Tipo de enemigo si el elemento estructural es de tipo “colina” (entre 0 y 4).
Param3	Número de enemigos si el elemento estructural es de tipo “colina”.

Con estos datos se puede codificar el tipo de elemento estructural, sus coordenadas y parámetros extra como altura, anchura u otros más específicos. Una vez se obtenga el nivel generado, estos datos permiten, haciendo uso de la interfaz de *Infinite Mario Bros*, construir el nivel al completo.

- **Inicialización de la población**: Para esta primera solución, la población se inicializa de forma totalmente aleatoria, aunque se consideran algunas restricciones para hacer que el nivel sea finalizable, generando niveles sencillos pero divertidos de jugar:

  - ➔ Los tipos estructurales “huecos” no tendrán una anchura superior a un cierto número de bloques, asegurando que Mario pueda saltarlos sin problemas, aunque se permite la existencia de anchuras que representan un desafío al estar en el límite del alcance del salto del avatar.

  - ➔ Después de un elemento de tipo “hueco” siempre se generará un elemento “plataforma”. Esta decisión hace que el nivel generado sea estructuralmente coherente, pues podría generarse de nuevo un elemento de tipo “hueco” que haría imposible acabar el nivel. Por otro lado, al asegurar un elemento de tipo “plataforma” y no otro se hace el nivel un poco más sencillo de superar.

  - ➔ Sólo el tipo estructural “colina” puede contener enemigos. Esto hace que el nivel sea más sencillo de superar al no disponer de enemigos por todo el mapa. Aunque a primera vista pueda parecer que no habrá demasiados enemigos repartidos por el nivel, el algoritmo genético se encargará de equilibrar el número de este tipo de elementos estructurales para llegar a la dificultad deseada.

- **Evaluación de la población**: Como valoración de cada individuo de la población se utiliza la función *fitness* descrita unas líneas más arriba, de forma que los miembros con una dificultad más cercana a la deseada se situarán mejor en la población.

- **Selección**: Se ha decidido utilizar el método de torneo binario para esta primera solución, esto es, se seleccionan dos padres y se elige el mejor de ellos, proceso que se repite dos veces para contar con dos progenitores. Se cuida que los dos padres no sean el mismo individuo, evitando una pérdida de diversidad.

- **Reproducción**: En este proceso se copia el primer individuo seleccionado y se intercambian un número aleatorio de elementos genéticos del hijo (que en este momento es exactamente igual al primer padre) por los del segundo padre. El número de elementos a intercambiar se define como un entero aleatorio entre 1 y el mínimo

entre el número de elementos genéticos de ambos padres.

Una vez se conoce el número de elementos genéticos a intercambiar, se elige un número aleatorio entre 0 y el mínimo antes mencionado. Este índice pertenecerá al elemento genético que se intercambie entre el segundo padre y el hijo.

- **Mutación**: Se propone un pequeño cambio basado en el tipo y número de enemigos en cada elemento estructural de tipo “colina”. Esta variación consiste en generar nuevos valores para estos dos parámetros, de igual manera que en el proceso de inicialización de la población.

Al ser un proceso de mutación, la probabilidad de realizar estos cambios en un nivel es muy baja. Para que sea un proceso más eficiente, se calcula inicialmente el número de individuos a mutar, aunque serán elegidos de forma aleatoria. Así se ahorra generar una gran cantidad de números aleatorios, disminuyendo el tiempo de ejecución del programa.

- **Reemplazamiento**: Se sigue una estrategia elitista simple: El hijo generado reemplaza al peor individuo de la población actual. Esto permite que la población siempre mejore, aunque si el hijo tiene peor valoración que el peor individuo, hace que decrezca la calidad de la población.

- **Ejemplos de niveles generados**

A continuación se muestran capturas de niveles generados donde se pueden observar las incoherencias estructurales que se comentaban al principio de este apartado, además de los problemas que ocasiona utilizar un generador de niveles casi puramente aleatorio.

➔ Al no controlar del todo la anchura de los elementos estructurales, éstos pueden solaparse entre sí, originando resultados visualmente poco realistas.

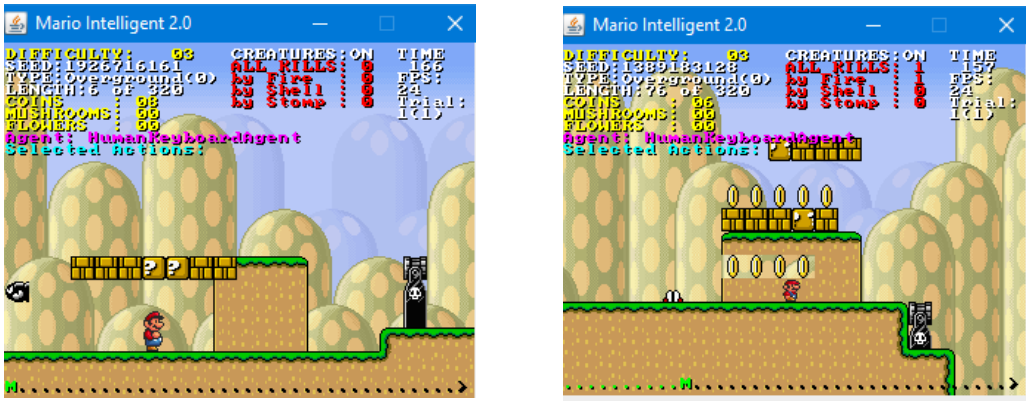


Figura 4.3.1.a: Solapamiento entre plataforma y colina.

➔ Otro ejemplo del punto anterior lo podemos encontrar en la siguiente imagen, donde la corta longitud de plataformas entre saltos hace que sea casi imposible saltarlos.

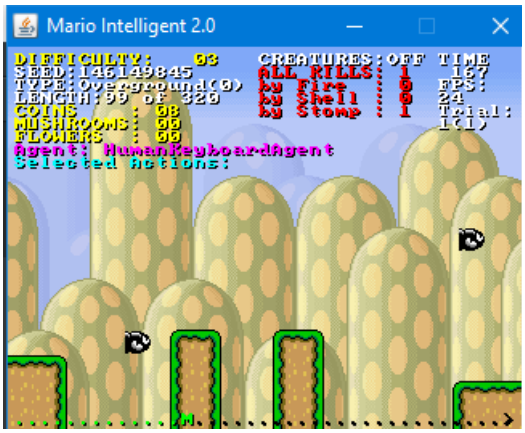
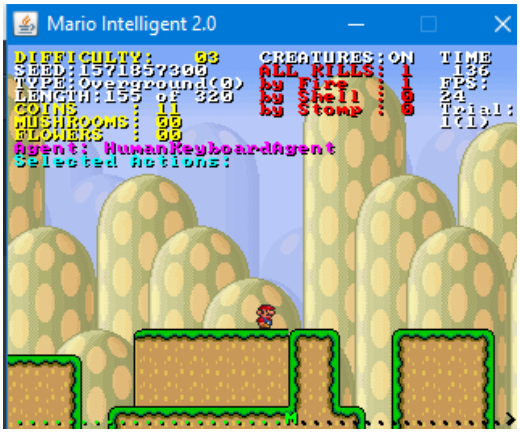


Figura 4.3.1.b: El ancho de estas plataformas es demasiado pequeño.

➔ También se pueden encontrar incoherencias estructurales no tan obvias, pero que también afectan a la jugabilidad. En el caso de la derecha, Mario no puede salir de ahí, haciendo que el nivel no sea finalizable.



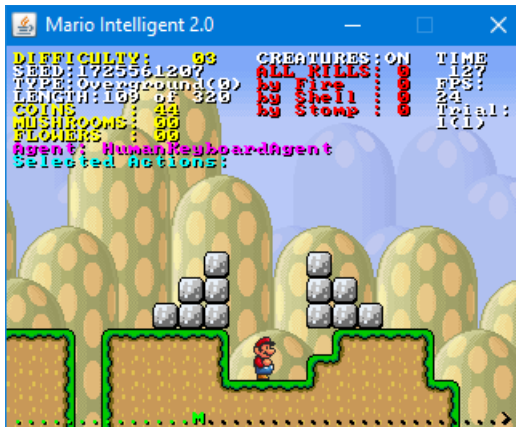


Figura 4.3.1.c: Incoherencias estructurales más sutiles.

➔ Por último, se observan otro tipo de incoherencias estructurales no deseadas que afectan a la monotonía de los niveles generados. Pueden aparecer más de dos elementos estructurales del mismo tipo seguidos, haciendo más monótono el nivel (si se trata de plataformas) o elevendo la dificultad de forma sinérgica (si se tienen elementos de tipo colina o cañón por ejemplo), aspecto no considerado en el algoritmo.

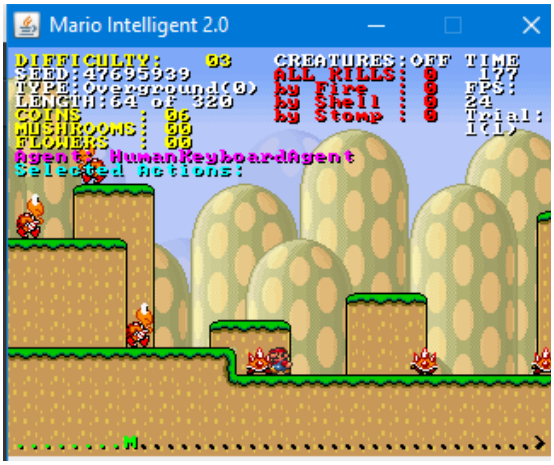


Figura 4.3.1.d: Tres elementos de tipo colina seguidos elevan la dificultad.

4.3.2. Segunda solución

En esta segunda propuesta se eliminan algunas incoherencias estructurales que afectaban a la solución anterior, además de modificar componentes evolutivos orientándolos a mejorar la experiencia de juego, permitiendo generar niveles según el nivel de habilidad (fácil, medio o difícil) del jugador. Se plantea además una función de valoración más completa, acercándonos a lo visto en el apartado (2.1).

- **Función de valoración (*fitness*)**: Se amplía la función usada en la solución inicial para considerar una dificultad estructural. Esto permite tener en cuenta la estructura de nivel como un elemento de dificultad, idea más que necesaria si se quiere adaptar los niveles generados a la habilidad del jugador.

  Por tanto, tendremos una dificultad que nos la dan los enemigos:

$$enemyDifficulty = \sum_{i=0}^{i=n} d_i \cdot e_i$$

  Donde:

  - **n** es el total de tipos de enemigos disponible en el nivel (generados en el proceso de inicialización o por mutación).
  - $\mathbf{d_i}$ es la dificultad del tipo de enemigo i.
  - $\mathbf{e_i}$ es el número de enemigos del tipo i en el nivel.

  Y una dificultad que tomaremos de los elementos estructurales del nivel:

$$structuralDifficulty = (\sum_{j=0}^{j=m} s_i)/m$$

Donde:

- **m** es el total de elementos estructurales en el nivel.
- $\mathbf{s_i}$ es el tipo de elemento estructural (1-4), ordenados de menor a mayor según la dificultad que supone para el jugador el sobrepasarlo.

Debido a que la medida de la dificultad estructural tendrá un valor de entre 1 y 4, es necesario ponderarla para que pueda tener un mayor peso en la función de valoración final, ya que como se apuntó en la solución anterior, los valores razonables del índice de dificultad deseada variaban entre 35 y 100. Por esta razón se lleva a cabo el siguiente proceso:

```
Si 1 <= structuralDifficulty => 1.5 entonces
        structuralDifficulty = structuralDifficulty*10

Si no si 2.5 <= structuralDifficulty > 1.5 entonces
         structuralDifficulty = structuralDifficulty*20

Si no entonces
         structuralDifficulty = structuralDifficulty*40
```

De este modo se lleva la dificultad a valores coherentes con la dificultad de enemigos y además se le da más importancia conforme más difícil sea el nivel, es decir, cuando, en media, las plataformas presentes en el nivel sean más complicadas.

En la solución anterior se definía el valor *fitness* de cada individuo como la resta entre el valor almacenado en *enemyDifficulty* y una dificultad deseada en valor absoluto. Pues bien, en esta solución se añade el valor de structuralDifficulty a enemyDifficulty y se resta este acumulado con la dificultad deseada:

$$individualFitness = |((enemyDifficulty + structuralDifficulty) - desiredDifficulty)|$$

De este modo se tienen en cuenta ambos conceptos con un peso similar, aunque ahora se debe recalcular el valor de dificultad medio óptimo, además, al contar con varios niveles de dificultad, se debe fijar una referencia de dificultad por cada uno de estos niveles.

Al añadir una medida extra al cálculo de la dificultad, los valores de este índice deben ser mayores si se quiere disfrutar de un buen desafío.

Nivel de dificultad	Dificultad media óptima
1 (fácil)	50
2 (medio)	100
3 (difícil)	200

- **Representación de la solución**: Se mantiene la misma representación de solución utilizada en la solución inicial.

- **Inicialización de la población**: Debido a que la función de valoración es más completa, la inicialización debe hacerse teniendo en cuenta los siguientes factores:

  - ➔ Cada dificultad hará que unos tipos estructurales sean más predominantes que otros. Además, según la dificultad elegida, existirán restricciones en el orden de generación de estos elementos.

  - ➔ Ciertos tipos de enemigo serán más comunes que otros o no aparecerán dependiendo del nivel de dificultad seleccionado.

  - ➔ Se mantienen las restricciones de la solución anterior: Sólo el tipo estructural “colina” puede contener enemigos, los tipos estructurales “huecos” no tendrán una anchura superior a un cierto número de bloques y el nivel debe de seguir siendo finalizable.

En este caso, al tener más de un nivel de dificultad, la estructura inicial de los niveles será distinta según el mismo. Los valores de inicialización de los parámetros se escogen aleatoriamente dentro de un intervalo, lo que dota de un poco de variabilidad a cada nivel.

- **Nivel de dificultad 1 (fácil)**:

Es el nivel de dificultad más sencillo, las plataformas son más largas (lo que hace que el número de elementos estructurales presentes en el nivel sea más pequeño), hay poca variedad vertical y la dificultad basada en enemigos es baja: Enemigos sencillos de eliminar y en poco número.

Tipo de elemento \	Ancho	Altitud	Tipo de enemigo	Número de enemigos	Probabilidad

Parámetro					
Hueco	[1, 2]	[10, 11]	--	--	15%
Cañón	[5, 9]	[10, 11]	--	--	20%
Colina	[10, 11]	[10, 11]	[Green Koopa, Red Koopa, Goompa]	[1, 2]	30%
Plataforma	[10, 11]	[10, 11]	--	--	35%

Restricciones adicionales:

- Después de un tipo estructural “hueco” sólo se puede generar uno tipo “plataforma”, de modo que el jugador no tiene que preocuparse de ninguna amenaza cuando acabe de saltar.

- Existe un 35% de probabilidad de que se genere un enemigo “Green Koopa” y un 65% de tener cualquier otro. Esto es así debido a que este tipo de enemigo es el más sencillo de entre los *koopas* y al derrotarlo deja su caparazón para que Mario pueda usarlo para eliminar a otros enemigos, acción muy divertida [4] y útil durante la partida.

- **Nivel de dificultad 2 (medio)**:

En este nivel de dificultad tenemos un compromiso entre los otros dos, resultando en niveles que son finalizables por el jugador medio pero con una dosis de desafío.

Tipo de elemento \ Parámetro	Ancho	Altitud	Tipo de enemigo	Número de enemigos	Probabilidad

Hueco	[4, 5]	[10, 12]	--	--	20%
Cañón	[5, 9]	[10, 12]	--	--	25%
Colina	[8, 9]	[10, 12]	[Green Koopa, Red Koopa, Goompa, Piranha Plant]	[2, 3]	25%
Plataforma	[8, 9]	[10, 12]	--	--	30%

Restricciones adicionales:

➔ Después de un tipo estructural “hueco” sólo se puede generar uno tipo “colina” o “plataforma”. Esto permite añadir algo de dificultad, pues al otro lado de un salto, Mario se puede encontrar con enemigos, de modo habrá que pensar muy bien cuándo saltar.

➔ Existe un 10% de probabilidad de que se genere un enemigo “Spiny”, un 25% de que se genera un “Green Koopa” y un 65% de tener cualquier otro.

- **Nivel de dificultad 3 (difícil)**:

El nivel de dificultad más alto de los tres. Supone un gran desafío para la mayoría de jugadores, con una mayor cantidad y peligrosidad de los enemigos, además de ofrecer un aumento sustancial en la dificultad estructural.

Tipo de elemento \ Parámetro	Ancho	Altitud	Tipo de enemigo	Número de enemigos	Probabilidad
Hueco	[4, 5]	[10, 13]	--	--	35%

Cañón	[5, 9]	[10, 13]	--	--	15%
Colina	[7, 8]	[10, 13]	[Green Koopa, Red Koopa, Goompa, Piranha Plant]	[2, 3]	30%
Plataforma	[7, 8]	[10, 13]	--	--	20%

Restricciones adicionales:

➔ Después de un tipo estructural “hueco” sólo se puede generar uno tipo “cañón”. Debido a que ya existen muchas más posibilidades de generar elementos de tipo “colina” en el nivel, se ha decidido acompañar cada salto con una plataforma de cañón al final. Esto hace que se tenga que escoger muy bien el momento en el que realizar el salto, pues Mario podría resultar dañado en mitad del mismo.

➔ Existe un 35% de probabilidad de que se genere un enemigo “Spiny” y un 65% de tener cualquier otro, lo cual aporta un mayor grado de dificultad a los niveles generados.

- **Evaluación de la población**: Como valoración de cada individuo de la población se utiliza la función *fitness* descrita unas líneas más arriba. En este paso se calcula además de la dificultad de los enemigos, la dificultad estructural asociada a cada nivel.

- **Selección**: Se sigue el mismo proceso que en la solución inicial: torneo binario.

- **Reproducción**: Para esta solución se ha actualizado la operación de cruce:

  - El hijo se inicializa como una copia del primer padre, como se hacía en la solución inicial.

  - Se seleccionan dos índices aleatorios: *childIndex* para el hijo y *p2Index* para el segundo padre, centrados en la mitad de cada individuo y contando con un pequeño desplazamiento que permite añadir algo de diversidad al proceso.

  - Se itera sobre el segundo padre, partiendo de *p2Index*, buscando el primer elemento que encaje con el elemento localizado en la posición *childIndex* del hijo, cumpliendo con las restricciones estructura-

les descritas en el proceso de inicialización. Si se llega al final sin haber seleccionado ningún elemento, se vuelve a elegir un nuevo valor para *p2Index*, aumentando el valor del desplazamiento, para tener más posibilidades de encontrar un elemento compatible con el seleccionado en el hijo.

- Se eliminan los elementos genéticos del hijo que no tendrán sentido, pues a partir de *childIndex* ya no existirán los elementos estructurales "colina" registrados en esta lista, serán sobrescritos con los elementos a copiar del segundo padre.

- Una vez se tienen ambos índices, se copian todos los elementos estructurales del segundo padre en el hijo, utilizando dichos índices como punto de partida del proceso. Si se copia algún elemento estructural de tipo "colina", se registra su índice en el vector de elementos genéticos.

De esta forma, el proceso cruce ahora se centra en la estructura del nivel, aunque también se considera a los enemigos dentro de este proceso de forma indirecta, ya que se pueden copiar elementos estructurales de tipo "colina", arrastrando también a los enemigos presentes en ellos.

- **Mutación**: Se sigue el mismo proceso que en la solución inicial, aunque esta vez han de tenerse en cuenta los vetos a ciertos tipos de enemigo según el nivel de dificultad, generando tantos enemigos como se permita según dicho índice.

- **Reemplazamiento**: Se sigue el mismo proceso empleado en la solución inicial, ya que aporta una componente de explotación necesaria dado el uso de estrategias más exploratorias en los procesos de mutación y reproducción.

- **Ejemplos de niveles generados**

En este caso se pueden observar mejoras visuales respecto a los niveles generados en la solución anterior, reduciendo en gran medida el número de incoherencias encontradas.

1. **Nivel de dificultad *fácil***: Los elementos estructurales tienen mayor anchura, existen menos enemigos y las incoherencias estructurales casi no existen (figura 4.3.2.a). Sin embargo, existen elementos que pueden afectar a la monotonía y diversión, por ejemplo: Si el índice de dificultad deseada es lo suficientemente bajo existirán muchos elementos estructurales de tipo "plataforma", pues son los que menos dificultad suponen para el jugador (figura 4.3.2.b).

Figura 4.3.2.a: Enorme diferencia con los resultados vistos en la solución inicial.

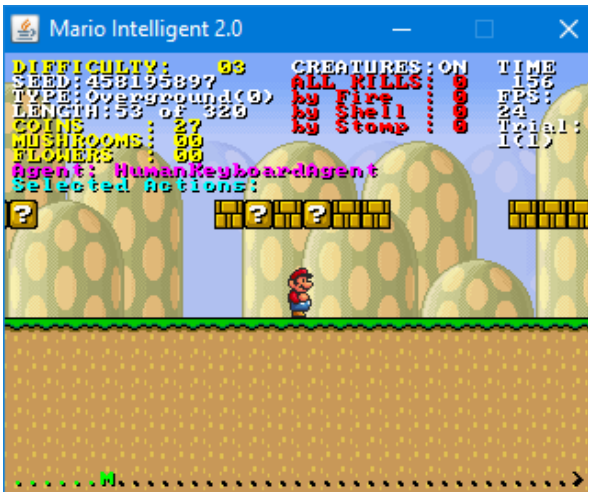


Figura 4.3.2.b: Pueden repetirse elementos estructurales sin límite.

2. **Nivel de dificultad *medio***: En este caso podemos apreciar cómo las plataformas son más cortas y existen un mayor número de elementos que hacen del nivel un desafío a tener en cuenta (figura 4.3.2.c): Existen más enemigos por tipo estructural “colina”, los elementos estructurales “hueco” tienen una mayor anchura, etc…

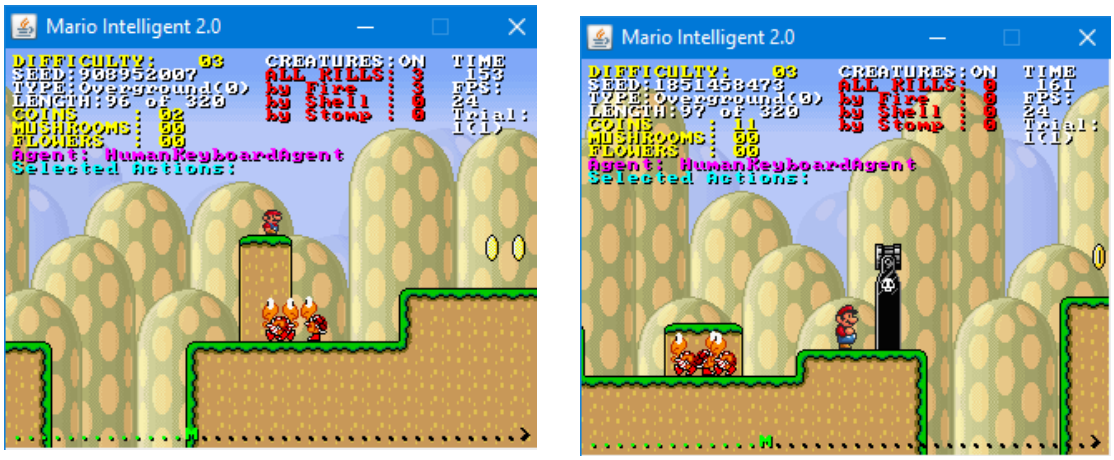


Figura 4.3.2.c: Estos nuevos niveles implican un desafío algo mayor.

3. **Nivel de dificultad *difícil***: En este nivel el algoritmo genético combina elementos que hacen que los niveles generados sean más difíciles de completar, como ya se vio en el punto 4.3.2. En las siguientes imágenes se puede apreciar la influencia de este nuevo índice de dificultad.

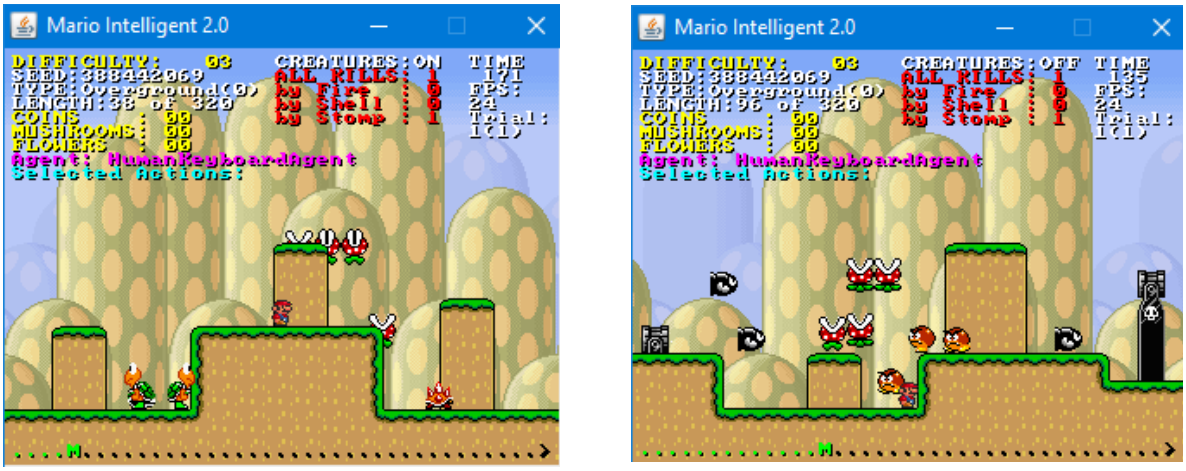


Figura 4.3.2.d: Una mayor cantidad de enemigos y de elementos estructurales difíciles de superar hace más difícil (aunque no imposible) el llegar a finalizar los niveles.

4.3.3. Tercera solución

En esta solución se añaden aún más elementos que permiten afinar la valoración de los niveles generados para adecuarlos al nivel de habilidad del jugador, introduciendo además un arreglo que permite restar monotonía a los resultados.

- **Función de valoración (*fitness*)**: En esta nueva versión, se añade a la dificultad estructural una valoración de la longitud de los elementos estructurales de tipo "hueco" de la siguiente forma:

  Tenemos la dificultad de enemigos como ya conocíamos:

$$enemyDifficulty = \sum_{i=0}^{i=n} d_i \cdot e_i$$

  Donde:
  - **n** es el total de tipos de enemigos disponible en el nivel (generados en el proceso de inicialización o por mutación).
  - $\mathbf{d_i}$ es la dificultad del tipo de enemigo i.
  - $\mathbf{e_i}$ es el número de enemigos del tipo i en el nivel.

  Y la dificultad estructural, a la que añadimos una valoración de la anchura de los elementos estructurales de tipo "hueco":

$$structuralDifficulty = \left(\sum_{j=0}^{j=m} s_i + gapDifficulty(i)\right)/m$$

  Donde:
  - **m** es el total de elementos estructurales en el nivel.
  - $\mathbf{s_i}$ es el tipo de elemento estructural (1-4), ordenados de menor a mayor según la dificultad que supone para el jugador el sobrepasarlo.
  - La función ***gapDifficulty*** se define como:

```
Si i == "hueco" y gapWidth < 4 entonces
        return gapWidth

Si no si i == "hueco" y gapWidth >= 4 entonces
        return gapWidth*2


Si no entonces
        return 0
```

Siendo *gapWidth* la anchura del elemento estructural de tipo “hueco” en la posición *i* del nivel

De esta forma, para el nivel de dificultad “fácil”, este nuevo valor no es demasiado grande, pues la longitud de un elemento estructural de tipo “hueco” no supone problema para el jugador, pero en niveles de dificultad superiores sí debe tenerse en cuenta, pues los huecos son más anchos y en al otro lado del salto se pueden encontrar elementos de tipo “colina” o “cañón”, los cuales representan una gran amenaza para Mario.

El cálculo final de valor *fitness* de cada individuo se calcula como ya se hacía en la solución anterior:

$$individualFitness = |((enemyDifficulty + structuralDifficulty) - desiredDifficulty)|$$

- **Representación de la solución**: Se sigue la misma representación dada en la segunda solución.

- **Inicialización de la población**: El proceso es el mismo que se usó en la segunda solución desarrollada, añadiendo algunas restricciones extra que mejoran los resultados:

    1. Los niveles serán más cortos en general, reduciendo su longitud en un 20%. Se ha tomado esta decisión debido a que ayuda a que la experiencia de juego sea algo más frenética y adictiva, además, permite subir el índice de dificultad media óptima en general, pues un nivel largo y difícil es mucho más complicado de finalizar que un nivel un poco más difícil pero más corto.

    2. Se comprueba que no se generen más de dos elementos estructurales del mismo tipo. Esto ya ocurría en las dos soluciones ya desarrolladas y, aunque no se daba con demasiada frecuencia, reducía la diversión y la sorpresa que podía experimentar el jugador.

- **Evaluación de la población**: Se emplea el mismo proceso que en la solución anterior, aunque en esta ocasión se tiene en cuenta la nueva longitud de los niveles, pues hasta ahora no era necesario.

- **Selección**: El proceso de selección no cambia de nuevo: Sigue siendo torneo binario.

- **Reproducción**: Sin cambios sobre el descrito en la segunda solución.

- **Mutación**: Mismo procedimiento que en la segunda solución.

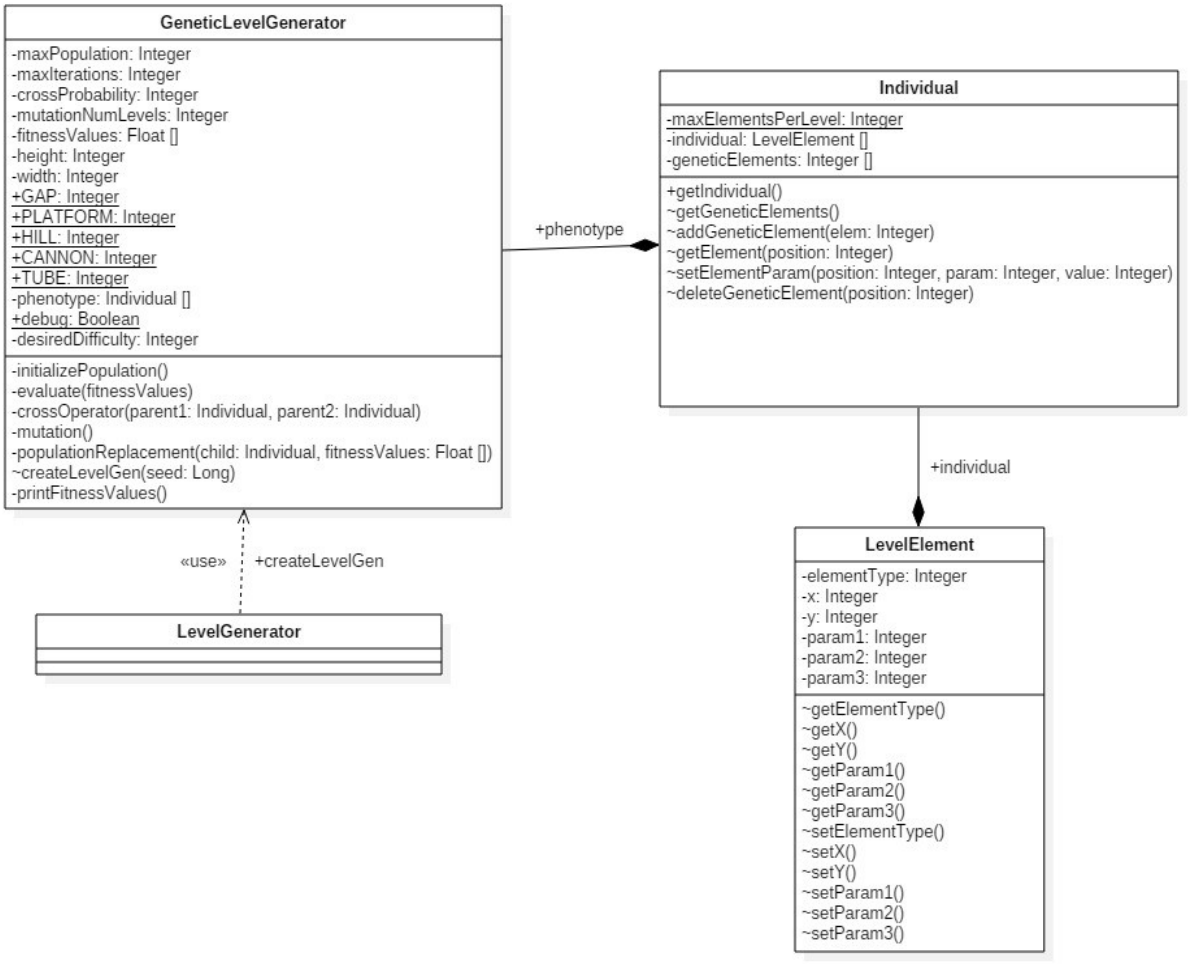
- **Reemplazamiento**: De nuevo sin cambios.

5. Diagrama de clases

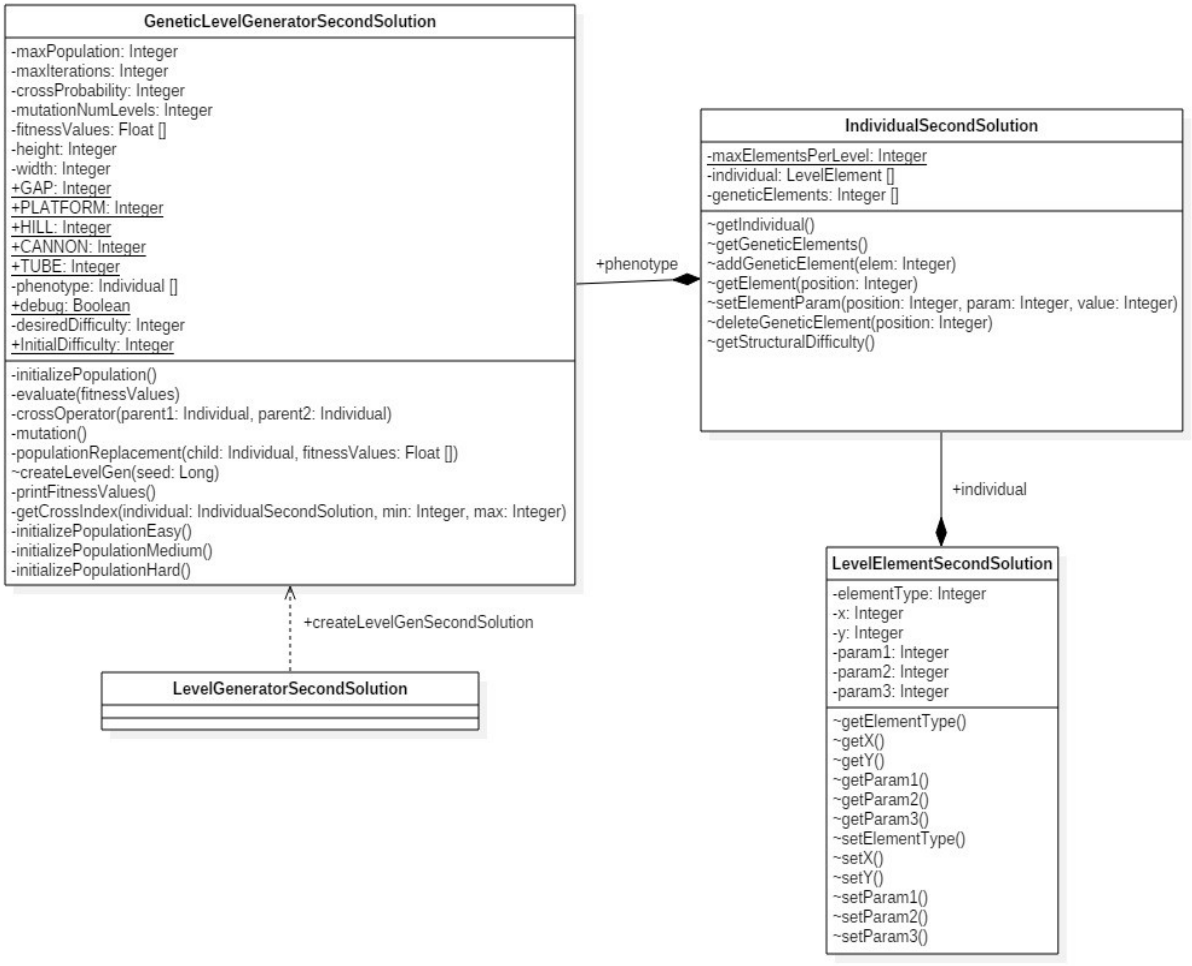
Una vez implementadas todas las soluciones técnicas podemos dar una vista general de la estructura software utilizada.

Se ha decidido hacer que las tres soluciones coexistan en la plataforma, por lo que tendremos tres diagramas distintos:

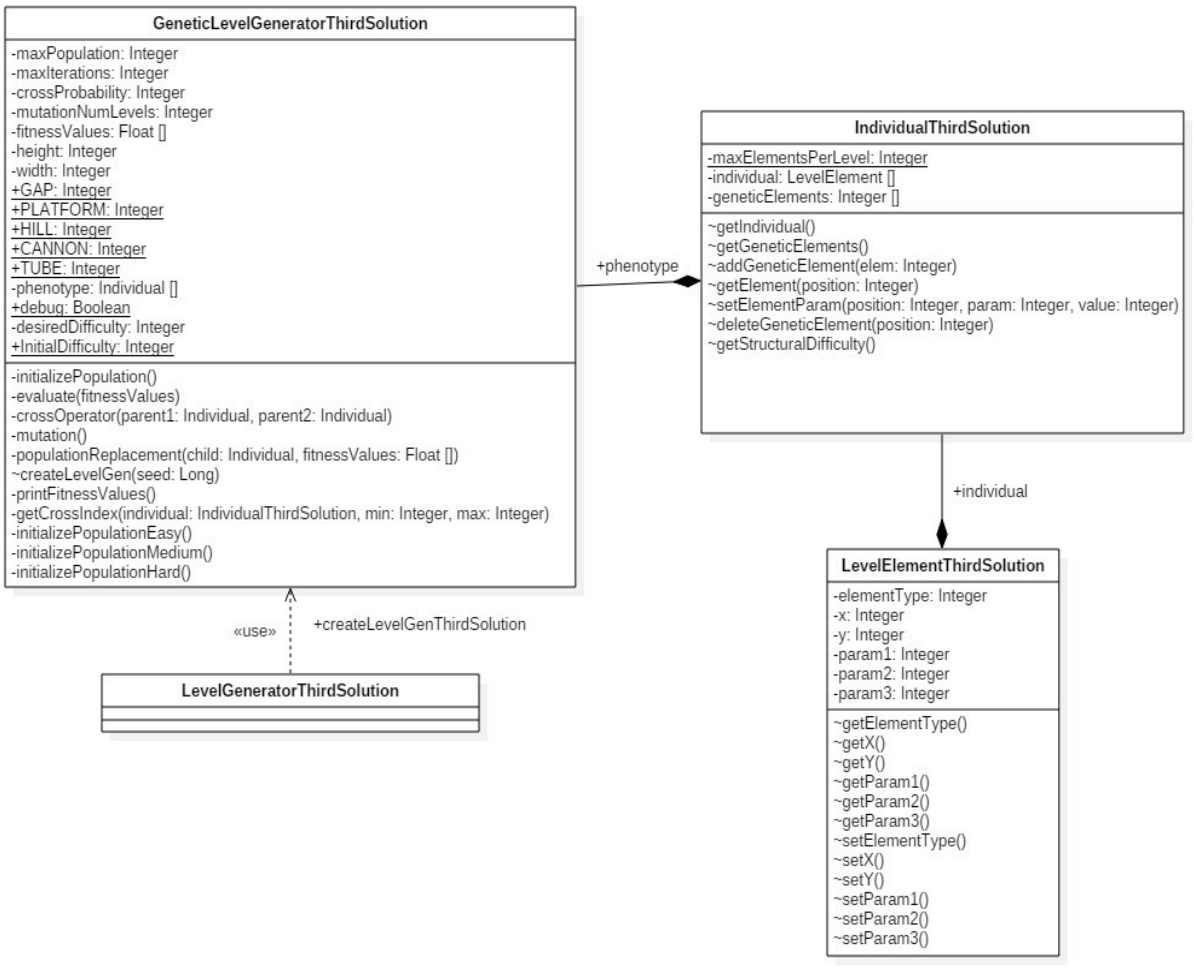
Solución inicial:



Segunda solución:



Tercera solución:



Como se puede apreciar en las imágenes anteriores, se ha optado por una descomposición modular por clases de la funcionalidad. Todas las soluciones desarrolladas comparten la siguiente estructura básica:

➔ **LevelElement**: Contiene la representación de cada tipo estructural, com puesto por su tipo (elementType), posición X, Y dentro del nivel (x e y res pectivamente) y tres parámetros extra que pueden contener información adicional según el tipo del elemento estructural.

➔ **Individual**: Esta clase representa a un individuo dentro de la población. Cada uno de los individuos está formado por un lista (*individual*) de *LevelE lement*, es decir, de elementos estructurales. Esta clase dispone, además, de herramientas para trabajar con los elementos genéticos y a partir de la segunda solución, puede devolver la dificultad estructural.

➔ **GeneticLevelGenerator**: En esta clase es en la que todo el proceso del al gorit-mo genético tiene lugar. Se crea una población (*phenotype*) como vec tor de *Individuals* y dispone de métodos que implementan todos los compo nentes descritos en el apartado (4.2). Además, a partir de la segunda solu ción, se incluyen las operaciones para inicializar la población según un nivel de dificultad y una función extra que permite

obtener un índice aleatorio dentro de un individuo (*getCrossIndex*), útil para el operador de cruce en estas dos implementaciones.

La clase *LevelGenerator* es un caso especial ya que en principio formaba parte de *Infinite Mario Bros*, pero debido a cambios necesarios en las soluciones técnicas segunda y tercera, se ha optado por incluir las clases dentro de los paquetes de cada solución. Inicialmente la finalidad de esta clase es equivalente a la de *GeneticLevelGenerator*: Se generan niveles aleatorios, aleatorizando a su vez en gran medida los parámetros de cada elemento estructural (número de cañones, enemigos, anchura de huecos…). Debido a que en nuestro caso se necesita un mayor control de los parámetros de los elementos estructurales, ha sido necesario crear las clases LevelGenerator para la segunda y tercera solución técnica, pues son más completas en su definición que la inicial, mucho más sencilla.

Como se comentó al principio de este apartado, todas las soluciones técnicas conviven en la plataforma. Para ejecutar una solución u otra se puede llamar al correspondiente *LevelGenerator* dentro del método *init* de la clase *ch.idsia.mario.engine.LevelScene,* clase propia de *Infinite Mario Bros* y que constituye el nexo de unión entre todas las soluciones desarrolladas.

6. Problemas encontrados

Durante el desarrollo de las soluciones técnicas se han encontrado algunos problemas tanto con la plataforma como con el propio desarrollo de las soluciones en sí.

- Pese a que *Infinite Mario Bros* nos ha permitido ahorrar muchísimo tiempo al no tener que programar el videojuego, la implementación de esta plataforma es algo caótica y difícil de leer, por lo que ha sido necesario comprender su funcionamiento y cómo poder modificarla de forma segura para obtener los resultados deseados.

- En la solución inicial veíamos cómo, aunque los niveles eran finalizables no tenían demasiado sentido, pues aparecían bloques donde no deberían, los enemigos muchas veces aparecían en el aire y debido a la libertad de restricciones, se podían ver muchas plataformas del mismo tipo seguidas. En parte se debe a la idea simple que podemos encontrar detrás del desarrollo de esta primera solución aunque también intervienen factores de la interfaz, pues no está pensada para dar control al programador sobre dónde aparecen los enemigos o dónde se dibujan los bloques. Después de un minucioso estudio de la plataforma, se ha logrado controlar todo estos aspectos, consiguiendo resultados como los que se pueden ver en la tercera solución técnica.

7. Trabajo futuro

En este punto se exponen algunas de las dificultades encontradas durante el desarrollo de este trabajo que se han podido sobrepasar y otras que se pueden solventar en un futuro.

Se comentarán aspectos de implementación y conceptos para la tercera solución desarrollada por ser la más completa de todas.

6.1. Utilizar distintos algoritmos de IA

En este trabajo nos hemos centrado en algoritmos genéticos para la implementación de las soluciones técnicas, pero existen otros algoritmos en el campo de la inteligencia artificial que se desenvuelven bien con este tipo de problemas, por ejemplo:

-

6.2. Equilibrio de dificultad

Hasta ahora, los valores propuestos para los índices de dificultad empleados en todas las soluciones técnicas desarrolladas se han obtenido de forma totalmente empírica. Si se desea afinar mucho más la adaptabilidad de los niveles generados a los distintos niveles de dificultad, es necesario establecer valores más ajustados para estos índices. En las siguientes líneas se proponen algunas formas de hacerlo:

1. Crear encuestas y entrevistar a un número razonable de personas que prueben baterías de niveles generados que implementan los distintos niveles de dificultad propuestos. Según los resultados registrados en las encuestas se darán nuevos valores a los índices de dificultad. Quizás esta solución es la más difícil de llevar a cabo, pues hay que preparar las encuestas, encontrar jugadores y sintetizar todos los resultados obtenidos en nuevos índices de dificultad.

2. Emplear agentes automáticos que evalúen niveles, obteniendo datos que se podrán utilizar para equilibrar la dificultad de nivel en cada índice de dificultad. La idea es tener varios agentes preparados, uno por nivel de habilidad de jugador a considerar y preparar una gran batería de niveles para cada dificultad. Cuando los agentes los jueguen, se recogen los datos resultantes para ajustar los índices de dificultad. Resulta más sencilla y rápida que la anterior, en esta ocasión lo más complejo es usar los datos devueltos por los agentes para establecer los nuevos valores de los índices de dificultad.

6.3. Crear menú inicial

Actualmente para obtener un nivel generado automáticamente es necesario ejecutar la clase *ch.idsia.scenarios.Play*. Aunque no es el objetivo de este trabajo, sería buena idea preparar un menú dentro de *Infinite Mario Bros* que permita elegir el nivel de dificultad y genere un nivel aleatorio. Una vez el jugador acabe de jugar el nivel será devuelto a este menú pudiendo elegir entre salir de la aplicación o jugar de nuevo.

6.4. Guardar y cargar niveles generados

Permitir guardar niveles permite utilizarlos como baterías de pruebas o como pequeños "paquetes de desafío", es decir, preparar un conjunto de niveles donde cada uno contará con un índice de dificultad distinto, de forma que represente un desafío para el jugador según se enfoque (se puede empezar por niveles complejos y terminar con otros de dificultad sencilla o variar su orden para provocar sorpresa en el usuario).

TODO LIST:

1. **Referencias. Agregarlas al documento. HECHO**
2. **Imágenes, hay algunas que no se muestran correctamente. HECHO**
3. **Actualizar e incluir diagrama de clases HECHO**
4. **6. Resultados y conclusiones.**
5. **7. Problemas encontrados/desarrollos futuros.**
6. **Añadir estilos a la memoria**

IDEA POR SI QUEDA TIEMPO: HACER DIAGRAMAS DE ESTOS COMPONENTES EVOLUTIVOS.

8. Referencias

[1] «Spelunky is a unique platformer with randomized levels that offer a challenging new experience each time you play», Wiki de Spelunky. http://spelunky.wikia.com/wiki/Spelunky_Wiki

[2] «Frantic rogue-lite platformer with dark-neon and randomly generated planets to explore and destroy», BohFam, creador de XO-Planets. https://bohfam.itch.io/xo-planets

[3] http://sci2s.ugr.es/graduateCourses/Metaheuristicas (Temas 1 y 5).

[4] [2.1] C. Pedersen, J. Togelius y G. N. Yannakakis, «Modeling player experience in Super Mario Bros,» de IEEE Symposium on CIG, 2009. https://www.researchgate.net/publication/224603701_Modeling_player_experience_in_Super_Mario_Bros

[5] [2.2] N. Shaker, G. Yannakakis y J. Togelius, «Towards Automatic Personalized Content Generation for Platform Games,» de AAAI Artificial Intelligence and Interactive Digital Entertainment, 2010. http://julian.togelius.com/Shaker2010Towards.pdf

[6] [2.3] K. Compton y M. Mateas, «Procedural Level Design for Platform Games,» de AAAI Artificial Intelligence and Interactive Digital Entertainment, 2006. https://www.aaai.org/Papers/AIIDE/2006/AIIDE06-022.pdf

[7] [2.4] J. Togelius, E. Kastbjerg, D. Schedl y G. N. Yannakakis, «What is Procedural Content Generation? Mario on the borderline,» de PCGames, Bordeaux, France, 2011. http://julian.togelius.com/Togelius2011What.pdf

[8] «Infinite Mario Bros is a Java-based browser game which offers unending 2D platforming action», IndieGames sobre Markus Persson, creador de Infinite Mario Bros. http://indiegames.com/2008/10/browser_game_pick_infinite_mar.html
http://www.mojang.com/notch/mario

[9] «Markus Alexej Persson, more commonly known as Notch, is the creator of *Minecraft* and one of the founders of Mojang AB». http://minecraft.gamepedia.com/Markus_Persson

[10] «A $2.5 billion cash deal with Microsoft catapulted Persson to billionaire ranks», revista Forbes sobre Markus Persson, creador de Infinite Mario Bros. http://www.forbes.com/profile/markus-persson/

[11] Julian Togelius y Sergey Karakovskiy, responsables de «Mario AI Competition (2009, 2012)». http://julian.togelius.com/mariocompetition2009/index.php, http://www.marioai.org/

[12] Julian Togelius, «Platformer AI Competition (2013)». https://groups.google.com/forum/#!topic/mariocompetition/lUrgJs0tcm8

[13] Infinite Mario Bros disfruta de una licencia de tipo Dominio Público (Public Domain) http://pcg.wikidot.com/pcg-games:infinite-mario-bros

[14] The Mario Encyclopedia, información sobre el universo de Mario Bros. http://www.mariowiki.com/

[15] Introduction to Evoultionary Computing, 2nd. Edition, J.E.Smith, A.E. Eiben, Springer.

[16]