UNIVERSIDAD CARLOS III DE MADRID
Departamento de Ingeniería Informática

TRABAJO FIN DE GRADO

Generación de niveles en juegos de plataformas mediante computación evolutiva

Autor: D.ª Beatriz Puerta Hoyas

Tutor: Prof. Dr. D. Gustavo Recio Isasi

Co-Director: Prof. Dr. D. Yago Sáez Achaerandio

Titulación: Grado en Ingeniería Informática

Leganés, Junio de 2012

Título: GENERACIÓN DE NIVELES EN JUEGOS DE PLATAFORMAS MEDIANTE COMPUTACIÓN EVOLUTIVA

Autor: D.ª Beatriz Puerta Hoyas

Tutor: Prof. Dr. D. Gustavo Recio Isasi

Co-Director: Prof. Dr. D. Yago Sáez Achaerandio

TRIBUNAL Nº 7

Presidente: Prof. Dr. D. Ignacio Aedo Cuevas

Secretario: Prof. Dr. D. Francisco Javier Calle Gómez

Vocal: Prof. Dr. D. Mario García Valderas

AGRADECIMIENTOS

Al conjunto de circunstancias que han hecho que hoy me encuentre aquí, viva, siendo parte de un gran universo. Pero, sobre todo, al conjunto de personas que me ha hecho ser quien soy.

Y aunque son muchas las personas a las que me gustaría mencionar, prefiero centrarme en agradecérselo a las principales: mi familia (por el amor incondicional que me demostráis siempre), Yago (porque he tenido una gran suerte al tener un gran tutor) y Lisardo (por ser el mejor novio, amigo y compañero que se puede desear; la vida es más bella gracias a ti).

RESUMEN

Shigeru Miyamoto creó a *Mario* en 1983. Desde ese momento, aquel entrañable fontanero que rescataba a una princesa en apuros mientras avanzaba por niveles llenos de enemigos de lo más variopintos, empezó a desarrollar un gran éxito en aquel mundo, todavía incipiente, que es el de los videojuegos.

Es por ello que el objetivo de este proyecto es la creación de un generador de niveles para juegos de plataformas (más concretamente, para uno de los juegos con mayor tradición dentro de la categoría de videojuegos de plataformas: el *Mario AI* [1]) que consiga realizar conjuntos de niveles de una determinada dificultad y, además, intente ser lo más divertido posible para el jugador. Según la comunidad científica, a lo largo de la historia de los videojuegos todavía no se ha desarrollado y comercializado un generador que cumpla estas características para juegos de plataformas (los juegos que han perfeccionado esta característica han sido de estrategia en su mayoría). Así que se propone un generador basado en la Inteligencia Artificial, más concretamente, que pertenezca a la computación evolutiva (es una rama de la *AI*, inspirada en la evolución biológica) mediante la aplicación de niveles que presentan aleatoriedad con parámetros definidos, para una vez creados dichos niveles, utilizar un algoritmo genético que sea capaz de generar una población cuyos individuos tengan todos la dificultad que se solicite (sin ser los niveles iguales entre sí).

Posteriormente, gracias a la investigación previa realizada, se empleó un agente inteligente (representando el personaje de *Mario*) desarrollado por Héctor Valero en su PFC [2] que era capaz de utilizar los algoritmos genéticos para finalizar casi cualquier nivel que no fuese imposible. Gracias a esta característica (y puesto que con algoritmos genéticos no se podía determinar con seguridad si un nivel era imposible de resolver o no), se empleó tanto una solución basada en la coevolución como un método de evaluación para el algoritmo genético en el que el agente inteligente determinase lo jugable que era dicho nivel, si su dificultad era verdaderamente la que se creía (o había alguna forma de resolver el nivel que, al realizar acciones concretas durante el juego, lo hiciese más sencillo) o si la imposibilidad de resolverlo era debido a su extrema dificultad (por lo que era realmente era un nivel posible pero muy complejo), o en efecto, tenía algún contenido que lo convertía en imposible.

Finalmente, ante los resultados obtenidos y para poder realizar una aportación en aquel mundo del que se hablaba en el primer párrafo, se pretende enviar estos avances al *2012 Mario AI Championship: Level Generation Track*, perteneciente al *IEEE, Computational Intelligence and AI in Game*.

Palabras clave: algoritmo genético, coevolución, computación evolutiva, controlador automático, evolución, generación automática de niveles, Inteligencia Artificial, juegos de plataformas, *Mario*, competición *Mario AI*, *Super Mario Bros*, videojuego.

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN Y OBJETIVOS

1.1. INTRODUCCIÓN

En este capítulo se pretende abordar la temática de este proyecto, explicando para ello qué **objetivos** se buscan cumplir, y qué ha llevado a que se realizase sobre dicha temática (**motivación**). También se explica la **estructura de la memoria**, para facilitar su lectura y comprensión.

1.2. OBJETIVOS Y MOTIVACIÓN

El objetivo de este proyecto es la generación automática de niveles para el videojuego utilizado como banco de pruebas *Mario AI*, se pretende el desarrollo del entorno utilizando para ello técnicas de inteligencia artificial y generando niveles para dicho juego que sean **entretenidos** y el usuario que juegue con él encuentre distintos tipos de niveles que se adapten a su pericia, es decir, que muestren **distintos tipos de dificultad**, para que el usuario aprenda de los niveles y pueda mejorar su interacción con dicho videojuego cada vez que juegue.

¿Por qué *Mario AI*? ¿Por qué esos dos objetivos?

De todas las posibles opciones disponibles para alcanzar el objetivo buscado (creación de niveles en juegos de plataformas), se eligió la utilización del *software* disponible *Mario AI* [1] por dos motivos:

- Mayor tiempo dedicado en el proyecto al **desarrollo de técnicas de Inteligencia Artificial** que permitan una aportación razonable a los conocimientos de generación de niveles en videojuegos. Puesto que la plataforma básica para la realización de pruebas (la base del videojuego) ya está desarrollada, se podrá focalizar el proyecto en mejorar la misma según función de las pruebas que se quieran realizar.

- Posible presentación del proyecto realizado en el congreso estipulado para tal fin: *2012 Mario AI Championship: Level Generation Track*, perteneciente al *IEEE, Computational Intelligence and AI in Game*.

En cuanto a los objetivos comentados, se desea encontrar un generador automático de niveles para juegos de plataformas por la sencilla razón de que este campo está todavía en desarrollo no habiéndose encontrado todavía una **técnica adecuada** en todos los casos ni tampoco se ha desarrollado una **evaluación apropiada** para los niveles generados, por lo que se pretende cumplir estos objetivos proponiendo nuevas soluciones a este problema, aplicando técnicas de **Inteligencia Artificial (en adelante también referido como *AI*)**.

El motivo por el que se proponen técnicas de *AI* es debido a la necesidad de presentar una solución al problema que no sea determinista, pero como no todas las técnicas de este

campo cumplen este factor, se recurrió al empleo de los **algoritmos genéticos y evolutivos**, es decir, a la **computación evolutiva**.

Otra razón por la que se usan estas técnicas (en vez de utilizar, simplemente, aleatoriedad para la generación de niveles) es por el propósito de encontrar **niveles muy variados** entre sí pero de la misma complejidad. Pudiendo de esta forma, tener lotes de niveles, teniendo cada lote una dificultad especificada, y cuyos niveles no se parezcan entre sí.

Por último, se busca que los niveles sean divertidos puesto que un usuario que se divierta (o entretenga) usando un juego o programa, lo usará más a menudo y se pueden llegar a emplear métricas para que esto se cumpla. De esta forma, a su vez, el jugador no sólo se divertirá, sino que podrá aprender de dicho videojuego si este se ha planteado con fines educativos. Igualmente, el propio hecho de realizar este proyecto implica un proceso de aprendizaje en el que se estudian características humanas con tal de encontrar el éxito en el desarrollo de este campo. Si en un futuro se encontrase una técnica adecuada y sencilla para el desarrollo de *AI* en los videojuegos de plataformas, es probable que llegase a comercializarse. Pero es obvio que, para llegar a dar ese paso, esas técnicas empleadas deberán buscar el divertimento del jugador, como es este caso.

1.3. ESTRUCTURA DE LA MEMORIA

Para facilitar la lectura del proyecto, se incluye a continuación un breve resumen de cada capítulo.

En el capítulo 2, se expone con detalle el **planteamiento del problema**, por lo que se realiza un estudio del estado del arte que recoge los artículos de investigación más importantes, buscando todo los trabajos relacionados con la temática del proyecto, para poder incluir un resumen desde el documento más antiguo (1980) hasta el más reciente (Marzo 2012). En este estado del arte se estudia la experiencia del usuario desde el punto de vista del divertimento y, en menor parte, de la educación. También se observa la evolución producida en las técnicas de Inteligencia Artificial, y se comentan los puntos más representativos en la progreso de los videojuegos, aunque al encontrarse este proyecto muy enfocado hacia un videojuego en concreto, teniendo definida ya la plataforma para ello; se resalta más el desarrollo de la *AI* y de la diversión o el entretenimiento del usuario.

Dentro de este capítulo también se explican los requisitos y restricciones necesarios para el desarrollo de este proyecto, llevándose a cabo un análisis de las restricciones aplicables en función de las necesidades expuestas a través de dichos requisitos.

En el capítulo 3, se presenta el diseño de la solución. A su vez, dentro de este capítulo se explica la solución inicial con las mejoras realizadas sobre la misma viéndose cómo dicha idea evoluciona hasta llegar a la solución final, siendo una idea nueva basada en la anterior. De esta forma, se desarrollan los diseños de las dos soluciones principales de Inteligencia Artificial (**algoritmo genético** y **coevolución**). Aunque, tal y como se argumenta en dicho capítulo, una prevalece sobre la otra.

Ambas propuestas de diseño desarrolladas presentan su codificación y algoritmo explicado, con sus diferentes funciones.

Se ha propuesto también otra alternativa más a las otras dos soluciones, basada en el aprendizaje automático, pero se ha rechazado tras la comparativa de ventajas e inconvenientes de distintas soluciones.

Se finaliza este capítulo con la elección y mejora del diseño final, en base a todas las alternativas explicadas y lo sucedido en el capítulo 4.

En el capítulo 4, se muestran los **resultados y la evaluación**, comenzando por la explicación de los métodos de evaluación empleados (puesto que para el sistema coevolutivo, además, se empleó un segundo método). Tras esto, se realiza una comparativa de los métodos y un análisis de los resultados, que llevan a determinar qué algoritmo propuesto da mejores soluciones al problema. Una vez hecho esto, se determina cuál es el mejor método y se comentan propuestas para su mejora.

En el capítulo 5, se presentan los **aspectos económicos y legales** del proyecto, para ello es imprescindible la explicación del presupuesto. Aunque, tampoco se descuida la explicación del entorno socio-económico actual para el desarrollo de dicho presupuesto, Y en último lugar, se enmarca todo en un marco regulador (esta vez legal, puesto que el técnico ya ha sido desarrollado en el capítulo 2 con los requisitos y las restricciones.

En el capítulo 6, se desarrolla la **planificación del trabajo**. Se exponen tanto la planificación inicial como la final, con sus respectivos cronogramas de tareas, haciendo especial mención a las tareas críticas. La secuenciación y temporización del trabajo ha quedado registrada gracias al uso de Diagramas de Gantt. Se finaliza este capítulo con una comparativa del trabajo estimado frente al realizado.

En el capítulo 7, se presentan las **conclusiones**. Para ello, se da comienzo con los objetivos cumplidos en el proyecto. Una vez desarrollado este punto, se continúa con los problemas encontrados al trabajar en el proyecto. Se cerrará el mismo con las líneas futuras de trabajo que podrían continuarse realizando a largo plazo; para con ello continuar mejorando las soluciones propuestas y llegar así a conseguir una efectividad todavía mayor.

2. PLANTEAMIENTO DEL PROBLEMA

2.1. INTRODUCCIÓN

En este capítulo se expone la información que se tomó como punto de partida para poder desarrollar el proyecto adecuadamente. Se comienza con un repaso del **estado del arte** (explicándose todo lo relevante en cuanto a los videojuegos y también su punto de vista desde la inteligencia artificial). Los demás apartados están enfocados a los **requisitos**, seguidos de las **restricciones**, puesto que sin ellos no se podrían marcar unas metas reales y alcanzables.

2.2. ANÁLISIS DEL ESTADO DEL ARTE

2.2.1. Introducción

A la hora de realizar un videojuego, hay que tener diversos factores en cuenta, por eso es imprescindible saber qué han descubierto anteriores investigadores sobre los patrones humanos: Qué hace que las personas se aburran o diviertan, qué les gusta o qué no, pero sobre todo, qué hace que aprendan de dichos videojuegos (siendo capaces de mejorar cada vez más en los mismos).

Para llegar a una respuesta, al interrogante que deja marcado el párrafo anterior, es necesario analizar los estudios más importantes relacionados con el campo de los videojuegos, la inteligencia artificial y las interacciones humanas con los mismos. Para ello, en el siguiente apartado se realizará una cronología con aquellos estudios más importantes respecto a estos temas.

2.2.2. Evolución de la inteligencia artificial, los videojuegos y el divertimento del usuario

Se procederá a explicar aquellos trabajos de investigación que hayan sido más influyentes para este proyecto, puesto que han sido el punto de partida fundamental para desarrollarlo. Para un mayor orden de los mismos, sólo se indicará su nombre y año de publicación, para más información se recomienda consultar la referencia facilitada de cada artículo.

1. **What makes things fun to learn? (1980)**

   Introducción: Este artículo [3] se centra en las características que debe reunir un juego para que sea divertido, para ello se proporciona una colección de heurísticas que se deberán emplear para conseguir tal objetivo. Es importante recalcar que las pautas dadas sirven para que los juegos sean divertidos, y con ello se pretende que cualquier juego sea divertido, sea del tipo que sea (aunque el autor proporciona

estas pautas para intentar que aquellos juegos que sean educativos sean también divertidos).

Resumen: Este artículo es el más antiguo de todos aquellos que han sido estudiados para este proyecto. Se han encontrado referencias anteriores al mismo; sin embargo, cronológicamente, este ha sido el primer artículo que resulta interesante para el desarrollo del presente proyecto.

Para hacer posible que los videojuegos resulten divertidos se procede a la división de sus posibles características en tres categorías, existiendo en cada una de ellas elementos comunes. Las categorías son: *desafío*, *fantasía* y *curiosidad*.

- **Desafío:** Está regido por el siguiente principio: *Para que un juego sea desafiante, debe proporcionar un objetivo cuyo logro sea incierto.* El cumplimiento de este principio conllevará las siguientes consecuencias:
  - **Objetivo:** El objetivo será uno de los factores más importante, puesto que la simple definición de "juego" implica la existencia de un "objetivo del juego". Sin embargo, no todos los objetivos son igual de buenos; por ello, es necesario encontrar un objetivo apropiado al juego. Se deberán tener en cuenta las siguientes premisas: Un juego simple debería tener un **objetivo obvio** (normalmente cuanto más obvio y fascinante es el objetivo, mejor), la **dificultad** deberá ser la apropiada, los mejores objetivos a menudo son **prácticos o fantásticos** (por ejemplo, alcanzar la luna en un cohete es normalmente más atractivo que pedir usar una habilidad simplemente, como resolver problemas aritméticos), los jugadores deben ser capaces de decir que se están acercando al objetivo (es información de **retroalimentación** muy útil).
  - **Resultado incierto:** Un juego normalmente resulta aburrido si el jugador tiene una certeza sobre si va a ganar o perder. Para que esto no suceda, será recomendable que exista: Una **variable de dificultad en el nivel** (bien puede ser determinada por el programa automáticamente, bien elegida por el jugador o bien determinada por las has habilidades del oponente –como en el ajedrez–), **múltiples objetivos en el nivel** (los buenos juegos suelen tener varios objetivos en un mismo nivel, por lo que es aconsejable definir un objetivo básico y luego un metaobjetivo, como alcanzar el objetivo básico eficientemente, para ello, se puede medir el nivel de puntuación o de rapidez), **información oculta** (deberá ser revelada selectivamente, para provocar la curiosidad y contribuir al desafío), **aleatoriedad** (con ella, se aumenta el interés en muchos tipos de juegos).
  - **Autoestima**: El éxito es un videojuego es como el éxito en cualquier otra actividad que suponga un desafío, puede hacer a la gente sentirse **mejor o peor consigo mismos**. Fracasar en un desafío supone que la persona mine su autoestima, y por tanto, que decrezca el deseo de jugar al videojuego otra vez. Todo esto lleva a buscar el equilibro entre la necesidad de proporcionar una clara

retroalimentación en el rendimiento (realzando el desafío) y la necesidad de no reducir la autoestima hasta el punto de que el desafío llegue a desanimar.

- **Fantasía:** *La utilización de fantasía provoca que los videojuegos sean más interesantes.* Se puede utilizar esta característica tanto para progresar hacia algún objetivo fantástico como para evitar alguna catástrofe fantástica. Se definen dos tipos de fantasía:
  - **Fantasía extrínseca:** La fantasía depende de la habilidad requerida, pero no viceversa. Lo que significa que las acciones desarrolladas no implican un cambio en el mundo fantástico (el mundo en el que está ambientado el videojuego) más allá de si la respuesta es correcta o errónea. También afecta lo rápido que se haya dado la respuesta.
  - **Fantasía intrínseca:** La fantasía depende de la habilidad pero la habilidad también depende de la fantasía. Esto quiere decir que el mundo fantástico en el que está ambientado el juego, cambia según las acciones que se realicen. Se tienen en cuenta más factores que en la fantasía extrínseca, como por ejemplo, si se pedía una respuesta numérica, no sólo indicar que es correcta o incorrecta, sino también lo alta o baja que es en función de lo que se esperaba.

  En general, *las fantasías intrínsecas son más interesantes e instructivas que las fantasías extrínsecas*. Asimismo, la habilidad adquirida con este tipo de fantasía puede ser usada para lograr un objetivo en el mundo real. Como en este tipo de fantasía se encuentran los simuladores, los jugadores son capaces de explotar las analogías existentes entre su conocimiento existente alrededor del mundo de la fantasía y aquello que todavía no le es familiar y está aprendiendo.

  Otro factor a tener en cuenta es el **aspecto emocional** de la fantasía. Sin embargo, a distintos tipos de personas les atrae diferentes tipos de fantasía, produciéndose significativas diferenciaciones entre los tipos de sexo, por ejemplo.
- **Curiosidad:** La curiosidad es la motivación de aprender, independientemente de cualquier búsqueda de objetivo o cumplimiento de la fantasía. Para ello, se debe buscar la complejidad óptima (entendiéndose "complejidad" como lo que un jugador puede entender, y que no se debe confundir con "desafío" que es lo que un jugador puede hacer). Para ahondar en esta categoría, se distinguirán dos tipos de curiosidad:
  - **Curiosidad sensorial:** Mejora la atención dando valor a los cambios en la luz, el sonido y otras estimulaciones del entorno. Se podría definir como un "evento técnico", como un cambio en el ángulo de una cámara, un zoom, u otros cambios similares en la imagen que está siendo presentada. Esta técnica también se usa en los programas de televisión para captar la atención.

Los videojuegos recurren a esta técnica mediante el uso de efectos visuales y de audio. Esto puede darse tanto en: la **decoración** (aunque puede lograr, después de un tiempo de juego, el efecto contrario), **la intensificación de la fantasía** (efectos especiales que no sólo cautivan por sí mismos, sino que también mejoran las asociaciones fantásticas que están evocando), como **recompensa** o como una rep**resentación del sistema** (quizá el mejor uso de los sonidos y gráficos es más efectivo que mostrarle al jugador número o palabras).

- **Curiosidad cognitiva:** Los alumnos están más motivados en aprender más para que su conocimiento cognitivo tengan estructuras mejor formadas. Se pueden usar para ello dos tipos de técnicas: Atrayendo la curiosidad del alumno, y la retroalimentación debería ser sorprendente (una manera "fácil" de hacer esto es usando la aleatoriedad) o siendo educativo, en cuyo caso la retroalimentación proporcionada debería ser constructiva (debería ayudar a los alumnos a mostrarles cómo cambia su conocimiento y llega a ser más completo, consistente o parsimonioso).

Una vez entendidas estas categorías con todas las características que conllevan, se debe pensar **adecuadamente cómo combinarlas**. Casi ningún videojuego incluye todas las características mencionadas anteriormente, y es posible pensar en maneras en las que el juego pueda incorporar más de estas características. Por ejemplo, muchos juegos no tienen opciones para variar la dificultad y probablemente, esos juegos mejorarían añadiendo esta característica.

Por último, se busca emplear lo expuesto en este artículo de investigación en aplicaciones educativas. Para ello, se puede hacer un compendio entre el entretenimiento y la diversión, llegando a conseguir que una situación aburrida sea más interesante (como un niño aprendiendo matemáticas).

Conclusiones: Principalmente, se puede observar que una combinación inteligente de características produce mejores videojuegos. Se pretende que las aplicaciones educativas de las heurísticas mostradas lleven a un aprendizaje más eficiente, más interesante y más divertido.

Finalmente, a pesar de que hayan pasado treinta y dos años desde su publicación, remarcar que este artículo ha servido como referencia a muchos otros, incluyendo el que en este proyecto se desarrolla. Es por esto, que al ser un artículo con bastante peso sobre el proyecto, se ha hecho una mayor insistencia en el mismo (desarrollándose con más detalle su resumen), para que de esta forma fuese posible entender el porqué de algunas de las decisiones tomadas en el diseño de la solución planteada.

2. Heuristics for designing enjoyable user interfaces – lessons from computer games (1981)

Introducción: Este artículo [4] pretende dar respuesta a las siguientes preguntas: ¿Por qué los videojuegos son tan fascinantes? y ¿cómo pueden las características que hacen los videojuegos tan atrayentes ser usadas para hacer otras interfaces de usuario interesantes y entretenidas de usar? Se sugiere, a su vez, cómo algunas de las mismas características pueden hacer los sistemas entretenidos y pueden también convertirlos en fáciles de aprender y usar.

Resumen: Este artículo está estrechamente ligado con el anterior expuesto, es por ello que se irá directamente a los puntos clave del mismo, sin repetir de nuevo definiciones básicas en ambos artículos.

Mediante estudios empíricos se busca determinar qué le gusta a la gente de los videojuegos. Para ello, en uno de los estudios realizados, se eligió un videojuego de la época sobre el que se realizaron 8 versiones del mismo, cada uno con distintas variantes. Las características modificadas estaban relacionadas con la música, la representación de la puntuación, la fantasía de algunos movimientos (imágenes para dar un mayor realismo al juego) y diferentes tipos de retroalimentación. Ochenta y cinco estudiantes jugaron a estas versiones y las valoraron. Se descubrieron significativas diferencias entre chicos y chicas, siendo el resultado principal de este experimento la preferencia de una versión con mayor fantasía por parte de los chicos. De esta forma, la fantasía utilizada puede llegar a ser muy importante, creando entornos motivadores, pero debe ser elegida con cuidado para atraer al público deseado. También se observa que las características más fuertemente correlacionadas con la popularidad del juego son aquellas en las que existe la presencia de un objetivo explícito, marcador de puntuación, efectos de audio y aleatoriedad.

Las implicaciones para el diseño de interfaces de usuario divertidas se basarán, por tanto, en los mismos principios expuestos en el anterior trabajo de T. W. Malone (1980) [3]: desafío, fantasía y curiosidad. Y a través de ellos, se expondrán las heurísticas adecuadas para este caso.

Lo primero que se debe considerar es que muchas de las heurísticas son sólo apropiadas para algunas personas en algunas situaciones y, por tanto, deben ser aplicadas con cuidado.

Las interfaces de usuario, para las que se busca el objetivo planteado, engloban tanto herramientas para la diversión (el mundo de los videojuegos, motivo por el que se intenta investigar y extraer las características válidas en este campo) como herramientas para el sistema (editores de texto, lenguajes de programación, etc.).

En los casos en los que el objetivo externo no sea demasiado motivador, las características que se comentarán a continuación son especialmente útiles para hacer que la actividad sea divertida. Además, la retroalimentación sobre el desempeño permitirá saber cómo de buenos son los logros realizados sobre los objetivos.

Las heurísticas para diseñar de manera divertida son, principalmente, las mismas que se comentaron en el artículo anterior, pero se refuerzan algunas de las ideas

explicadas. Por ejemplo, lo importante que es el fomento de la curiosidad del alumno por medio del deseo de tener estructuras de conocimiento mejor formadas, pero no diciéndole directamente que su conocimiento es incompleto, inconsistente o no parsimonioso (sino queriendo mejorar su conocimiento), es decir, no podemos llegar a mostrarle conocimiento totalmente incomprensible. Otro ejemplo, es la exaltación de la fantasía por medio de **metáforas** (como se realiza en los ordenadores con la creación de un "escritorio" que es simulado en la pantalla. Un último ejemplo, es el empleo (para la curiosidad, de nuevo) de la **aleatoriedad** y el **humor**, que ayudan a crear entornos con complejidades óptimas; sin embargo, en cuanto al humor, se debe evitar que sea inapropiado (no se debe abusar de esa técnica).

Conclusiones: En este artículo se muestran las características más relevantes de los videojuegos (para que sean entretenidos) y cómo hacer otro tipo de interfaces divertidas gracias a estas características. Se presenta una lista con las ideas fundamentales en las que debe basarse, aunque se especifica que no todas las características son útiles en todas las interfaces. Esto lleva a la observación de que es fácil usar estas características inadecuadamente.

Ante todo, según este artículo se debe evitar los gráficos chillones, las fantasías inapropiadas y el humor fuera de lugar (o "enfermo"), pero con creatividad, una fuerte estética y sensibilidad psicológica, las interfaces (ya sean para videojuegos o herramientas) serán más productivas en cuanto a su uso, pero también más interesantes, más divertidas y más satisfactorias.

Es un documento influenciado en gran medida por el anterior y escrito por el mismo autor; no obstante, ha sido muy útil para ahondar acerca de algunas de las heurísticas planteadas.

3. Considering Games as Cognitive Tools In Search of Effective Edutainment (1996)

Introducción: Este artículo [5] trata sobre los beneficios del uso de los videojuegos como herramientas cognitivas, y discute la complejidad de evaluar estos beneficios. El uso de juegos educativos pretende incrementar el interés, la motivación y la retención, además de mejorar elevadamente las habilidades de razonamiento y agilidad mental. Para evaluar la eficacia, muchas variables como las diferencias entre alumnos, los métodos de evaluación y el conocimiento implícito, deben ser consideradas.

Resumen: Para comenzar, se plantea el siguiente interrogante: ¿un juego puede mezclar a la vez una lección de conocimiento y ser una herramienta de evaluación educativa? Como respuesta, se puede observar que los videojuegos que son vendidos como educativos, a menudo parece que tienen una falta obvia de valor cognitivo, mientras que muchos "juguetes" educativos son para nada divertidos ni adictivos. El propósito de este trabajo es revisar las propuestas beneficiosas del uso de los juegos como herramientas cognitivas, y discutir acerca de las complejidades en la evaluación de estos beneficios.

Una forma adecuada de aprender conocimiento mediante los videojuegos es mediante el uso de los simuladores. Pero, ¿qué sucede con los objetivos que los estudiantes no están motivados para alcanzar? Los videojuegos educativos, generalmente, pretenden provocar el interés y la motivación del estudiante, por lo que no debería ser sorprendente que este formato sea usado en un intento de crear entornos de aprendizaje menos tediosos. Muchos profesores e investigadores usan los juegos para suplir o remplazar la formación tradicional pero según se recoge en [5] la efectividad de este tipo de educación no ha sido adecuadamente documentada.

Sin embargo, los investigadores sugieren que jugar incrementa la retención del material de la asignatura y mejora la habilidad de razonamiento y pensamiento de orden superior. Esto se ve facilitado gracias a que los elementos de los juegos a menudo incrementan la motivación incluyendo desafíos y provocando la curiosidad. Gracias a esto, los estudiantes muestran más interés en los juegos (aunque sean educativos), que en las actividades escolares.

El problema surge al intentar medir directamente la efectividad de los videojuegos como herramienta cognitiva. Se deben considerar muchas variables, no sólo las que tienen como finalidad el juego, sino también las que dan importancia al contexto en el que está siendo usado.

Estrechamente relacionado con esto, otra dificultad en la valoración de los juegos de simulación es la tendencia de los investigadores a ignorar las diferencias entre los tipos de personalidad o los estilos cognitivos de los diferentes estudiantes. Además de factores como el estrés o la ansiedad, que pueden afectar a los resultados.

Conclusiones: En este artículo de investigación se muestra que los juegos educativos puede ofrecer una amplia variedad de beneficios. Sin embargo, varios factores deben ser considerados en el diseño de un juego educativo y en el diseño de su evaluación. Por este motivo los investigadores deben ser cuidadosos con sus métodos. Los objetivos educativos del juego deben estar claros y coincidir con la herramienta de evaluación. Para finalizar, las evaluaciones deberían ser consideradas según los tipos individuales de personalidad y estilos de aprendizaje, y sobre todo se debe considerar cuidadosamente cómo el alumnos puede demostrar lo que ha ganado (o aprendido) de la actividad.

4. Procedural Level Design for Platform Games (2006)

Introducción: Este artículo [6] busca explicar que los juegos de plataformas no han desarrollado todavía una buena generación de niveles automática con éxito (tomando como referencia para hacer dicha afirmación, la fecha en la que se publica este trabajo). Por lo que este artículo propone una jerarquía de cuatro capas para representar los niveles de plataformas, con un foco sobre la representación de la repetición, el ritmo y la conectividad. También, se propone una forma de usar este modelo para generar procedimentalmente nuevos niveles (*Procedural Level*).

Resumen: La generación procedimental de niveles ha sido exitosamente implementada en varios géneros de juego (por ejemplo, en *Rogue, Diablo II* o *Civilization*), es por esto que los juegos de plataformas obtendrían beneficios con la generación de niveles, ya que sería muy interesante la opción de poder volver a ser jugados. Sin embargo, en el momento en el que se escribió este artículo, no existían distribuidores comerciales de juegos de plataformas que usasen generación de niveles. Esto tiene una sencilla explicación: la generación de niveles de plataformas es más difícil que la generación de niveles en RPG o juegos de estrategia, puesto que en los juegos de plataformas, cambios muy pequeños pueden cambiar el mundo virtual definido convirtiéndolo en otro **físicamente imposible**.

Usando una rítmica pero variada repetición de los elementos se puede crear una función económica que presente al jugador niveles más largos en los que jugar a través de sólo unos pocos objetos del juego. Por ejemplo, gracias a la repetición y remodelación de unos pocos elementos básicos como tuberías, bloques y plataformas, es posible para los diseñadores construir largos e interesantes niveles en los juegos de ***Mario Bros***.

El modelo expuesto en este trabajo define cuatro tipos de patrones:

- **Patrones básicos (*basic*):** Consiste en un único componente o en una repetición sin variación de ese componente.
- **Patrones complejos (*complex*):** Es una repetición del mismo componente pero con cambios en los ajustes de acuerdo a algunas secuencias del conjunto (como las series de saltos horizontales, incrementando el tamaño de los mismos).
- **Patrones compuestos (*compound*):** Alterna entre patrones básicos hechos con dos tipos diferentes de componentes. Por ejemplo, una serie compuesta por tres saltos seguidos de tres vallas con pinchos para volver a realizar otros tres saltos.
- **Patrones compuestos (*composite*):** Dos componentes colocados demasiado cerca el uno del otro que requieren un tipo de acción diferente o una acción coordinada para seguir adelante, es decir, no requieren para cada uno de los componentes una acción individual. El jugador debe sintetizar su conocimiento en una solución, coordinando distintas capacidades.

Los elementos no lineales que conforman un nivel son necesarios para dar al jugador la habilidad de elegir su propio camino.

Una forma de representar estos niveles es mediante una estructura de celdas con los distintos elementos que nos podemos encontrar en el juego, unidos mediante flechas en la dirección que se debe seguir para pasar de unos a otros. Una estructura de celdas describe cómo las celdas pueden ser conectadas para que afecte al juego (por ejemplo, creando dos caminos que lleguen al mismo lugar). Este modelo representa un nivel como una combinación de patrones y celdas.

La física del modelo calcula con precisión cómo el avatar (el agente que es dirigido en el juego) será movido en un mundo con bordes construido en dos

dimensiones. Por ejemplo, teniendo el conocimiento de puntería de un jugador y sus habilidades respecto al tiempo, se permite al sistema aproximar la dificultad de un salto.

El constructor de patrones se implementó utilizando para ello un algoritmo simple de escalada, intentando llegar a obtener el número de componentes óptimos. Las estructuras mediante celdas no se llegaron a implementar.

Conclusiones: En este trabajo cabe destacar, principalmente, el gran cuidado que se debe tener cuando se trabaja con generación de niveles procedimental, puesto que se pueden generar muy fácilmente niveles imposibles de finalizar, algo completamente inadmisible desde el punto de vista del jugador. También, es relevante que el algoritmo del constructor de patrones funcionó con éxito, siendo capaz de construir y optimizar patrones individuales. El algoritmo de decisión para las estructuras de celdas fue diseñado, pero no implementado.

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

Introducción: Este artículo [7] está enfocado a la experiencia que tiene un jugador con el videojuego *Supero Mario Bros* e investiga la relación entre el diseño de los parámetros en los juegos de plataformas, las características individuales en la manera de jugar y la experiencia del jugador. El diseño de los parámetros investigados relaciona los lugares y tamaños de los saltos (huecos) en el nivel y la existencia de cambios de dirección; y muestra que los componentes de la experiencia de un jugador incluyen la diversión, la frustración y el desafío.

Los resultados muestran que el desafío y la frustración pueden ser predichos con una alta precisión.

Resumen: En este artículo se estudia cómo hacer un generador de niveles para el juego de *Infinite Mario Bros*, para ello se utilizarán:

- **Características controlables** (número de saltos, tamaño medio de los saltos, diversidad en los componentes, etc.) que permitirán hacer una selección de cuántas variantes posibles existen para poder generar un nivel. Para ello, sólo se toman en cuenta las variantes que se consideran más relevantes según los investigadores.
- **Características de la partida** (tiempo que ha invertido el jugador para terminar el nivel, el número de saltos realizados, el número de objetos recogidos, el número de muertes, cuántos enemigos ha eliminado, etc.) que permitirán saber la mayoría de factores necesarios para así responder si un nivel es difícil o no para un usuario y llegar a descubrir qué emociones siente al jugarlo.
- **Experiencia del usuario y protocolos experimentales** que surgen a raíz de las emociones del jugador. Se probaron distintos tipos de algoritmos procedentes del campo de la Inteligencia Artificial para ver qué tipo de técnicas devolvían mejores resultados en cuanto a los efectos de las emociones que experimentase el jugador en la partida. Las emociones estudiadas eran la **diversión**, el **desafío** y la **frustración**; y se pudo

observar que, diferentes algoritmos, daban mejor precisión en unos casos u otros en función del número de características empleadas. Por lo general, los resultados eran aceptables y se podrían emplear posteriormente para mejorar los entornos de desarrollo también con técnicas de Inteligencia Artificial.

Conclusiones: Las características controlables y las que emergen durante el juego, están correlacionadas con una serie de emociones registradas durante el juego. Se ha encontrado un elevado número de correlaciones estadísticamente relevantes, y con ellas se intenta entrenar buenos predictores de las emociones del jugador usando las preferencias de aprendizaje. Estos resultados serán mejorados al intentar generar automáticamente entornos para este juego usando Inteligencia Artificial (teniendo presente los modelos de experiencia del jugador como una función de *fitness* o función de mejora).

Los resultados expuestos en este artículo de investigación han influido directamente en el proyecto aquí presentado, puesto que dicho artículo está fervientemente recomendado para la generación de niveles (*LGT*) en el juego de *Infinite Mario Bros* utilizado en al *Mario AI Championship* [1] sobre el que también se basa este proyecto.

6. Analyzing the Expressive Range of a Level Generator (2010)

Introducción: Este artículo [8] explora un método para analizar el rango de un generador procedural de niveles (*PCG*) y aplica este método a *Launchpad* (un generador de niveles para juegos de plataformas en 2D). A su vez, se examina la variedad de niveles generados y el impacto de los cambios en los parámetros de entrada. Debido a la popularidad de *PCG*, es importante ser capaz de evaluar justamente y comparar diferentes técnicas de generación de niveles dentro de dominios similares.

Resumen: Este artículo busca hacer un enfoque analítico sobre el rango expresivo de un generador procedimental de niveles. Para ello, se explica que es necesario seguir los pasos adecuados, consistiendo estos en determinar las métricas adecuadas (capaces de medir y comparar resultados), generar contenido, visualizar el espacio generado y analizar el impacto de los parámetros.

Este artículo extrae conclusiones sobre generación de niveles a partir del juego *Launchpad*. Pero quizá lo más importante es cómo evaluar el rango de expresividad de un generador de niveles; existen dos formas: considerándolos aplicaciones inherentes en el algoritmo genético o analizando el espacio de niveles que pueda ser creado. Como objetivo, la combinación de las distintas características que se pueden extraer del juego deben dar niveles con contenidos diversos y variados entre sí.

Las métricas que se vayan empleando permitirán comprobar los niveles producidos y describir el rango expresivo del generador de niveles, generando un número de niveles y una ordenación de los mismos mediante el uso de distintos parámetros.

Aunque parezca obvio, es necesaria la aportación que puedan hacer los críticos sobre el modelo y los niveles de los mismos para una correcta evaluación y poder tomar decisiones más o menos acertadas.

Conclusiones: En primer lugar, es posible para un sistema ser expresivo sin necesidad de ser creativo. Por ejemplo, en el caso de *Launchpad*, éste no es capaz de evaluar la calidad de su salida utilizando algún método que no esté predefinido en sus funciones de *fitness*. Sin embargo, *Launchpad* es capaz de crear una gran variedad de niveles según las métricas humanas definidas.

Sin embargo, queda bastante trabajo por realizar, como por ejemplo determinar mejores métricas para la comparación de niveles. También sería necesario algo similar al modelo de comportamiento del jugador, para recoger los diferentes estilos que podrían ser interesantes para abordar el problema. La dificultad es una medida obvia, pero otras medidas también deben ser percibidas en el desafío al que se enfrenta el jugador, haciendo del mismo un reto más sencillo o complejo (las posiciones de la cámara, por ejemplo).

Se debe tener en cuenta que este artículo se ha centrado exclusivamente en niveles de juego que no son en línea (con un diseño mínimo de entradas), pero si estuviésemos ante un sistema en línea estaría menos claro qué debería ser comparado, puesto que el contenido cambia en tiempo real y, obviamente, esto es una dificultad añadida para el diseño del modelo.

Con este artículo se ha buscado incentivar la discusión de las técnicas que son apropiadas para evaluar un generador de niveles, **no sólo por la calidad de los niveles individuales**, sino también por todo el rango de niveles que se pueden crear.

Este último punto se ha tenido muy en cuenta durante todo el desarrollo de este proyecto, puesto que se ha buscado que los niveles generados tengan diversidad genética y no evaluarlos de forma individual únicamente.

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

Introducción: Este artículo [9] muestra que la personalización de niveles puede ser generada automáticamente para los juegos de plataformas, donde los modelos derivan de la predicción hecha en base a la experiencia del jugador, que cuenta con características de diseño del nivel y estilos de juego. Estos modelos son construidos usados preferencias de aprendizaje, basadas en cuestionarios administrados a los jugadores después de jugar diferentes niveles.

De esta forma, las contribuciones de este artículo son: (1) una mayor precisión de los modelos basada en una gran cantidad colección de datos, (2) un mecanismo para niveles adaptativos en base a parámetros que les permita tener niveles adecuados a su estilo de juego y (3) una evaluación de este mecanismo de adaptación usando algoritmos y jugadores humanos. Los resultados indican que el mecanismo de adaptación optimiza efectivamente el diseño de parámetros del nivel para los jugadores particulares.

Resumen: La generación procedimental de contenido (*PCG*) es una importante técnica para el desarrollo de los videojuegos. Es por esto, que se busca emplear esta técnica en la resolución de *Infinite Mario Bros*.

Para ello, inicialmente son extraídos tres tipos de datos de un total de 327 jugadores: las características controlables del juego, las características de la partida y la experiencia del usuario.

Para comenzar el desarrollo del modelo, se prueba a usar perceptrones simples. Después de seleccionar conjuntos adecuados de características que maximizasen la precisión en la predicción de las preferencias de los jugadores, se mejora la topología y los pesos, utilizándose perceptrones multicapa para coincidir con las preferencias de los jugadores.

Un conjunto de 58 características son extraídas durante el juego (tiempo, acciones del jugador, causa de la muerte, etc.). A pesar de ello, se busca que el modelo fuese dependiente del menor número de características posible, no sólo para que fuese más sencillo de analizar, sino para hacerlo más útil e incorporarlo a futuros juegos. Es por este motivo que al realizar el mecanismo de adaptación para poder utilizar el juego en línea, se utilizan sólo cuatro características que se emplearán para la búsqueda del espacio.

En cuanto a los experimentos, se utilizan dos tipos de controladores con el fin de probar la adaptabilidad del generador de niveles, siendo el resultado bastante adecuado en ambos casos (los porcentajes de éxito conseguidos en cuanto al resultado de "diversión" –que es el objetivo final por el que se busca hallar las preferencias del usuario– oscilan entre el 68.71% y el 93.88%, por lo que aunque los resultados son mejorables, llegan a ser bastante exitosos.

Conclusiones: Se introduce un mecanismo de adaptación para juego en línea que puede ser usado para optimizar efectivamente la experiencia del usuario. Se generan, usando este método, niveles a medida para los usuarios específicos. Los experimentos realizados dan resultados exitosos, mejorando aceptablemente la precisión y la adaptación del mecanismo (se muestran prometedores resultados).

Por último, al igual que sucedió con otro documento, los resultados expuestos en este artículo investigación han influido destacadamente en el proyecto desarrollado, ya que dicho artículo está recomendado para la generación de niveles (*LGT*) en el juego de *Infinite Mario Bros* utilizado en al *Mario AI Championship* [1] sobre el que se basa este proyecto.

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

Introducción: Este artículo [10] intenta clarificar el concepto de generación procedimental de contenido (*PCG*), discutiendo hacer de algunas propiedades de *PCG* que se creían a veces necesarias pero que actualmente no lo son.

Se presentan dos versiones de un sistema de generación de contenido para *Infinite Mario Bros*. Y se explica mediante esas dos versiones cuál es un ejemplo de *PCG* y cuál no, a pesar de ser muy parecidas. Este trabajo puede servir a los

investigadores y desarrolladores a no equivocarse en el uso del *PCG* y desarrollar técnicas más eficaces.

Resumen: Generación procedural de contenido (*PCG*) en los juegos se refiere a la creación de contenido automático en el juego usando para ello algoritmos. Sin embargo, esta definición está lejos de ser precisa. *PCG* es un concepto con límites difusos y poco claros. Por dicho motivo, en este trabajo se presentan unos pocos ejemplos que sirven para concretar qué es *PCG* o qué puede serlo.

- Algunos juegos tienen editores de contenido para que los jugadores puedan crear su propio contenido, un ejemplo de esto es *Little Big Planet* (Sony 2009). Se podría afirmar que el contenido generado por un jugador no es contenido generado procedimentalmente porque ese contenido ha sido creado por jugadores humanos. No obstante, hay algoritmos de *PCG* que tienen como característica el tener en cuenta como entrada ciertos datos introducidos por humanos.

  La diferencia entre la iniciativa mixta de *PCG* y un **editor de niveles** parece residir en cómo se edita el contenido. En un editor de niveles el usuario crea y sabe cómo el contenido cambia. Pero, en un caso de *PCG* no es sencillo para un humano predecir los **cambios exactos** que vendrán como resultado de una entrada particular.

- Muchos juegos tienen como temática realizar construcciones; un ejemplo de ello es *Sim City* (Maxis 1990). Si la entrada realizada por una persona al generador de contenido es una parte del juego, y el jugador directamente tiene la intención de crear contenido en el juego, no es generación procedural de contenido (*PCG*). *PCG* es algo distinto a los juegos de estrategia. Por supuesto, esto no quiere decir que no exista *PCG* en los juegos de estrategia, pero para que podamos denominarlo *PCG* el jugador no puede predecir con detalle cómo responderá ese contenido, cómo actuará o estará definido exactamente.

- Los algoritmos de *PCG* podrían utilizar parámetros que afectasen a constantes en la generación del contenido o propiedades estadísticas de un proceso estocástico o ambas opciones. Siempre sin llegar a confundir lo que es información determinista de lo que es el empleo de un *PCG*.

De las dos versiones que se crearon para explicar qué es y qué no es *PCG*, se puede decir que la versión fuera de línea (*offline*) es un ejemplo de *PCG*, mientras que la versión en línea (*online*) es probablemente una herramienta para la creación de contenido por parte del jugador.

La versión fuera de línea no sólo basa su adaptación en métricas de alto nivel, sino que también utiliza aleatoriedad. En teoría, sería posible predecir y dar forma al siguiente nivel al realizar las acciones del jugador de una determinada manera en el nivel actual, pero aun así sería realmente dudoso que la capacidad del ser humano pueda definir exactamente cómo será el próximo nivel.

La versión que está en línea es directa y las acciones realizadas tienen consecuencias en el momento, al instante. Como los resultados de una acción son

visibles y explotables en el mismo nivel en el que se encuentre el jugador, es posible intencionadamente modificar el nivel actual con diferentes objetivos.

Para finalizar, *PCG* se podría redefinir como una creación algorítmica del contenido del juego con límites o entradas del usuario indirectas. Cabe resaltar que en esta definición no aparecen las palabras "aleatoriedad" ni "adaptabilidad", porque los métodos de *PCG* podrían contener ambos, alguno o ninguno.

Conclusiones: En este artículo se busca clarificar qué es la generación procedimental de contenido y qué no lo es. Para ello, se han empleado dos versiones de un generador de niveles que estaban diseñados para generar contenido. Una de las versiones no se considera que pertenezca a un *PCG*.

Aunque en el artículo se argumentan los motivos del por qué, es posible que haya personas que discrepen con este planteamiento, por eso se les ruega que estudien dichas explicaciones, y así poder decidir si esa versión que no pertenece a un *PCG*, se aproxima más a un generador adaptativo y directo de niveles o, simplemente, a un generador en línea determinista. Y estas definiciones (con las características que conllevan) están muy alejadas de poder ser un sinónimo de *PCG*.

Se han utilizado las ideas expuestas en este trabajo para tomar algunas decisiones de diseño fundamentales en el proyecto, por ejemplo, a la hora de elegir si los enemigos que aparecen en los niveles deben emerger siguiendo unas estadísticas o aleatoriamente, ya que el objetivo es crear un tipo de *PCG*.

9. The 2010 Mario AI Championship: Level Generation Track (Diciembre 2011)

Introducción: Este artículo [11] ofrece información útil acerca de una reconocida competición de generación de niveles. La *Level Generation Competition* es parte de la *IEEE Computational Intelligence Society*, patrocinada por *2010 Mario AI Championship*, en la que se muestra el conocimiento siendo la primera competición de generación de contenido. Los competidores participan enviando generadores de niveles (*software* que genera nuevos niveles de una versión de *Super Mario Bros* a medida para los jugadores individuales). Este trabajo presenta las reglas de la competición, el *software* utilizado, el procedimiento para la puntuación, los generadores de niveles participantes, y los resultados de la competición. También se discute acerca de qué se puede aprender de esta competición, sobre la organización de las competiciones de generación procedural de contenido y sobre la generación automática de niveles para juegos de plataformas. Este trabajo ha sido realizado tanto por los organizadores de la competición como por los competidores.

Resumen: Este artículo explica en gran profundidad los temas comentados en la introducción; no obstante, para evitar volver a explicar aquellas definiciones y datos que ya han sido proporcionadas en los artículos anteriores, se derivará a explicar lo más interesante de este trabajo de cara al proyecto (además de su posterior presentación a la competición).

Seis fueron los trabajos admitidos en la competición y que, por tanto, participaron generando diferentes niveles para conseguir desarrollar los niveles más divertidos (que es la medida utilizada en esta competición, que los niveles sean entretenidos y capten la atención del jugador). Todos los participantes utilizaron, obviamente, técnicas de *AI*: búsqueda con algoritmos de optimización, análisis discriminante lineal (*LDA*), algoritmos que tienen constantes de satisfacción… Los mejores resultados, con diferencia respecto a los demás, fueron ofrecidos por un generador múltiple probabilístico (*ProMP*). Para ello, se siguen seis pasos en los que se explica cómo crear un nivel básico y luego iterarlo varias veces, para en cada pasada colocar un nuevo tipo de componente. Se procede a la creación del nivel en este orden: suelo, colinas, tuberías, enemigos, bloques y monedas.

En esta competición se busca poder generar contenido para los niveles y atraer a los jugadores a estos nuevos entornos que se alejan de la monotonía, dándole un nuevo atractivo a cualquier juego. Aun así, es realmente complejo establecer correlaciones entre las preferencias de los jugadores (principalmente, con sus emociones) y características del nivel con su calidad. Por eso, es necesario un análisis al detalle, para poder intentar extraer un conjunto de métricas con las que definir qué juegos son más entretenidos y así diseñar mejor diferentes aspectos en los niveles.

Conclusiones: En este artículo se busca aclarar el objetivo de la competición *Mario AI Championship: Level Generation Track*. Para ello, se explica con detalle qué es el *PCG*, en qué se basa la idea de competición (concretamente el subapartado generación de niveles) y se explican las reglas, puntuación, interfaz…

Al igual que en los artículos que han hecho años anteriores, en este se relata quiénes compitieron dicho año (2010) y qué técnicas usaron para ello. Posteriormente, en un apartado dedicado a la evaluación, se explica qué resultados se obtuvieron y quién ganó dicha competición.

Este es otro de los artículos más interesantes para el proyecto, puesto que uno de los objetivos es poder realizar, a posteriori de este proyecto, un artículo de investigación basado en los conocimientos extraídos de *Mario AI*.

10. The Mario AI Benchmark and Competitions (Marzo 2012)

Introducción: Este artículo [12] describe el banco de pruebas *Mario AI*, un juego basado en la realización de pruebas para el refuerzo del aprendizaje de algoritmos y técnicas de *AI* desarrolladas por los autores.

Durante los últimos dos años, este banco de pruebas ha sido usado en varias competiciones asociadas con conferencias internacionales, colaborando investigadores y estudiantes de todo el mundo en diversas soluciones para intentar superar lo conseguido anteriormente. Este artículo resume esas contribuciones, dando una visión general del estado del arte en el juego de *Mario AI*, y ofreciendo crónicas de desarrollo del banco de pruebas. Este trabajo pretende ser el punto definitivo de partida como referencia para usar el banco de pruebas, tanto para investigación como para enseñanza.

Resumen: Se presenta el banco de pruebas *Mario AI*, que sirve para satisfacer numerosos criterios (aunque investigación y enseñanza sean los principales). Como se explicó en los anteriores trabajos, *Mario AI* está basado en *Infinite Mario Bros*, que a su vez es una versión de dominio público del clásico juego de plataformas *Super Mario Bros*.

Muchos juegos comerciales de plataformas incorporan poca o nula *AI*. La principal razón de este hecho es, probablemente, que una gran cantidad de juegos de plataformas no tienen adversarios; un jugador individual puede controlar un personaje individual que realice una secuencia de niveles, cuyo éxito dependa únicamente de la habilidad del jugador. Incluso cuando aparecen enemigos, en la mayoría de juegos de plataformas se mueven según unos patrones definidos o simplemente imitan comportamientos.

Una de las ventajas de utilizar *Mario AI* como banco de pruebas, es que se puede obtener información a través de la interfaz sobre, por ejemplo, la posición exacta de los enemigos o sobre el estado de *Mario* (pequeño, grande o de fuego).

Aunque existen distintas competiciones utilizando este banco de pruebas, este documento se centra en aquellas en las que se busca la **generación de niveles**. Es por esto que este artículo explica el código necesario para realizar algunas llamadas básicas al sistema y generar niveles sencillos.

Las tres competiciones en este campo más destacadas fueron las que tuvieron lugar en 2009, 2010 y 2011 (aunque de este último año todavía no haya sido publicado un trabajo que explique todas las conclusiones obtenidas). Sin embargo, de los dos años anteriores se pueden extraer un par de conclusiones:

- En **2009**, los algoritmos desarrollados eran aún sencillos y estaban basados en algoritmos como el A*.
- En **2010**, se desarrollan otro tipo de niveles algo más complejos que los anteriores que obligan al agente que juegue en ellos a realizar niveles más complejos (por ejemplo, dos caminos para escoger dentro de un mismo nivel y tener que retroceder porque uno de ellos no tiene salida). Lo que lleva a que el agente realice acciones de un relativo alto nivel. También está más presente evitar a los enemigos o utilizar correctamente los saltos que puede realizar *Mario* para evitar huecos y llegar a zonas elevadas.

Por último, aunque con este banco de pruebas se presenta el desarrollo de juegos de plataformas mediante Inteligencia Artificial, también puede ser usado para investigación fundamental y enseñanza, como ya se puntualizó.

Conclusiones: En este artículo se describe el banco de pruebas de *Mario AI*, y las distintas competiciones en las que se ha empleado como base en 2009 y 2010.

El trabajo se ha centrado en describir la tecnología detrás del banco de pruebas, la organización de las competiciones y las razones detrás de ambos. A su vez, se extraen unas conclusiones acerca del problema de la Inteligencia Artificial en los juegos de plataformas.

Existe gran cantidad de artículos de investigación que no han sido expuestos en este estado del arte pero cuya lectura sí es recomendable. Tales como:

- *Super Mario Evolution* (2009) [13]
- *The 2009 Mario AI Competi*tion (2010) [14]
- *A Generic Approach to Challenge Modeling for the Procedural Creation of Video Game Levels* (Septiembre 2011) [15]
- *Dynamic Game Difficulty Scaling Using Adaptive Behavior-Based AI* (Diciembre 2011) [16]

Estos tres artículos, que aquí se nombran, han tenido un importante peso sobre el proyecto, aunque en menor medida que los diez descritos anteriormente. Sin embargo, todos ellos han servido para dar las directrices adecuadas para este proyecto y poder desarrollar un generador de niveles mejorando las ideas ya existentes sobre este campo. Además de los artículos mencionados, existen bastantes más referencias investigadas para este estado del arte; no obstante, no han tenido tanto peso en el desarrollo del proyecto. Aun así, todos ellos se encuentran en las referencias, puesto que ha sido necesario tener en cuenta esa información.

Por último, como se puede observar, los artículos expuestos van complementándose entre sí cronológicamente, y para el problema que se pretende abordar en este proyecto, se emplean los conocimientos adquiridos de estos trabajos, teniendo en cuenta las soluciones aportadas (especialmente, las de los dos últimos artículos desarrollados [11] [12] que ofrecen distintas soluciones al problema planteado).

2.2.3. Conclusiones: Comparativa y resumen

Para alcanzar un mayor grado de compresión se va a proceder a realizar una **comparativa y resumen** de las soluciones aportadas:

Los artículos expuestos con el paso del tiempo han ido centrándose más en *AI*, aunque el objetivo sigue siendo la diversión y el entretenimiento del jugador. Se plantea una interesante relación entre los juegos educativos y la diversión, que termina convirtiéndose en la búsqueda mediante la Inteligencia Artificial a que el jugador experimente (y por tanto, aprenda) a desarrollar mejor sus capacidades, nunca dejando la autoestima y entretenimiento del jugador para que este siga participando en el juego.

Puesto que las únicas soluciones aportadas como tal al problema son aquellas que han participado en la competición, se observa que la aplicación de **algoritmos de búsqueda** como se explica en los artículos realizados sobre la competición de 2009 [12] [14], no es una solución recomendable, ya que se sugiere enfocar al problema mediante opciones que fortalezcan **el desafío, la fantasía y la curiosidad** [3]; para ello, una buena alternativa es la utilización de las emociones del jugador y la aleatoriedad. Basándose en estos principios se exponen distintos algoritmos para resolver el problema en 2010 [11] [12], que ofrece mejores expectativas y resultados. Se estudian las distintas técnicas empleadas y se observa que una buena forma de enfrentar este problema es mediante **algoritmos genéticos y evolutivos**, puesto que no es recomendable el uso de una técnica determinista. Se propone en este proyecto una solución basada en *AI* basada en los criterios de las técnicas especificadas.

2.3. REQUISITOS

Siguiendo las enseñanzas recibidas a lo largo de la carrera, se ha decidido utilizar la plantilla *Volere* [17] para la especificación de requisitos. En los siguientes sub-apartados se desarrollarán **los más importantes**.

2.3.1. Restricciones globales

Las **restricciones globales** *"afectan a todo el producto y son determinadas por el usuario y los que administran el proyecto/producto"* [17]. Existen diferentes tipos de restricciones en función del proyecto en el que se trabaje. Los tipos a los que pueden pertenecer las restricciones son los siguientes:

- Restricciones de Solución.
- Ambiente de Implementación del Sistema Actual.
- Aplicaciones de Colaboración o Sociedad.
- Software Disponible.
- Ambiente Anticipado del Lugar de Trabajo.
- Restricciones Cronológicas.
- Restricciones Presupuestarias.

# Requisito: 1	**Tipo de requisito: Restricción global - Cronológica**	
Descripción: Se finalizará el proyecto, como máximo, a día 15 de junio de 2012.		
Razón: Entregar el proyecto puntualmente para su valoración.		
Creador: Beatriz Puerta Hoyas		
Criterio de aceptación: El plazo de entrega de la documentación finaliza el día 15 de junio del 2012 para poder optar a presentarlo en la primera sesión.		
Satisfacción del cliente: 5	**Insatisfacción del cliente:** 5	**Conflictos**: -
Prioridad: 5		
Materiales de soporte: -		
Historia: Creado el 23 de marzo de 2012		

# Requisito: 2	Tipo de requisito: Restricción global – Software Disponible	
Descripción: Se emplearán herramientas gratuitas para la realización del proyecto, o versiones libres / de prueba		
Razón: Abaratar al máximo el coste del proyecto		
Creador: Beatriz Puerta Hoyas		
Criterio de aceptación: Las aplicaciones ofimáticas y de desarrollo de código empleadas para la realización del proyecto serán gratuitas o bien en distribuidas mediante canales especiales de *software* (como Microsoft™ DreamSpark) que permitan ahorrar el máximo posible el coste en licencias de *software*.		
Satisfacción del cliente: 5	**Insatisfacción del cliente:** 5	**Conflictos**: -
Prioridad: 5		
Materiales de soporte: -		
Historia: Creado el 23 de marzo de 2012		

2.3.2. Requisitos funcionales y de datos

Los **requisitos funcionales** *"describen una acción que debe realizar el producto y son independientes de cualquier tecnología usada en el producto"* [17]. Y es necesario que cada requisito de este tipo tenga un criterio de aceptación o una prueba.

# Requisito: 3	Tipo de requisito: Requisito funcional	
Descripción: La aplicación debe permitir generar niveles para una variante del juego *Infinite Mario*.		
Razón: El objetivo del proyecto es desarrollar una herramienta que permita generar niveles de *Mario AI* mediante algoritmos evolutivos.		
Creador: Beatriz Puerta Hoyas		
Criterio de aceptación: Se debe proveer al menos de un método que genere niveles empleando algoritmos evolutivos, como los algoritmos genéticos.		
Satisfacción del cliente: 5	**Insatisfacción del cliente:** 5	**Conflictos**: -
Prioridad: 5		
Materiales de soporte: -		
Historia: Creado el 23 de marzo de 2012		

# Requisito: 4	**Tipo de requisito: Requisito funcional**	
Descripción: La aplicación debe permitir guardar los niveles generados		
Razón: En caso de necesitar acceder a los niveles generados posteriormente será necesario poder guardarlos en fichero.		
Creador: Beatriz Puerta Hoyas		
Criterio de aceptación: Se proporcionará funcionalidad para salvar el nivel tras generarlo o bien desde dentro del juego, mientras se prueba un nivel.		
Satisfacción del cliente: 5	**Insatisfacción del cliente:** 5	**Conflictos**: -
Prioridad: 5		
Materiales de soporte: -		
Historia: Creado el 23 de marzo de 2012		

# Requisito: 5	**Tipo de requisito: Requisito funcional**	
Descripción: La aplicación debe permitir cargar los niveles generados		
Razón: En caso de necesitar acceder a los niveles guardados previamente será necesario poder cargarlos desde fichero.		
Creador: Beatriz Puerta Hoyas		
Criterio de aceptación: Se proporcionará funcionalidad para cargar el nivel en el momento de ejecutar la aplicación o bien desde dentro del juego, mientras se prueba un nivel.		
Satisfacción del cliente: 5	**Insatisfacción del cliente:** 5	**Conflictos**: -
Prioridad: 5		
Materiales de soporte: -		
Historia: Creado el 23 de marzo de 2012		

En cuanto a los **requisitos de datos**, se refieren a la información empleada por el sistema. Para más información sobre este tipo de requisitos, se aconseja recurrir al punto 3.5.2. de

este proyecto (en dicho apartado se relaciona el diagrama de clases en *UML* con la codificación final de la solución).

2.3.3. Requisitos no funcionales

Las restricciones globales *"describen las propiedades o características que el producto debe tener (cualidades)"* [17]. Estos requisitos se agrupan en las siguientes categorías:

- Apariencia.
- Usabilidad.
- Rendimiento.
- Operación y entorno.
- Mantenimiento y soporte.
- Seguridad.
- Políticas y culturales.
- Legales.

# Requisito: 6	**Tipo de requisito: Requisito no funcional - Usabilidad**	
Descripción: La aplicación debe ser sencilla de utilizar		
Razón: Debe estar orientada a un público que tenga unos conocimientos medios-básicos de informática		
Creador: Beatriz Puerta Hoyas		
Criterio de aceptación: La aplicación debe presentar información de ayuda en caso de introducir parámetros incorrectos en su invocación. Igualmente sería recomendable utilizar teclas de control estándar (como los cursores, o asdw).		
Satisfacción del cliente: 5	**Insatisfacción del cliente:** 5	**Conflictos**: -
Prioridad: 5		
Materiales de soporte: -		
Historia: Creado el 23 de marzo de 2012		

# Requisito: 7	Tipo de requisito: Requisito no funcional - Operacional	
Descripción: La aplicación debe ser distribuida en un archivo .jar autocontenido		
Razón: De esta forma no hay problemas con las rutas a bibliotecas externas a la hora de invocar la aplicación en Java™, siendo mucho más sencilla su distribución y uso.		
Creador: Beatriz Puerta Hoyas		
Criterio de aceptación: Una vez generado el código ejecutable, deberá ser empaquetado en un .jar que incluya recursos y librerías externas empleadas en el proyecto.		
Satisfacción del cliente: 5	**Insatisfacción del cliente:** 5	**Conflictos**: -
Prioridad: 5		
Materiales de soporte: -		
Historia: Creado el 23 de marzo de 2012		

# Requisito: 8	Tipo de requisito: Requisito no funcional - Operacional	
Descripción: La aplicación debe ser ejecutada sin necesidad de su instalación		
Razón: De esta forma es más cómoda su distribución y uso, a la vez que mejora la portabilidad.		
Creador: Beatriz Puerta Hoyas		
Criterio de aceptación: Tras generar el .jar se comprobará que funcione correctamente sin necesidad de tener directorios con bibliotecas y/o recursos.		
Satisfacción del cliente: 5	**Insatisfacción del cliente:** 5	**Conflictos**: -
Prioridad: 5		
Materiales de soporte: -		
Historia: Creado el 23 de marzo de 2012		

2.4. CONCLUSIONES

En este capítulo se ha desarrollado el **planteamiento del problema** que, a grandes rasgos, se puede dividir en dos puntos: Una parte de **investigación** (en donde se plantea el estado del arte) y otra de mayor **exhaustividad técnica** en cuanto al procedimiento a seguir para desarrollar un buen proyecto (en donde se explican las restricciones y los requisitos fundamentales para poder tener en cuenta todos aquellos factores que son relevantes a la hora de elaborar una idea).

En el **estado del arte**, desarrollan en profundidad los diez trabajos que han sido más relevantes para el proyecto, comentándose también algunos otros cuya importancia en el mismo ha sido destacable. Sin embargo, existen muchos más artículos apreciables para este proyecto, pero que no han influido de la misma forma que aquellos que se han nombrado a lo largo de este apartado. Para una mayor comprensión de la evolución en esta investigación, se ha seguido un orden cronológico para no repetir aquellos conceptos que ya se habían explicado en artículos anteriores.

Por otro lado, en el apartado de **requisitos** se observan, a su vez, varios apartados que son necesarios para establecer una coherencia. Estos apartados son: **restricciones globales**, requisitos **funcionales** y de datos, y requisitos **no funcionales**. Dentro de cada uno de ellos se exponen aquellos requisitos (o restricciones) que son necesarios apreciar para el correcto desarrollo del diseño del proyecto.

En el próximo capítulo (el tercero), se avanzará un paso más en el proyecto, desarrollándose el **diseño de la solución técnica**, donde se podrán observar todo el proceso de diseño (ideas básicas de la solución inicial, algoritmos para el desarrollo de la misma, problemas de esa solución, alternativas, ventajas e inconvenientes, elección y mejora de una solución final…).

3. DISEÑO DE LA SOLUCIÓN TÉCNICA

3.1. INTRODUCCIÓN

Para abordar el problema planteado, se empezará desarrollando el diseño de la solución inicial, en el que se podrá observar que se partió de un generador que proporcionaba niveles **aleatorios** en función de unos **parámetros** específicos (como la dificultad), para pasar al diseño de un **algoritmo genético**. Sin embargo, como se pudo comprobar, surgieron una serie de problemas que dicho algoritmo genético no puede resolver (determinación de algunas condiciones de imposibilidad).

Por el motivo anterior, se elaboró un generador **coevolutivo** que, basándose en un agente inteligente de *Mario* existente (que tuvo que ser adaptado) y en un enfoque de generación genético de niveles empleando las evaluaciones de dicho agente como *fitness*, pudiese mejorar aun más los resultados obtenidos con el algoritmo genético, puesto que al "jugar" los niveles generados es capaz de solventar los problemas encontrados en el enfoque genético.

Con la **combinación** de esas tres formas de diseño se plantea la solución final.

Por último, se propuso un **cuarto diseño** (enfoque basado en el Aprendizaje Automático) que no llegó a ser implementado debido a los inconvenientes que ofrecía respecto a uno de los objetivos buscados (generación aleatoria y no determinista de los niveles).

3.2. DISEÑO DE LA SOLUCIÓN INICIAL

3.2.1. Ideas básicas: ¿Cómo evolucionará el diseño?

En primer lugar, es necesario recordar cuál es el **objetivo** que se pretende resolver: Se busca generar niveles para la plataforma *Mario AI* y que dichos niveles cumplan con una serie de objetivos: ser **divertidos** (cualidad abstracta difícil de medir) y **adaptarse al jugador** (en cuanto a la dificultad de los mismos).

Con el objetivo presente, ya se pueden comentar las ideas básicas que son necesarias para enfrentarse a este problema:

- La primera idea indispensable para hacer frente este problema es el de investigar **cómo generar niveles** en el juego.
- Ligado al punto anterior, se necesita de un sistema para **guardar y cargar niveles** (¿de qué sirve tener niveles que resuelvan el problema si no los almacenamos para poder jugar con ellos o evaluarlos posteriormente?)
- Primera solución propuesta: A partir de la idea básica expuesta en la primera idea de generación aleatoria de niveles, se desarrolla ***la primera opción de diseño: el generador de niveles en base a la aleatoriedad con parámetros.***
- Sin embargo, un buen generador de niveles no puede basarse sólo en ese concepto de aleatoriedad, puesto que las reglas tomadas para la generación de contenido en base a la dificultad deseada son demasiado estrictas y se puede acabar generando

niveles muy similares (problema que se observó tras realizar muchas pruebas de generación de aleatorios con distintas dificultades). Así pues, ante la pregunta ¿cómo generar conjuntos de niveles que tengan un mismo tipo de dificultad y que no sean igual o demasiado parecidos entre sí? Surge ***la segunda opción de diseño: el generador de niveles en base a un algoritmo genético.***

- Como aun así, siguen existiendo algunos problemas que se comentarán más adelante, se propone ***la tercera opción de diseño: el generador de niveles en base a un algoritmo coevolutivo***. Para ello, se empleará el código del agente de *Mario* diseñado por Héctor Valero en su Proyecto Fin de Carrera [2]. Gracias a la combinación de un agente cuyo objetivo es poder encontrar a solución a todos los niveles que se le planteen, se podrá comprobar cómo de difíciles son los niveles del generador encargado de realizarlos, descartar los niveles imposibles gracias al agente inteligente (o ver si sólo eran niveles en apariencia imposible) y evitar los "*trucos*" (por ejemplo, que existan niveles en apariencia muy difíciles pero gracias a algún objeto o tipo de enemigo, se pueda resolver el nivel con relativa facilidad). Respecto a ésta última idea podría pensarse en un nivel totalmente llano y a la misma altura lleno de enemigos tipo "*spikey*" (tortuga con pinchos), imposibles de eliminar saltando encima de ellos o lanzándoles bolas de fuego. El nivel sería imposible de superar al no poder saltar sobre todos los enemigos a la vez (*fitness* muy elevado según la solución 2). Ahora pensemos en que al principio del todo, encabezando los "*spikeys*" se encuentra una tortuga roja. Si saltamos sobre ella, luego sobre su concha proyectándola hacia la hilera de "*spikeys*" y corremos detrás, no sólo conseguiremos superar el nivel sino que además obtendremos una altísima puntuación (muchos enemigos eliminados y poco tiempo empleado en completar el nivel). Así pues y como idea importante, esta última solución permite evaluar la dificultad respecto a la jugabilidad del nivel y no en base a la configuración y distribución de elementos.

3.2.2. Elementos que puede contener un nivel

Antes de proceder a describir los distintos algoritmos propuestos como solución, vamos a comentar los distintos elementos que puede contener un nivel. Podemos clasificar los elementos que componen un nivel en dos grupos: **tipos de fragmento** que componen el nivel a modo estructural y **enemigos** que pueden aparecer en dichos fragmentos. La longitud de los fragmentos puede tener un tamaño arbitrario a priori. También es posible definir otros **elementos de "decoración"** que pueden ayudar a *Mario* a la hora de completar el nivel facilitando su tarea. Estos elementos son los **bloques de ladrillos** (se puede saltar sobre ellos para evitar peligros), los **bloques con *power-ups*** (si se golpean desde abajo sueltan elementos que otorgan habilidades especiales a *Mario*) y finalmente las **monedas**, que son acumuladas y al alcanzar el valor de 100 le otorgan a *Mario* una vida extra. Estos elementos de decoración se muestran en la figura correspondiente al tipo de fragmento "*llano*" que se describe a continuación.

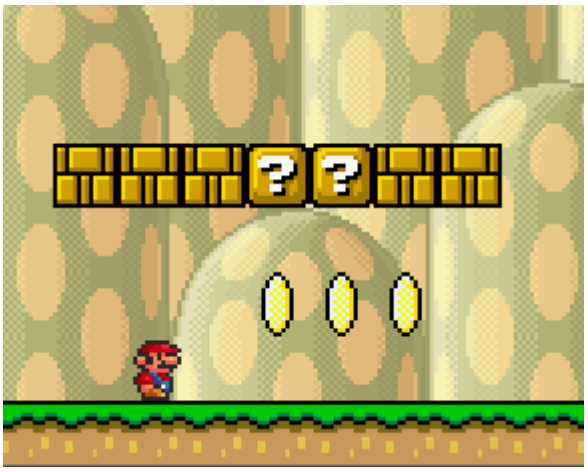


Figura 1 – Fragmento de nivel "llano"

Tipo llano: contiene una hilera de bloques en el suelo a la misma altura. En la figura de la izquierda se puede apreciar un ejemplo de fragmento llano con tres monedas y siete bloques, dos de los cuales contienen *power-ups* (los bloques con la "?"). Este tipo de fragmento puede contener cualquier tipo de enemigo.

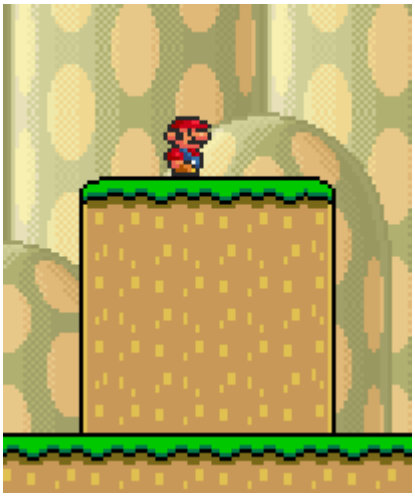


Figura 2 – Fragmento de nivel "colina"

Tipo colina: contiene una hilera de bloques en el suelo a la misma altura y sobre ellos, de un ancho arbitrario no mayor al del tramo llano una serie de bloques que forman una columna (o colina) a una altura homogénea y arbitraria (pero accesible para *Mario*) En la figura de la izquierda se puede apreciar un ejemplo de fragmento colina de tres bloques de altura y anchura. Este tipo de fragmento puede contener cualquier tipo de enemigo.

Figura 3 – Fragmento de nivel "tubería"

Tipo tubería: contiene una hilera de bloques en el suelo a la misma altura y sobre ellos, de un ancho fijo, una tubería color verde de la que pueden salir únicamente dos tipos de enemigo (*flor saltarina* y *flor de fuego*). En la figura de la izquierda se puede apreciar un ejemplo de fragmento tubería a distinta altura del fragmento anterior de nivel.

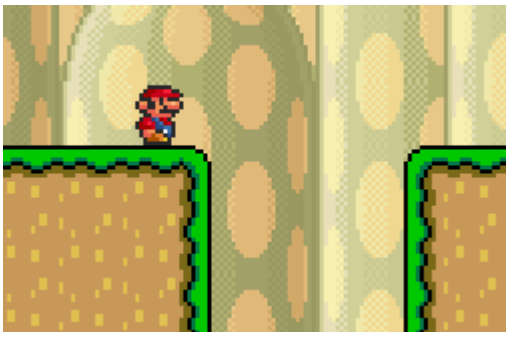


Figura 4 – Fragmento de nivel "salto"

Tipo salto: contiene una hilera de bloques en el suelo a la misma altura y entre ellos, un vacío de longitud arbitraria (pero en principio salvable por *Mario*). En caso de no saltar dicho vacío, *Mario* cae perdiendo una vida. Este tipo de fragmento no contiene enemigos (caerían al vacío).

Figura 5 – Fragmento de nivel "cañón"

Tipo cañón: contiene una hilera de bloques en el suelo a la misma altura y sobre ellos, de un ancho igual a un bloque y sobre una columna negra de altura arbitraria descansa un enemigo de tipo *cañón*, capaz de disparar proyectiles a izquierda o derecha (dependiendo de dónde se encuentre *Mario*). En la figura de la izquierda se puede apreciar un ejemplo de dos fragmentos tipo cañón.

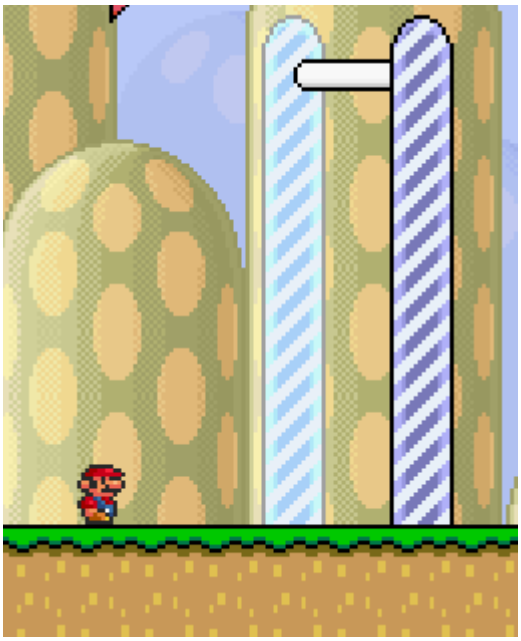


Figura 6 – Fragmento de nivel "llano" con fin de nivel

Salida: realmente consiste en un fragmento de tipo llano sobre el que se dibuja el elemento que indica la salida del nivel. Como es lógico se sitúa al final, rellenando los huecos que falten para completar la longitud del nivel, siempre y cuando dicha longitud restante sea inferior a un umbral determinado usualmente definido en los algoritmos de generación de niveles.

Figura 7 – Goompa

Goompa: el enemigo más sencillo de la saga *Super Mario Bros*. Es una especie de champiñón malvado con colmillos que se mueve en una misma dirección hasta que topa con un obstáculo. No cambia de dirección en caso de llegar a un desnivel o salto, cayendo por ellos. Es posible eliminarlo saltando encima, golpeándolo con una concha de tortuga o lanzándole una bola de fuego (si *Mario* está en dicho modo).

Figura 8 – Tortuga verde

Tortuga verde: su comportamiento es igual al del Goompa. Además de en la apariencia, se diferencia de éste último en que si la forma de eliminarlo es saltando encima, la primera vez que se salta sobre el mismo deja en el sitio su concha vacía, pudiendo cogerla o proyectarla en una dirección (por ejemplo para matar a una serie de enemigos agrupados).

Figura 9 – Tortuga roja

Tortuga roja: análoga a la tortuga verde, aunque algo más "inteligente". En caso de encontrar un desnivel o un salto, la tortuga roja cambia su dirección para evitar caerse.

Figura 10 – Spikey

Spikey: es uno de los enemigos más duros. Se comporta como una tortuga verde (o goompa), con la salvedad de que la única forma de eliminarlo es lanzándole una concha de tortuga o esperar a que caiga por un salto.

Figura 11 – Forma alada

Los cuatro tipos de enemigos mencionados anteriormente disponen también de su **forma "alada"**. Una característica común entre ellos es que se mueven siguiendo una dirección fija pero sin variar su altura. La forma alada de los citados enemigos hace que vayan "*dando saltos*". Es decir, a medida que avanzan no sólo se ve modificada su trayectoria en el eje X, sino también en el Y, haciéndolos más difíciles de esquivar. Igualmente, en caso de intentar saltar encima de ellos en su forma alada, en lugar de eliminarlos les quitaremos dicha propiedad, convirtiéndolos en su versión normal.

Figura 12 – Planta carnívora

Planta carnívora: sólo está presente en algunas tuberías. Existen dos versiones, la "**saltarina**" que sale periódicamente de la tubería y se vuelve a esconder y la de "**fuego**", que en *Mario AI* sale de la tubería y se comporta como un goompa. La única forma de acabar con una planta saltarina es lanzándole la concha de una tortuga o lanzarle una bola de fuego (cuando sea posible).

Figura 13 – Bala de cañón

Bala de cañón: proyectil disparado por los cañones. Sigue una trayectoria recta hasta que sale del nivel o se le salta encima. Puede resultar útil para salvar ciertos desniveles.

3.2.3. Algoritmos planteados

Como se comentó anteriormente, la primera solución desarrollada consistió en implementar un generador de niveles aleatorio, al que pasándole una dificultad esperada y mediante una serie de reglas de generación, pudiera construir niveles con dicha dificultad. Para realizar este generador se partió del implementado por defecto en *Mario AI*. Fue necesario realizar una exhaustiva evaluación previa del código de *Mario AI* a fin de determinar cómo se generan e interactúan los niveles con el resto de componentes del *framework*. Una vez comprendido el mecanismo, fue necesario adaptar las funciones de forma que se aislara la parte de generación aleatoria (generador aleatorio) y a su vez pudiera ser invocado desde una interfaz genérica en Java™ (lenguaje de programación en el que está implementado *Mario AI*). De la versión inicial del generador de aleatorios se conservaron las reglas de generación de *fragmentos de nivel*, de *enemigos* y el "*decorador*" (monedas, bloques y *power-ups*). Además se implementaron gracias a su *super-clase* (generador de niveles genérico) métodos para **guardar** y **cargar el nivel en fichero**, siendo necesario definir también el formato del mismo.

La idea básica detrás del generador de aleatorios es "*tirar un dado*" (generar un número aleatorio) tantas veces como fragmentos de nivel sea necesario generar. En base a la dificultad pasada por parámetro, se determinan cuántos elementos (en porcentaje total) debe haber de cada tipo. Así pues, se genera mediante un bucle una serie de fragmentos hasta completar la longitud máxima del nivel. Cuando falta menos de un umbral determinado (unos 20 bloques), cesa la generación mediante el bucle y se configura manualmente la generación del "*fin de nivel*" rellenando el espacio restante. En las generaciones intermedias de fragmentos, dependiendo del tipo seleccionado se pueden añadir enemigos (ej. en un fragmento tipo llano o colina), cuyo tipo dependerá en mayor medida de la dificultad elegida y su número de un número generado aleatoriamente.

Con el mapa del nivel almacenado en un *array* bidimensional de datos binarios (contiene los tipos de *sprite* correspondientes a los distintos elementos generados), se procede a llamar a una función de decoración. Dicha función determina las zonas en las que hay fragmentos "decorables" (principalmente llanos) y en función de números generados aleatoriamente en un umbral dado por la complejidad, asigna zonas con monedas, bloques y/o *power-ups*.

Llegado este punto, se tiene generada de manera aleatoria el nivel completo y está listo para ser utilizado (ya sea evaluándolo o jugándolo mediante la interfaz gráfica de *Mario AI*).

Después de realizar un gran número de pruebas variando el valor de la dificultad, se observó un problema con este generador. Aunque a priori los niveles se ajustan relativamente bien a la dificultad seleccionada, el uso de reglas tan estrictas hace que llegado cierto valor de dificultad las soluciones sean muy similares. Concretamente se puede apreciar que a partir de valores entre 25 y 32 los niveles están compuestos principalmente de zonas de saltos y zonas de cañones. Igualmente los enemigos que aparecen en este tipo de niveles son "*spikes alados*" o "*goompas alados*", haciendo niveles no tan complicados como se puede pretender y bastante monótonos al no haber variedad de zonas y/o enemigos.

A fin de resolver este problema se desarrolló una solución basada en un algoritmo genético.

La principal ventaja de la solución mediante un algoritmo genético es que al generar los individuos (niveles) de forma totalmente aleatoria, aunque se incremente la dificultad del nivel no tiene por qué haber pérdida de variedad de enemigos y zonas, y los niveles van a poder tener dificultades mucho más ajustadas (en lugar de un número entero entre cero y treinta y dos, será un número de varias centenas). A continuación entraremos en detalle sobre la codificación e implementación de éste algoritmo.

3.2.4. Codificación de la solución

Uno de los principales aspectos a la hora de definir un algoritmo genético es el de determinar una codificación adecuada para representar a los individuos que van a someterse a los distintos operadores genéticos del algoritmo. En el caso que nos concierne, queremos generar niveles para *Mario AI*, con lo que tenemos que tener en cuenta los distintos elementos que componen el nivel y sus características (ver punto 3.2.2).

Un individuo está formado por una serie de genes, y cada gen tiene a su vez una serie de cromosomas. Cada cromosoma representa una propiedad asociada al gen. En nuestro caso, hemos extendido la funcionalidad de generación de tramos de nivel, haciendo corresponder a cada gen un tramo del nivel generado (el individuo estará formado por un conjunto de tramos). Si analizamos un tramo, podemos observar que dispone de una serie de características que lo diferencian del resto y lo hacen "único". Dichas características serán las que modelen los cromosomas (cada cromosoma representará una propiedad). Las características que representan un tramo son las siguientes:

- **Tipo del tramo**: llano, colina, salto, tubería o cañón.
- **Ancho del tramo**: valor entre 0 y 7 bloques, ya que es la máxima distancia por defecto que puede cubrir *Mario* con un salto.
- **Altura del tramo**: valor entero positivo entre 0 y 3 bloques. Altura relativa respecto a la parte más baja del nivel.
- **Número de enemigos en el tramo**: valor menor o igual al ancho del tramo.
- **Tipo de los enemigos en el tramo**: goompa, goompa alado, tortuga verde, tortuga verde alada, tortuga roja, etc.
- **Altura del cañón**: define la altura que tendrá el cañón respecto al tramo.
- **Altura de la tubería**: define la altura que tendrá la tubería respecto al tramo.
- **Altura de la colina**: define la altura que tendrá la colina respecto al tramo.
- **Anchura de la colina**: valor menor o igual al ancho del tramo, y como tamaño mínimo igual a dos.
- **Número de bloques**: cantidad de bloques de ladrillos que pueden aparecer en un tramo de tipo llano. Valor menor o igual al ancho del tramo.
- **Número de power-ups**: en caso de haber bloques, puede que algunos contengan elementos especiales (setas y flores). Valor menor o igual al número de bloques.
- **Número de monedas**: en tramos llanos (o colinas si se modifica la función que "decora" el nivel) indica la cantidad de monedas presentes en el mismo. Valor menor o igual al ancho del tramo.

Transformar cada característica anterior en un cromosoma implicaría tener 12 de ellos. Podemos simplificar la codificación agrupando valores que serán tratados de una forma u otra a la hora de calcular el *fitness*. Por ejemplo, las características "altura de la tubería",

"altura del cañón" y "altura de la colina" se pueden reducir a la característica **altura del elemento secundario**. De la misma forma, "anchura de la colina", "número de bloques", "número de *power-ups*" y "número de monedas" se pueden reducir a **anchura del elemento secundario**. En este caso, el valor de la variable será un entero de tres cifras, donde las centenas indicarán el "número de monedas" del tramo, las decenas el "número de bloques" y las unidades el "número de *power-ups*" o "anchura de la colina". Con una simple función matemática se puede obtener rápidamente cada uno de los valores de una misma variable, al igual que codificar distintos valores en uno.

De esta forma, un nivel será representado por **7 cromosomas**. Debemos emplear este número porque se necesita una representación **inequívoca** de cada nivel para que cuando se vuelva a generar un individuo idéntico, dé como resultado el mismo nivel, y que no haya factores que queden olvidados, como el número de enemigos o la altura de algún elemento secundario, puesto que esto haría que cambiase el *fitness* de dicho individuo. Vamos a representar dichos cromosomas mediante la letra **C** con un subíndice numérico. La correspondencia entre cromosomas y características es la siguiente:

C_0 **Tipo del tramo**

C_1 **Ancho del tramo**

C_2 **Altura del tramo**

C_3 **Número de enemigos en el tramo**

C_4 **Tipo de los enemigos en el tramo**

C_5 **Tipo de los enemigos en el tramo**

C_6 **Altura del elemento secundario**

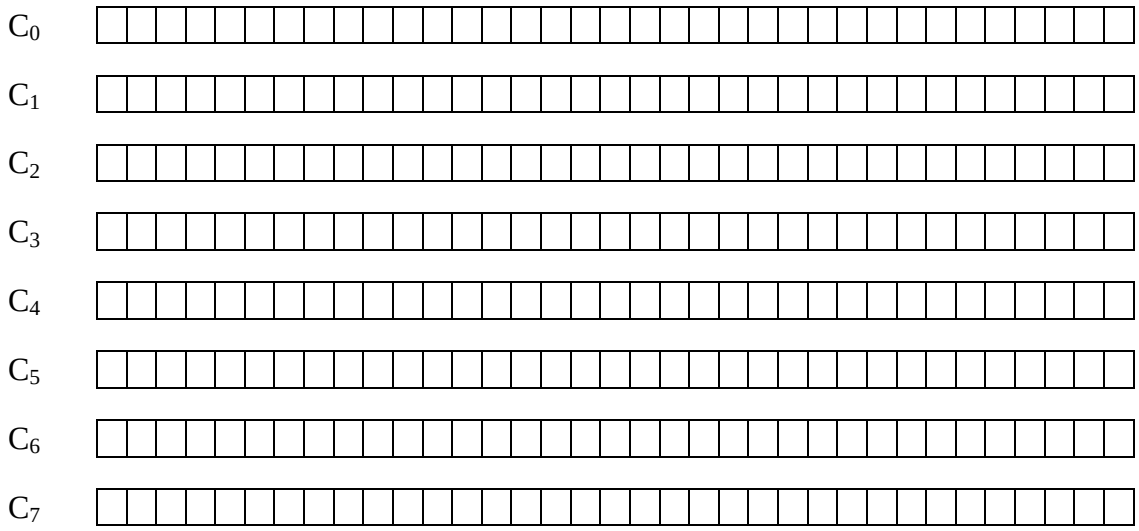
C_7 **Anchura del elemento secundario**

El conjunto anterior va a definirnos un gen.

Respecto a un individuo, aunque se ha codificado la solución para que se pueda modificar por parámetro de forma sencilla el número de genes a procesar, se ha fijado un valor estándar para la creación de niveles. En un principio se pensó en 100 genes, pero al tener un tamaño variable de tramo por gen, se podían construir niveles desde longitud 135 hasta 735 bloques, que podían resultar demasiado largos. El valor final asignado es 35. La justificación es la siguiente: como se ha comentado, implícitamente el número de genes va a definir el ancho de un nivel (suma de los valores de todos sus cromosomas 1 más 35 bloques destinados a crear el inicio y el fin del nivel). El valor de la longitud de cada tramo se va a inicializar aleatoriamente, con lo que en un caso peor, si tenemos un elevado número de genes, por muy sencillo que sea el nivel (llano, con la misma altura y sin enemigos) podría darse el caso de que no diera tiempo a que *Mario* pudiera alcanzar el final del mismo (por defecto se le otorga al usuario un tiempo de 200 segundos). Tras hacer varias pruebas y viendo que la longitud media de tramo generada era de 4, pareció adecuado dar un valor de 35, ya que en caso peor construiría un nivel de 280 bloques. Un

nivel de dicha longitud a priori podría ser completado en tiempo (aunque tenga dificultad elevada).

Así pues, un individuo de una población se representa de la siguiente forma (donde cada cuadro vacío es un valor que dependerá del rango de valores admisibles por ese cromosoma):



Con todo lo anterior, el generador de niveles genético recibirá como parámetros principales el número de individuos que compondrá la población (se ha elegido un valor de 50, al resultar el tiempo medio de convergencia más rápido que en poblaciones de 20 y 30 individuos), el número de genes que tendrá cada individuo de la población y la dificultad (*fitness*) que se pretende conseguir. Una vez inicializada la población de forma totalmente aleatoria, se aplicará sobre la misma una serie de operadores genéticos hasta que se alcance un criterio de parada. Dicho criterio consiste en que el *fitness* de la población se encuentre en un rango de valores entre mas-menos el uno por cien del *fitness* buscado. Para evitar largas generaciones en las que pudiera no converger, se añadió también un criterio de parada después de 10000 generaciones, número suficientemente grande ya que en casos peores, las convergencias rondaban en torno a las 9000 generaciones, y siendo de media entre 23 y 200.

Para la función de *fitness*, se evalúa cada tramo en función de el tipo que sea (es más difícil un cañón que un tramo recto sin enemigos), del tipo de los enemigos que tenga y del tipo de mejoras que contenga (monedas, *power-ups*), asignando una serie de valores de dificultad a cada elemento. Todos estos valores se han establecido en base a la retroalimentación obtenida tras probar múltiples niveles de diversas dificultades generados en *Mario AI* (observando qué situaciones eran más complejas) y ajustando los valores fijados a la complejidad de cada elemento a fin de obtener resultados coherentes. Por otra parte, como se intenta aproximar el valor del *fitness* a uno dado, la estrategia utilizada ha sido la de restar al *fitness calculado* el valor del *fitness dado,* en *valor absoluto* y tratar de hacer que dicho valor tienda a cero (minimizar dicho valor, al que llamaremos ***fitness* normalizado**).

Como operadores genéticos se han implementado los siguientes:

- **Torneo**: copia directamente el individuo más prometedor tres veces a la siguiente generación, y que aleatoriamente selecciona dos individuos, compara sus *fitness normalizados* y promociona al mejor a la siguiente generación. Así hasta completar una población completa (50 individuos).
- **Mutación**: cambia un porcentaje de genes aleatorios (especificados por parámetro) de individuos seleccionados aleatoriamente de la población teniendo en cuenta qué porcentaje de individuos se mutarán. Es decir, dicha operación recibe dos parámetros, el porcentaje de individuos a mutar y el porcentaje de genes a mutar.
- **Cruce**: realiza un cruce multipunto aleatorio entre dos individuos (gen a gen puede o no cruzarlos), devolviendo dos individuos nuevos resultado de la mezcla entre los dos.
- **Cruce porcentual**: recibe un valor entero correspondiente al porcentaje de individuos a cruzar y les aplica la operación de cruce definida anteriormente.

El modo de proceder por el algoritmo genético hasta que se llega al criterio de parada es el siguiente: en la generación inicial se realiza un torneo, a la población resultante se les cruza el 50% de sus individuos y se muta en un 50% de los individuos el 50% de los genes. Se revalúan los *fitness*, se re computa el *fitness normalizado*, se reordena la población en base a dicho *fitness normalizado* (teniendo al principio a los elementos más prometedores) y se repite el proceso decrementando en una unidad el valor de los porcentajes hasta que se estanca el valor en un 10%, mantenido para que no se estanque la población en un mínimo local. Una vez alcanzado el criterio de parada se devuelve la población generada.

3.2.5. Problema de la solución

A priori existen un par de problemas con la solución propuesta. El primero tiene que ver con la **diversidad genética**. Puede darse el caso de que los individuos de la población acaben tendiendo a parecerse a un *súper-individuo* que tenga valores iguales o muy parecidos (en cuanto al valor de sus cromosomas se refiere) respecto al resto de individuos, o dicho de otra forma, que los niveles generados por la población sean el mismo. Para evitar este caso, se ha implementado una función que evalúa la diversidad genética (o el parecido de un individuo con el resto). Esta función consiste en sumar una unidad por cada valor igual de cada cromosoma de cada gen entre dos individuos. Es decir, cuanto más alto el resultado de dicha función, más se parece este individuo al resto de individuos de la población. El caso ideal sería que el resultado fuese cero para cada individuo de la población, lo que significaría que todos los niveles generados son distintos y no tienen ninguna característica común (entre los mismos genes de distintos individuos). La operación de cálculo de diversidad genética se lleva a cabo después de cada generación, teniendo que calcular para cada individuo su valor respecto al resto de individuos. Al ser una población relativamente pequeña con pocos genes no requiere excesivo tiempo de cómputo. Por último, en caso de encontrar individuos con mismo *fitness* se favorece al que tiene el valor más bajo de ésta función (nivel más distinto al resto) y se penaliza al que tiene valor más alto.

El segundo problema tiene que ver con la evaluación de los niveles. Puede darse el caso de que aunque se haya generado un nivel coherente de una determinada dificultad, debido a la aleatoriedad de los valores presentes en dos genes adyacentes, exista un desnivel

suficientemente alto como para no poder ser salvado. En algunos casos es posible que se de una situación que permita salvar el desnivel, como por ejemplo que exista un cañón anteriormente a una altura adecuada que dispare una bala sobre la que saltar, y así alcanzar una altura extra, pero determinar tales situaciones puede resultar complicado, incluso en caso de utilizar una función de evaluación global del *fitness*. Por otra parte puede suceder que la configuración del nivel tenga algún elemento situado de forma que permita completar un nivel de dificultad aparentemente enorme con mucha facilidad (comentado anteriormente. Ejemplo de los *spikeys* precedidos de una tortuga roja).

Para evitar este último problema se ha dado un paso más. Además de emplear un algoritmo genético para la generación de los niveles se ha pensado en utilizar un sistema automatizado de resolución de niveles que los evalúe de forma independiente en cada generación, de forma que el resultado (puntuación) de dicho sistema pueda ser utilizado a modo de *fitness* en el algoritmo genético. Básicamente la estrategia propuesta es una *estrategia coevolutiva*.

3.3. DESARROLLO DE ALTERNATIVAS DE DISEÑO

Aparte de las dos primeras propuestas (*aleatorio con parámetros* y *algoritmo genético*), se sugieren otras dos, una de las cuales (*estrategia coevolutiva*) fue implementada y probada con éxito. La otra queda como línea de investigación en un trabajo futuro, aunque fuera una de las estrategias pensadas inicialmente. Consiste en un generador de niveles basado en el Aprendizaje Automático. Dicho generador propondría un nivel base al jugador y se realimentaría con las operaciones que realizase el mismo. Es decir, si por ejemplo el jugador completa con éxito el nivel, el agente evaluaría los elementos que más le han costado resolver (es factible, ya que los niveles generados con la interfaz de generación de niveles almacenan todos los eventos y momento en el que fueron realizados, permitiendo evaluar posteriormente la forma en que se completó el nivel), y una vez determinados dichos elementos, aumentaría la complejidad de los mismos un poco. En caso de fallar, el siguiente nivel generado reduciría la complejidad específica de dichos elementos, de forma que cada partida del jugador (o agente automático) pudiera generar un nuevo nivel teniendo en cuenta qué elementos debe potenciar a fin de que el usuario aprenda progresivamente a vencer los retos propuestos en el nivel. Esta alternativa no se llegó a desarrollar por su fuerte componente determinista. Tal y como se estudió en el estado del arte, es adecuado que en los niveles haya cierto grado de aleatoriedad y factores que este tipo de solución no podría cubrir. Sin embargo, como se ha comentado, sería una posibilidad para desarrollar una futura mejora en la que estudiar sus resultados.

Respecto a la estrategia implementada, se codificó un generador de niveles coevolutivo. Fue necesario analizar a fondo y modificar el código fuente del agente genético de *Mario* diseñado por Héctor Valero en su Proyecto Fin de Carrera [2] a fin de integrarla en el proyecto actual, implementando en una serie de clases auxiliares que proporcionan funcionalidad extendida, como es la de cargar un nivel de memoria o disco para ser evaluado con el agente de *Mario* y mostrar o bien estadísticas de la evaluación por pantalla o un demostrador gráfico de cómo completa el nivel el agente genético más prometedor generado. Esta última funcionalidad puede resultar útil, ya que si un jugador

se queda atascado en un nivel que no sabe cómo completar, puede recurrir a visualizar cómo completa el nivel un agente automático y tratar de emular las acciones posteriormente de forma manual.

El generador de niveles coevolutivo funciona de la misma forma que el generador genético anterior con un par de salvedades. Como resulta obvio, ha sido necesario adaptar el cálculo del *fitness* para tener en cuenta el resultado obtenido por el agente genético *Mario* de Héctor en lugar del resultado de una función de evaluación del contenido del nivel. Esto hace que los niveles tengan una complejidad que depende de la puntuación obtenida por el agente en los mismos. Como ventaja, se puede ver que los niveles generados se clasifican en base a la complejidad de ser completados (*jugabilidad*) en lugar de en base al tipo y cantidad de elementos que contienen. Esto permite detectar situaciones imposibles (el agente no es capaz de completar el nivel en ningún caso devolviendo una puntuación muy baja) o detectar posibles "trampas" para completar el nivel de forma más sencilla de lo que estaba previsto (devolviendo una alta puntuación al obtener más puntos). El problema es que no se puede establecer una relación directa entre el fitness del algoritmo genético y el del coevolutivo, entre otras cosas por culpa de estos factores.

Los niveles generados por el coevolutivo tienen una complejidad más objetiva a nivel jugador que los ejecute para divertirse (o pasar un mal rato), sin embargo, la gran "contra" de este generador es el elevado tiempo que tarda en conseguir la convergencia de una población. Utilizando un PC de última generación con un procesador Intel de cuatro *cores* y velocidad de procesamiento a 3.2 GHz, dotado de 16 GB de RAM DDR3 conectadas en *Tri-Channel*, la evaluación de cada nivel por parte del agente genético *Mario* oscila entre 35 y 50 segundos debido a la carga computacional derivada de la configuración de parámetros óptima sugerida en [2]. Teniendo en cuenta una población de 50 individuos, cada generación tarda aproximadamente 35 minutos. Cien generaciones son 58 horas. Teniendo en cuenta que la media de convergencia del generador de niveles genético oscila entre las 23 y las 200 generaciones en caso promedio, computar múltiples poblaciones de *fitness* específico puede llevar meses. Para simplificar el coste temporal del problema y poder hacer pruebas en tiempo, se relajó el criterio de parada y se redujo el número de individuos de la población a 15 y se estableció un número bajo de evaluaciones límite (25) teniendo en cuenta los mejores casos para el generador genético. Las pruebas que se hicieron, tardaron unas dos horas y media en obtener un individuo con el *fitness* buscado (cerca de 12 generaciones), aunque con las evaluaciones límite no llegaron a obtener una población completa con dicho *fitness* y una diversidad genética aceptable. Otro problema observado para este generador es que al utilizar un agente de evaluación genético, a la hora de procesar un mismo nivel en algunos casos puede no encontrar solución para el número de generaciones configurado por defecto, en otros casos puede encontrar una solución aceptable y en otros excelente. Esta diversidad de valores hace que la evaluación sea estocástica y no determinista, lo cual puede hacer que el tiempo de convergencia (si no fijamos un número máximo de evaluaciones) diste mucho en dos escenarios con los mismos parámetros. Cabe destacar que aun así, el algoritmo genético *Mario* es bastante fiable y eficiente, encontrando solución a los niveles evaluados en 50 iteraciones o menos.

3.4. VENTAJAS E INCONVENIENTES DE LOS DISEÑOS

	VENTAJAS	INCONVENIENTES
Opción 1: Aleatoriedad con parámetros	- Algoritmo sencillo. - Generador bastante eficiente y rápido. - Los niveles generados no presentan incoherencias o zonas imposibles de completar.	- Los niveles tienden a converger en tipo y número de elementos cuando la dificultad es elevada.
Opción 2: Algoritmo Genético	- Genera niveles más variados y de una dificultad más precisa que los aleatorios - Muy buena relación de tiempo consumido en generar los niveles frente a la calidad de los niveles obtenidos	- Puede generar niveles imposibles de completar debido a la generación aleatoria de todos los parámetros que definen a los niveles. - La complejidad de los niveles está basada en su contenido y no en la jugabilidad de los mismos
Opción 3: Coevolución	- Genera niveles muy adaptados a las necesidades del jugador (dificultad acorde con la puntuación que se consigue. Cuanto más baja, más difícil).	- Presenta un elevado coste computacional, lo que deriva en que el tiempo de convergencia del algoritmo sea extremadamente alto. - El proceso de evaluación (*fitness*) depende de un algoritmo genético que no siempre devuelve los mismos valores, con lo que el tiempo de convergencia puede variar enormemente para mismos parámetros de entrada.
Opción 4 (no implementada): Sistema basado en Reglas (Aprendizaje Automático)	- Se podrían desarrollar niveles más personalizados para cada usuario (si el usuario frecuentemente no puede completar el nivel porque se cae en los "saltos", se podría aprender de los movimientos de ese usuario para realizar un nivel con "saltos" más sencillos y según vaya avanzando ese tipo de nivel, hacer "saltos" más complejos)	- Determinista, por lo que ante las mismas circunstancias siempre se devolverá las mismas soluciones (los mismos niveles).

3.5. ELECCIÓN Y MEJORA DEL DISEÑO FINAL

3.5.1. Elección del diseño final

Para el diseño final, después de haber analizado y estudiado tanto el código de *Mario AI* como el código del proyecto del agente genético *Mario* de evaluación de niveles se decidió implementar un **entorno genérico de generación de niveles extensible**, desarrollando una serie de funcionalidades adicionales para dotarlo de una gran extensibilidad y un bajo acoplamiento con otros componentes. A modo de prueba y para aprovechar la toma de contacto con *Mario AI* se portó el sistema de generación aleatoria de niveles al entorno genérico, desarrollando un **generador aleatorio de niveles**. Seguidamente se implementó la solución basada en un **algoritmo genético** y se terminó por implementar la tercera mejora, basada en un **algoritmo coevolutivo**.

A nivel de **características útiles** directamente utilizables por el usuario de la aplicación final, se implementaron las siguientes opciones:

- Generar niveles aleatorios en caso de invocar la aplicación sin parámetros. La aleatoriedad de los niveles en este caso no sólo radica en el generador utilizado, sino también en la complejidad del nivel y en el tipo o apariencia dotado al mismo (superficie, subterráneo o castillo).
- Generar un nivel para ser jugado empleando parámetros personalizados, como son el tipo de generador a utilizar (aleatorio, genético o coevolutivo), la dificultad (o *fitness*) deseado y el tipo o apariencia dotado al nivel.
- Generar un nivel para ser guardado empleando parámetros personalizados. Análogo al anterior, pero teniendo que especificar un nombre de fichero para guardar el resultado. En caso de no dar un nombre se asignará un nombre único basándose en el generador, dificultad, tipo, fecha y hora en la que fue generado el nivel solución.
- Generar un lote de niveles al igual que un fichero de estadísticas sobre los mismos. Igual que en el caso anterior, pero en este caso el nombre sigue una secuencia incremental y lo que se puede especificar opcionalmente es el prefijo a utilizar en los nombres para distinguir los niveles resultantes de otros niveles generados mediante este mismo método.
- Cargar un nivel para jugar con él (interacción manual).
- Cargar un nivel para evaluarlo mediante el algoritmo genético de evaluación utilizado en el enfoque coevolutivo. En este modo se puede especificar si se desean obtener por consola las estadísticas tras haber evaluado el nivel (como son el estado de *Mario* al acabar, el número de enemigos eliminados, el tiempo empleado, etc.) o bien se puede especificar si se desea ejecutar de forma visual la resolución del nivel. Esta funcionalidad puede resultar útil en caso de que un jugador no encuentre la forma de completar un nivel, ya que la solución del mismo de forma visual y automática le puede aportar pistas para su próxima partida.
- Finalmente se implementaron una serie de opciones dentro del juego accesibles mediante atajos de teclado, que se vieron necesarias después de enfrentarse a algunas situaciones concretas. Estas opciones consisten en permitir guardar el nivel en curso utilizando como nombre por defecto "*level.mlv*" o cargar un nivel

con mismo nombre ubicado en el directorio de ejecución mediante la pulsación de las teclas "**Q**" y "**W**" respectivamente. Análogamente se implementó un sistema para silenciar los sonidos del juego o restaurar dichos efectos sonoros (pulsando "**M**", de *mute* o silenciar y "**N**" de *noisy* o ruidoso). En el caso de guardar/cargar, la decisión se tomó tras emplear más de dos horas en obtener un nivel mediante el generador coevolutivo y perderlo al finalizar la partida. En el caso de silenciar el juego, se contempló la opción de que el usuario de la aplicación puede que no se encuentre en un entorno propicio para ejecutar el juego con música y sonidos, o simplemente que se encuentre escuchando su propia música y no quera que se mezcle con los sonidos del juego.

3.5.2. Codificación del diseño final

Análogo a las nuevas funcionalidades implementadas y comentadas en el apartado anterior, se realizaron una serie de correcciones de errores, entre los que destacan la correcta asignación del fondo del escenario y música en función del tipo de nivel. En el código descargable de *Mario AI* sólo se muestra el nivel tipo superficie, con la música de superficie, independientemente de cómo se configure la clase *Play / PlayCustomized* (ejecutora). Otro fallo importante corregido es que al perder una vida en modo personalizado (PlayCustomized), *Mario AI* mostraba la ventana de juego en negro y no se podía continuar jugando. Este fallo fue el primero en ser corregidos.

Por otra parte, respecto al proyecto de algoritmo genético de evaluación de niveles, se implementaron un par de clases adicionales con el fin de proporcionar una serie de utilidades a la herramienta de evaluación así como una interfaz de evaluación genérica que aprovecha las ventajas del proyecto. Ambas clases se integraron dentro de un paquete llamado *BeaUtils*.

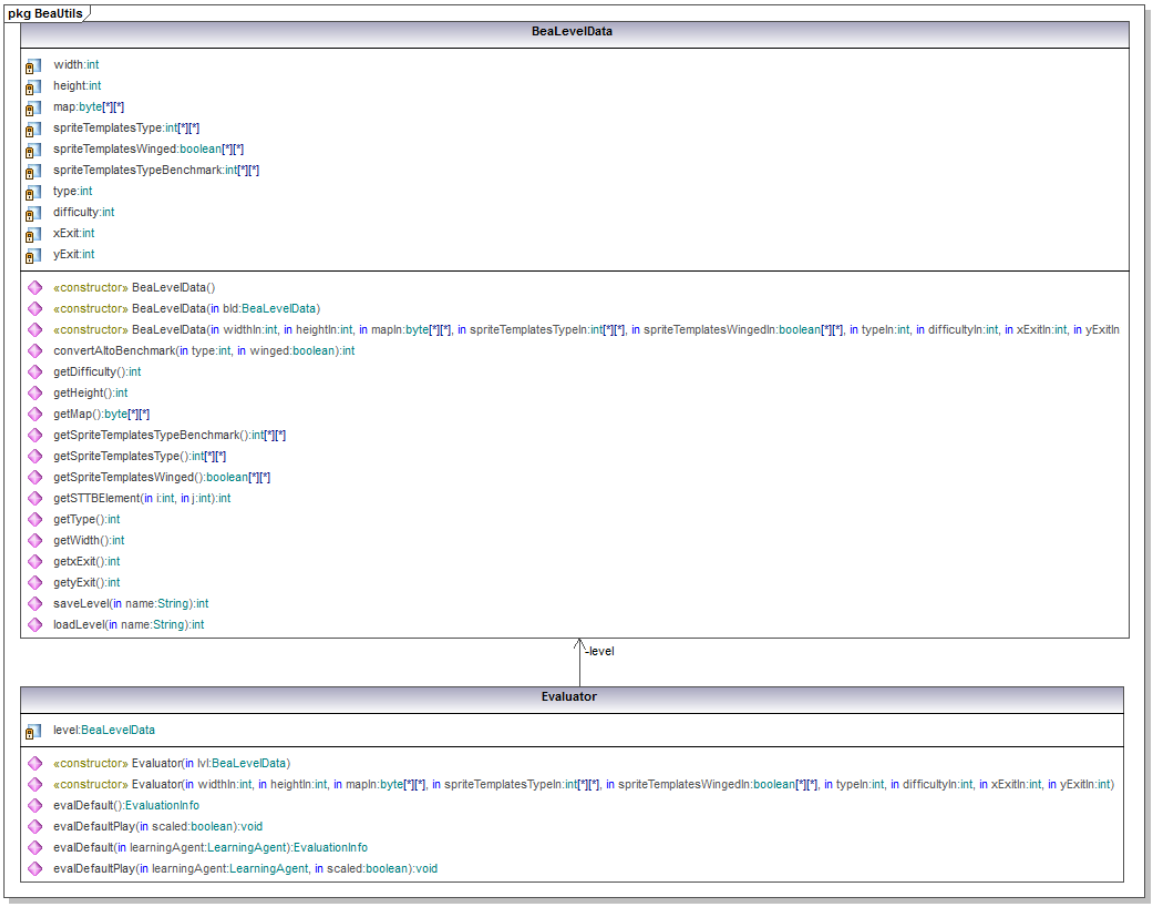


Figura 14 – Contenido del paquete BeaUtils añadido al proyecto de Héctor Valero

Como puede apreciarse en la figura anterior, la clase *BeaLevelData* proporciona métodos para almacenar y cargar en / de fichero las características que determinan un nivel. Dichas características son el ancho y alto del mismo, el *array* bidimensional que contiene el mapa del nivel, una serie de *arrays* bidimensionales que almacenan las características de enemigos del nivel en sus respectivas posiciones, el tipo o apariencia del nivel, el generador empleado, la dificultad (o *fitness*) asociada a dicho nivel y la posición de la salida del mismo. Cabe destacar que en el código original, la información sobre los enemigos se almacena en una clase tipo *SpriteTemplates*, pero debido a que la versión de *Mario AI* es diferente respecto a la del proyecto actual, la clase *SpriteTemplates* es incompatible y no puede ser convertida directamente. Además la representación de los enemigos se realiza mediante diferentes valores en las versiones, de forma que lo que en una versión era un hueco en blanco, en la otra una tortuga roja. Por ello resultó necesario realizar una función encargada de hacer traducción y conversión entre los tipos de *SpriteTemplates*. Una vez implementada la clase *BeaLevelData* ya era posible realizar la carga de niveles construidos por el generador genérico implementado en el presente proyecto, independientemente del generador específico utilizado.

La segunda clase implementada íntegramente fue la de *Evaluator*. Dicha clase se comporta como un conector con los mecanismos de evaluación de niveles. Instanciando la clase con un nivel almacenado en un objeto tipo BeaLevelData es posible realizar la

ejecución de un evaluador de niveles cualquiera soportado por la herramienta de pruebas *Mario AI*, o bien por el evaluador específico desarrollado por Héctor (perteneciente al grupo anterior), con los parámetros óptimos configurados por defecto. Además las funciones implementadas permiten obtener las estadísticas resultado del evaluador o bien mostrar la ejecución del mismo completando un nivel en una ventana.

Operaciones adicionales que se realizaron fueron modificar muchas de las clases pertenecientes al proyecto del Héctor para poder integrar los nuevos elementos en el entorno de evaluación consiguiendo una evaluación genérica de los niveles (no sólo por el agente genético) y limpiar mucha información irrelevante mostrada por pantalla (trazas). No menos importante resultó la tarea de migrar el proyecto de la herramienta IntelliJ (de pago) con la que fue desarrollado a NetBeans IDE. Una vez migrado, se intentó y consiguió generar un paquete (compendio de bibliotecas y clases) mediante OneJAR y Apache ANT (integrado en NetBeans). Esto permite utilizar la nueva versión del proyecto en forma de librería compilada y portable en cualquier otro proyecto de Java™ de forma limpia.

Respecto al código implementado en el proyecto, a continuación se muestra una visión global del paquete *BeaGenerator* implementado. Se verá por partes para tener una visión más clara de lo que hace cada clase.

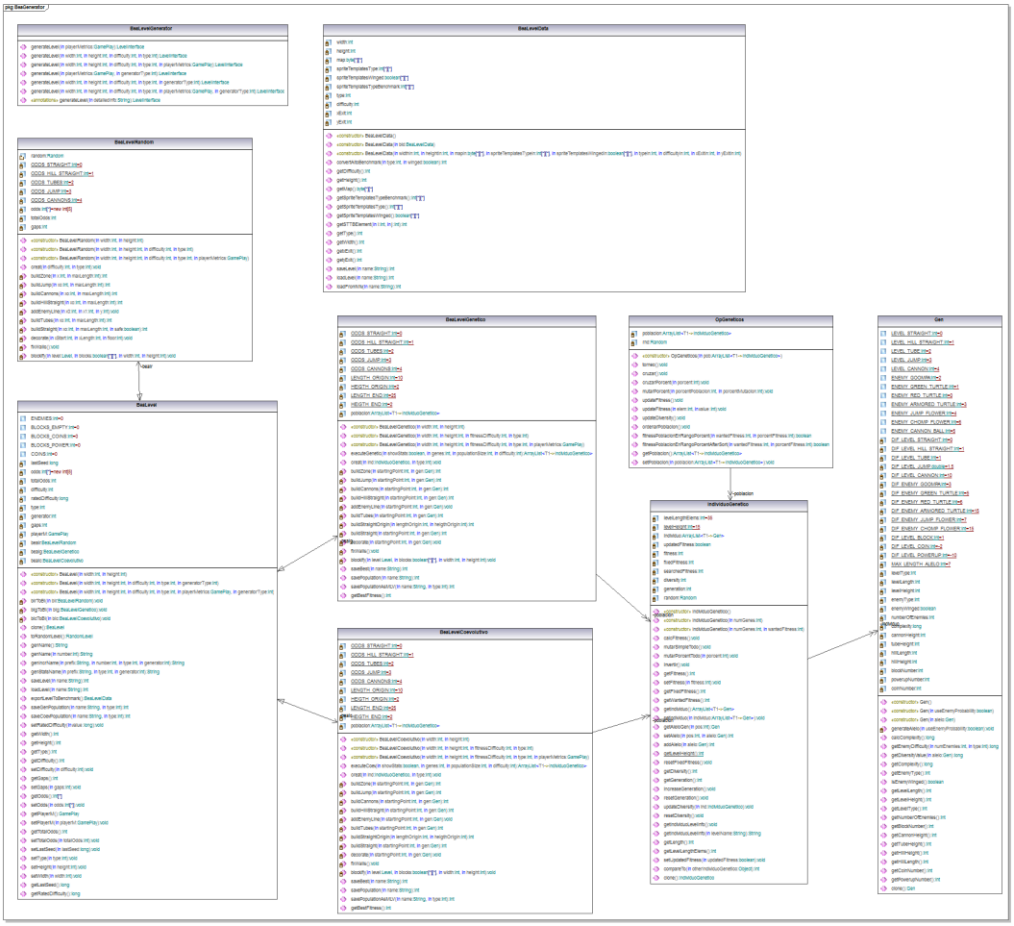


Figura 15 – Paquete *BeaGenerator*

Figura 16 – Clase *BeaLevelData* del paquete *BeaGenerator*

En la figura superior podemos observar la clase *BeaLevelData*. Su comportamiento es el mismo que el implementado en la clase análoga del paquete *BeaUtils* del proyecto del Héctor Valero. Es precisamente esta clase la que se ocupa de convertir los datos del nivel para ser procesados mediante el al evaluador automático.

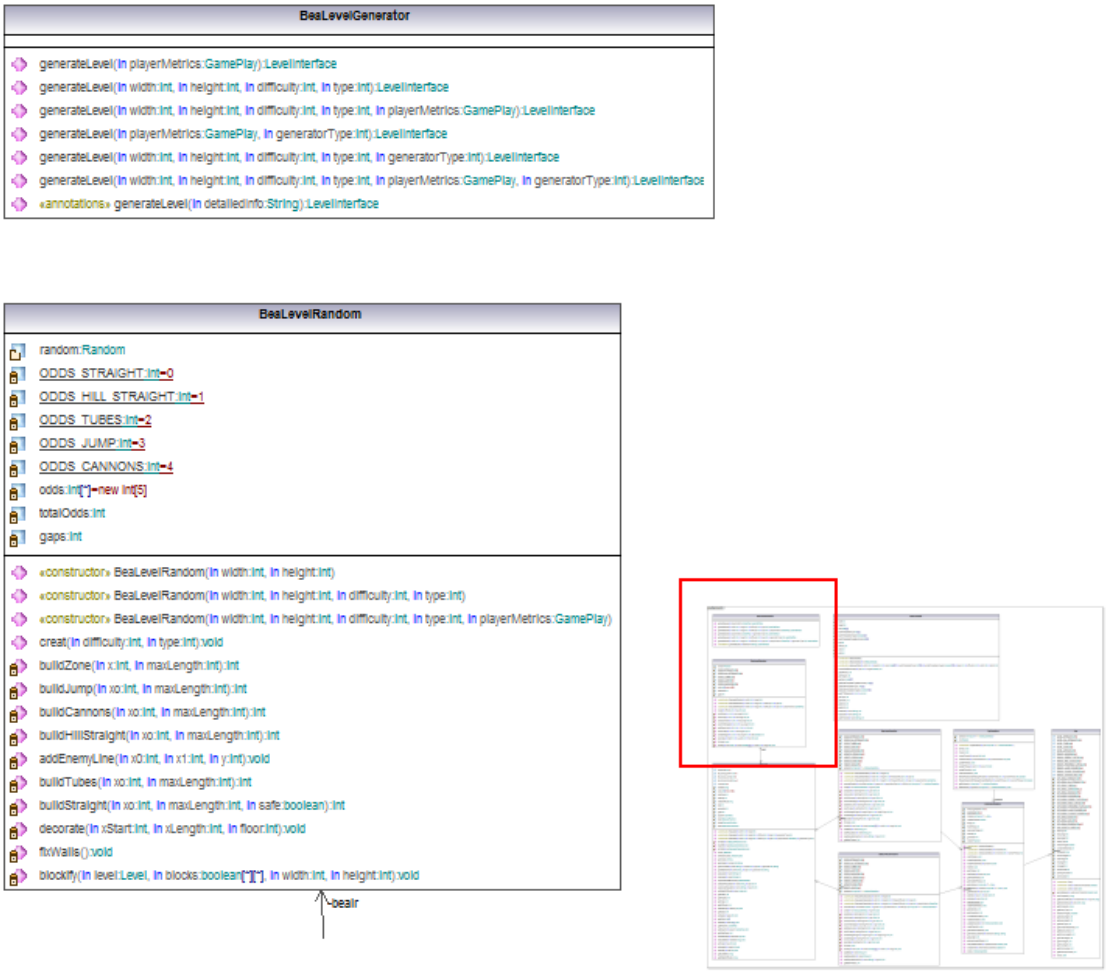


Figura 17 – Clases *BeaLevelGenerator* y *BeaLevelRandom* del paquete *BeaGenerator*

La clase *BeaLevelGenerator* es una interfaz utilizada por *Mario AI* para invocar una serie de métodos con distintos parámetros de entrada a fin de obtener niveles instancias de niveles o generadores de niveles.

La clase *BeaLevelRandom* se corresponde con el generador aleatorio de niveles. Las funciones más importantes de este método son: "*creat*", que genera un nivel entero en base al tipo y dificultad pasados por parámetro, "*buildZone*", invocada desde "*creat*" que en base del tipo pasado por parámetro invoca a una función u otra de creación de fragmentos de nivel. La función "*decorate*" se ocupa de rellenar el nivel con enemigos, monedas, etc., "*fixWalls*" genera el contorno (techo y paredes) en caso de niveles tipo castillo o subterráneo y "*blockify*" decora con bloques y *power-ups* algunas zonas del nivel.

Figura 18 – Clases *BeaLevel*, *BeaLevelGenetico* y *BeaLevelCoevolutivo* del paquete *BeaGenerator*

La clase *BeaLevel* es quizá la más importante del paquete en lo referente a los generadores de niveles. Es la clase padre de todos ellos. Extiende la clase *Level* de *Mario AI* e implementa *LevelInterface* y *Serializable*. La primera para devolver los niveles a *Mario AI* y la segunda para poder guardar y cargar de fichero los distintos atributos que definen un nivel. En esta clase se definen funciones para guardar una posible población de individuos a fichero, para exportar los datos del nivel a un objeto tipo *BeaLevelData*, funciones para generar estadísticas sobre el contenido de los niveles (cantidad de enemigos y tipos de elementos) y funciones para convertir cualquier nivel generado de

forma que se puedan emplear los módulos de estadística de juego de *Mario AI*. Dichos módulos guardan en fichero y muestran por consola cada uno de los eventos que suceden a lo largo de la partida. De esta forma se puede analizar al detalle el comportamiento del jugador frente a un nivel determinado.

Las clases *BeaLevelGenetico* y *BeaLevelCoevolutivo* son prácticamente iguales, diferenciándose en la forma de ejecutar el algoritmo de generación de niveles. Como se comentó anteriormente, para el algoritmo genético, el *fitness* se basa en una fórmula que tiene en cuenta el contenido del nivel. En el caso del coevolutivo, el *fitness* se obtiene de probar el nivel mediante un evaluador genético. Las funciones más importantes en ambos casos son *executeGenetic* y *executeCoev*, encargadas de ejecutar los algoritmos de generación correspondientes. El resto de funciones sirven para instanciar el nivel en base al individuo y para exportar los individuos o la población completa a fichero.

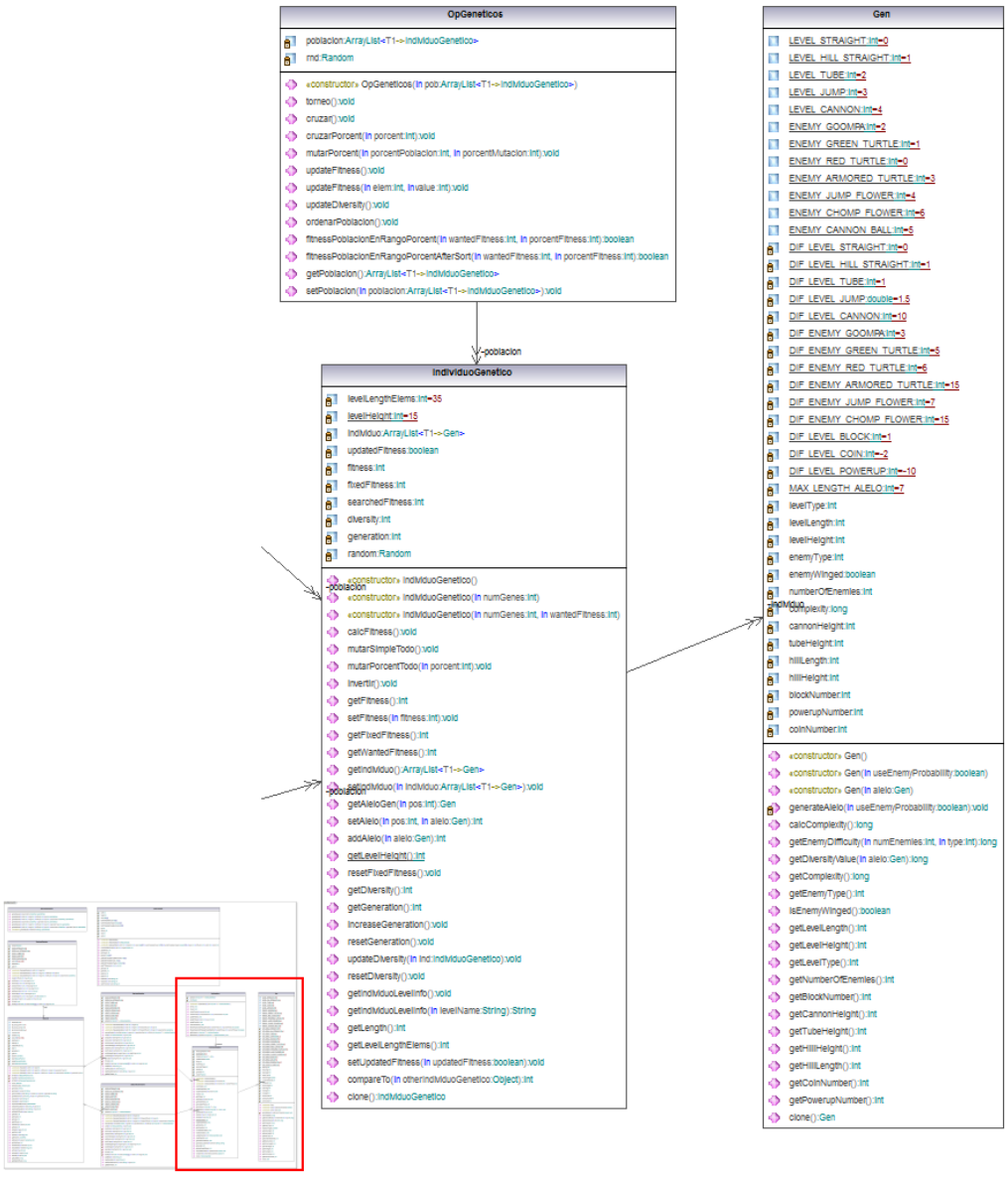


Figura 19 – Clases *OpGeneticos*, *IndividuoGenetico* y *Gen* del paquete *BeaGenerator*

Estas tres últimas clases definen los distintos operadores genéticos empleados en el generador de niveles genético y coevolutivo, así como la definición de individuo y de gen.

La clase *OpGeneticos* contiene una población y una serie de métodos para aplicar operaciones genéticas sobre ella. Las principales operaciones son las definidas en los métodos *torneo*, *cruzar*, *cruzarPorcent*, y *mutarPorcent*. Encargadas de hacer los torneos, cruces y mutaciones. Por otra parte se han implementado funciones para actualizar los valores de la diversidad genética de la población (*updateDiversity*), para actualizar el *fitness* (*updateFitness*) e incluso para automatizar la comprobación que determina si todos los individuos de la población tienen un valor de *fitness* comprendido en un intervalo determinado especificado por parámetro.

Las clases *IndividuoGenetico* y *Gen* son las que implementan conceptos definidos anteriormente, como el de individuo y gen. Proporcionan métodos específicos para, por ejemplo comparar dos individuos diferentes (*compareTo*) o para obtener la diversidad genética de un individuo respecto a otro (*updateDiversity*).

El proyecto final se ha desarrollado en NetBeans IDE, y se ha configurado una tarea específica de compilación que genera un único fichero .jar con el contenido de todas las librerías, recursos y clases del proyecto, de forma que sea muy sencilla su distribución y uso. Para ello, al igual que en la migración del proyecto de Héctor se han empleado, las librerías OneJAR y un script XML de Apache ANT.

3.6. CONCLUSIONES

En este capítulo, se han presentado una serie de propuestas para afrontar el problema de generación de niveles para *Mario AI* de forma automática. Se han propuesto ideas básicas para el diseño que posteriormente se han ido mejorando y madurando hasta terminar obteniendo varias alternativas válidas de generación de niveles.

Las alternativas implementadas en el proyecto han sido un enfoque aleatorio (adaptado del existente en *Mario AI*), un enfoque de generación mediante un algoritmo genético, y finalmente un enfoque basado en la coevolución, a fin de para resolver algunos problemas encontrados en algún nivel generado genéticamente, y también para proponer otra metodología de evaluación de los niveles (en vez de basada en la complejidad de los elementos que contiene, basada en la experiencia de juego).

Se han recogido y comentado las principales ventajas e inconvenientes de cada método, siendo el aleatorio el generador más rápido y sencillo, pero cuyos niveles a partir de una determinada dificultad únicamente contienen unos pocos tipos distintos de elementos haciéndolos monótonos. El algoritmo genético mejora la diversidad de los niveles generados y permite especificar mayores niveles de complejidad. Su tiempo de convergencia es relativamente pequeño (no llega al medio minuto en un PC de gama alta), pero presenta problemas para detectar algunos niveles imposibles. Finalmente la estrategia coevolutiva permite detectar este tipo de problemas mediante la evaluación de los niveles generados, repercutiendo en un elevadísimo coste computacional.

Finalmente se han presentado las distintas clases y funcionalidades implementadas a fin de conseguir un diseño final funcional.

En el siguiente capitulo (cuarto), se comentarán los resultados obtenidos tras las pruebas y evaluación de los enfoques de generación de niveles implementados.

4. RESULTADOS Y EVALUACIÓN

4.1. INTRODUCCIÓN

En este apartado vamos a explicar las pruebas llevas a cabo en el proceso de evaluación y a mostrar y comentar los resultados obtenidos en el mismo.

4.2. EXPLICACIÓN DEL MÉTODO DE EVALUACIÓN

Básicamente el proceso de evaluación ha consistido en comprobar que los distintos generadores producían niveles acorde a los requisitos definidos para ellos (valores de dificultad o *fitness* y diversidad genética). Recurriendo a funciones específicas que permiten mostrar información sobre los individuos se ha podido comprobar que los niveles generados genética y coevolutivamente son correctos.

Análogamente al análisis teórico de los niveles, se ha realizado un análisis práctico, probando distintos tipos de niveles generados y comprobando si se pueden apreciar diferencias notables entre ellos. Este análisis práctico es más subjetivo, ya que cada persona puede encontrar más o menos dificultad a la hora de superar determinadas partes de un nivel.

Debido a la cantidad de tiempo empleado en crear niveles mediante el generador coevolutivo, se ha decidido utilizar el agente automático incorporado en el mismo como sistema evaluador de los niveles generados por la propuesta aleatoria y la propuesta genética, observando detalles bastante interesantes que se comentarán más adelante. Concretamente en este método de evaluación nos centraremos en un estudio realizado tomando en cuenta poblaciones de 50 individuos generados mediante el enfoque genético, para *fitness* comprendidos entre 50 y 750 (dificultad baja a moderada alta). Cabe destacar que debido a la codificación de las soluciones evolutivas, el valor del *fitness* va a estar ligado al tamaño del individuo, por ello todas las pruebas se han hecho tomando como referencia individuos de 35 genes (cuanto mayor número de genes, más partes puede tener un nivel, pudiendo hacerlo más complicado). A estos individuos generados (750 niveles) se les ha evaluado mediante la funcionalidad de la aplicación final y un pequeño script de automatización, recogiendo los datos en ficheros de texto y obteniendo unas medias finales que se mostrarán en el punto siguiente. Así hemos podido observar detalles al comparar características promedio de los niveles con características promedio del evaluador para dichos niveles, al igual que el *fitness* obtenido en ambos casos (asociado al nivel y asociado a la evaluación).

4.3. COMPARATIVA DE LOS DISTINTOS MÉTODOS EMPLEADOS Y ANÁLISIS DE RESULTADOS

Respecto a los métodos expuestos en el apartado anterior, podemos concluir que el método puramente teórico para comprobar que los *fitness* o dificultades se corresponden

a los requeridos no nos proporciona mucha información sobre lo bueno o malo (fácil o difícil) que es en realidad dicho nivel. Sin embargo este método nos sirve para comprobar que efectivamente se cumplen los requisitos de generación definidos anteriormente.

El método de prueba manual es bastante efectivo, aunque por desgracia es subjetivo. Se hizo la prueba mostrando tres niveles de dificultades distintas a varias personas y cuando se preguntó cómo de difícil les había parecido el nivel, en el caso de dificultad intermedia o más cada uno se quejaba de la dificultad de algún elemento del nivel, siendo las respuestas bastante heterogéneas.

Finalmente, con el tercer método se quería determinar si existe algún tipo de relación entre la dificultad asociada a un nivel en base a su contenido (enemigos, elementos del nivel y su distribución), que es en lo que se basa el genético para evaluar los niveles frente a la puntuación obtenida por un agente inteligente a la hora de completar un nivel (jugabilidad de dicho nivel). Tras ejecutar los scripts de creación de niveles y prueba de los mismos se obtuvieron los siguientes resultados medios:

Población	Fitness medio	Longitud media	Llanos	Colinas	Tuberías	Saltos	Cañones	Goompas	Tortugas Verdes	Tortugas Rojas	Spikeys	Flores
g050	51	152	8	8	4	7	8	0	1	3	1	3
g100	98	143	7	5	6	10	7	1	2	2	1	3
g150	150	124	14	5	6	5	5	1	3	3	0	4
g200	200	152	13	4	4	8	6	4	1	6	2	2
g250	250	148	12	8	5	5	5	2	3	3	0	0
g300	299	157	10	6	5	4	10	4	2	3	0	1
g350	350	162	6	9	5	8	7	1	1	1	2	3
g400	400	150	4	11	8	6	6	1	0	3	1	4
g450	450	151	4	10	9	7	5	1	1	3	1	3
g500	500	144	7	6	8	5	9	3	2	0	3	1
g550	550	152	4	10	6	6	9	3	1	1	5	4
g600	600	148	3	10	6	5	11	0	2	2	3	3
g650	649	130	6	6	5	7	11	3	1	0	3	2
g700	701	163	5	6	6	13	5	1	2	3	1	2
g750	749	142	3	8	9	6	9	1	0	1	1	6

Tabla 1. Resultados promedio para las poblaciones de 50 niveles generadas genéticamente

De la tabla anterior se pueden obtener dos representaciones gráficas, una que relaciona los elementos del nivel con el *fitness* promedio de cada población y otra que relaciona el tipo y número de enemigos con el valor del *fitness* promedio.

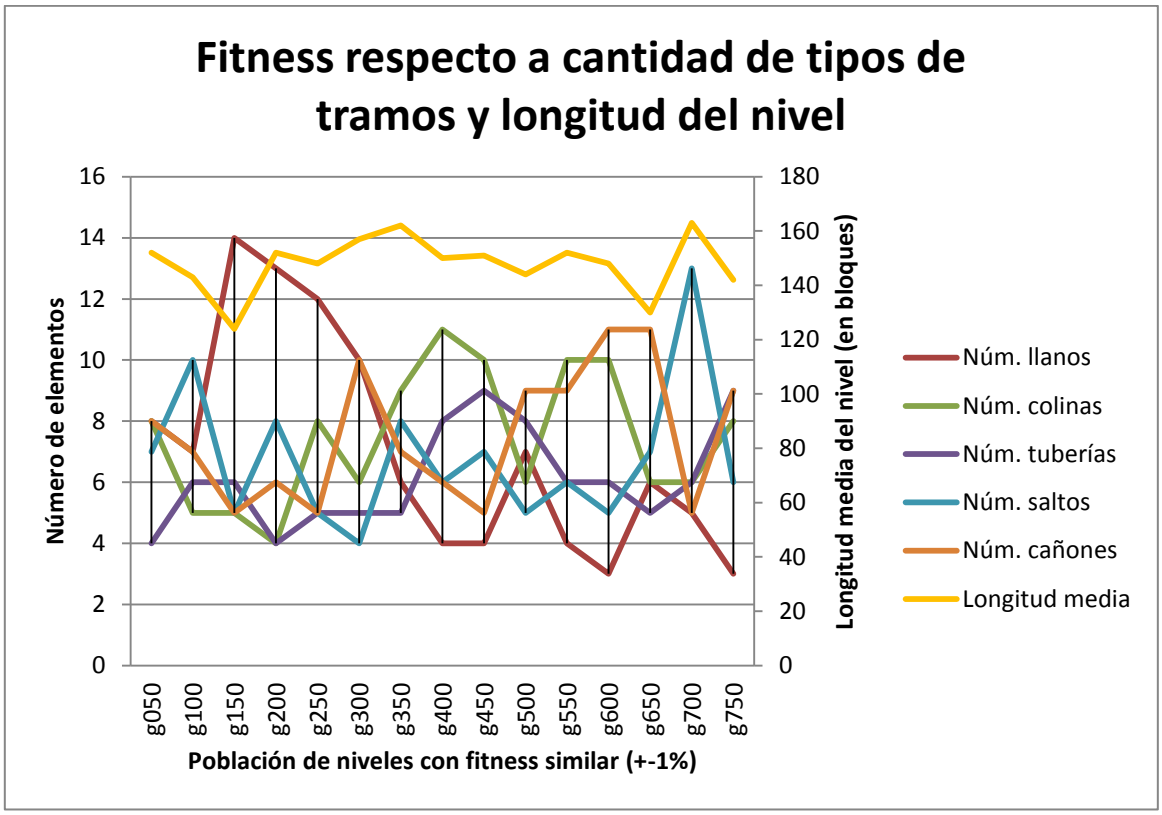


Figura 20 – Fitness respecto a cantidad de tipos de tramos y longitud del nivel (niveles genéticos)

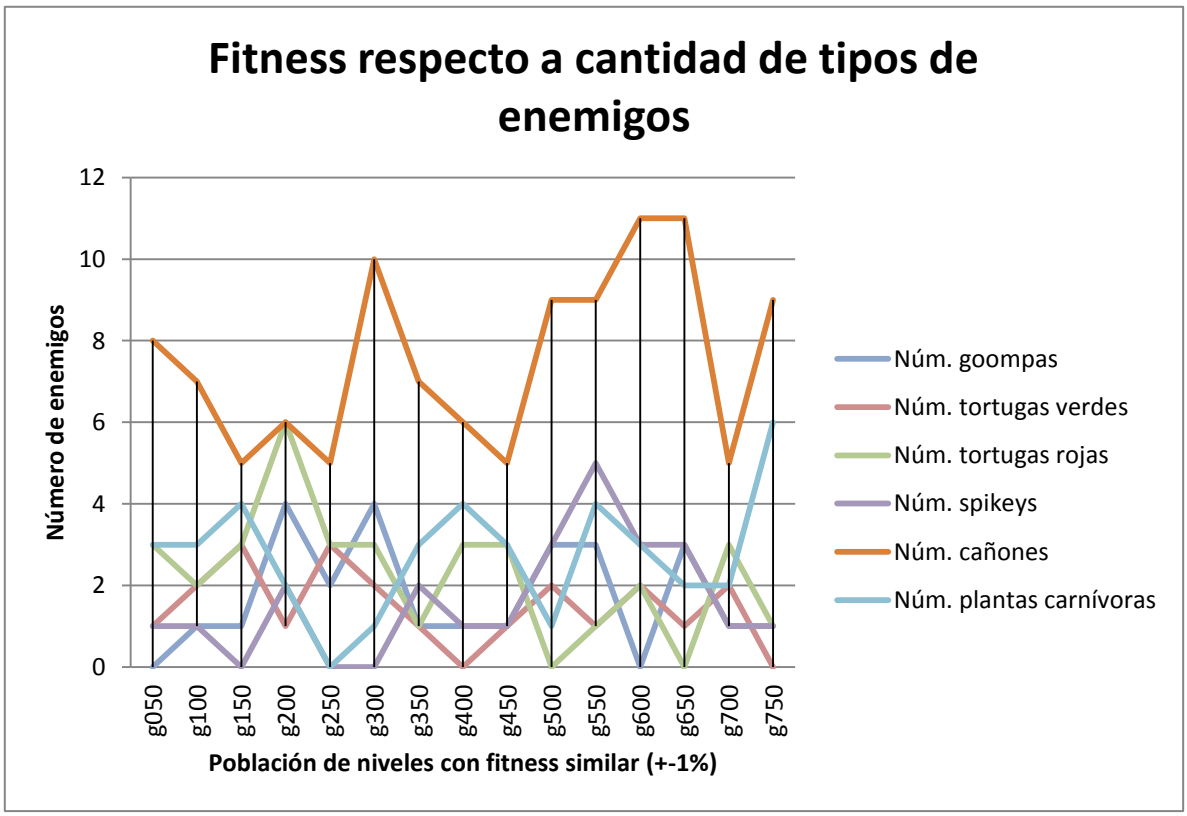


Figura 21 – Fitness respecto a cantidad de tipos de enemigos (niveles genéticos)

Los resultados obtenidos visibles en las gráficas nos hacen pensar que el criterio de evolución se ajusta bien al *fitness* (o al menos a los pesos otorgados a cada elemento). Se puede observar por ejemplo que en el caso de las poblaciones con *fitness* bajo, predominan niveles llanos (los más sencillos) y escasean saltos, cañones y tuberías (de mayor complejidad). A medida que aumenta la dificultad se puede observar cómo se reduce el número de tramos llanos y empiezan a aumentar principalmente los saltos (cuya dificultad crece exponencialmente en función de la longitud del hueco a saltar) y los cañones (que de base tienen una dificultad alta). Respecto a los enemigos, se puede observar un ligero balanceo determinado por el número de cañones. Si hay presencia de muchos cañones y el *fitness* es bajo, se reduce el número de enemigos complicados (*spikeys*) aunque pueda haber presencia de enemigos más sencillos (goompas y tortugas). En niveles con *fitness* más elevado, se aprecia que hay un gran número de cañones al igual que de enemigos complicados (g600, g650, g750), como *spikeys* y plantas carnívoras.

Población generada	Fitness genético	Fitness evaluación
g050	51	5540
g100	98	5549
g150	150	5317
g200	200	5546
g250	250	5609
g300	299	5742
g350	350	5806
g400	400	5732
g450	450	4957
g500	500	5641
g550	550	5595
g600	600	5066
g650	649	5290
g700	701	5619
g750	749	5476

Tabla 2. Comparativa entre el fitness del algoritmo genético y el fitness de la evaluación

Si comparamos los valores de la tabla anterior, podemos apreciar que aparentemente no hay correlación entre la dificultad teórica del nivel en base a su contenido y en base a la jugabilidad. Lo que se puede concluir es que la dificultad a nivel práctico (a la hora de jugar un nivel) posiblemente no tenga que ver tanto con los enemigos y los elementos del nivel, sino más con su distribución y la de posibles trampas que los hagan difíciles de jugar. Sólo se aprecia un descenso razonable del *fitness* en la población de niveles con *fitness* genético 450, pero a nivel global no es relevante, ya que hay poblaciones posteriores con un nivel evaluado medio superior (lo que implica que *Mario* obtuvo más *power-ups*, tardó menos en completar el nivel, eliminó más enemigos, etc. consiguiendo una mayor puntuación). Estos resultados se pueden apreciar mejor en la siguiente gráfica:

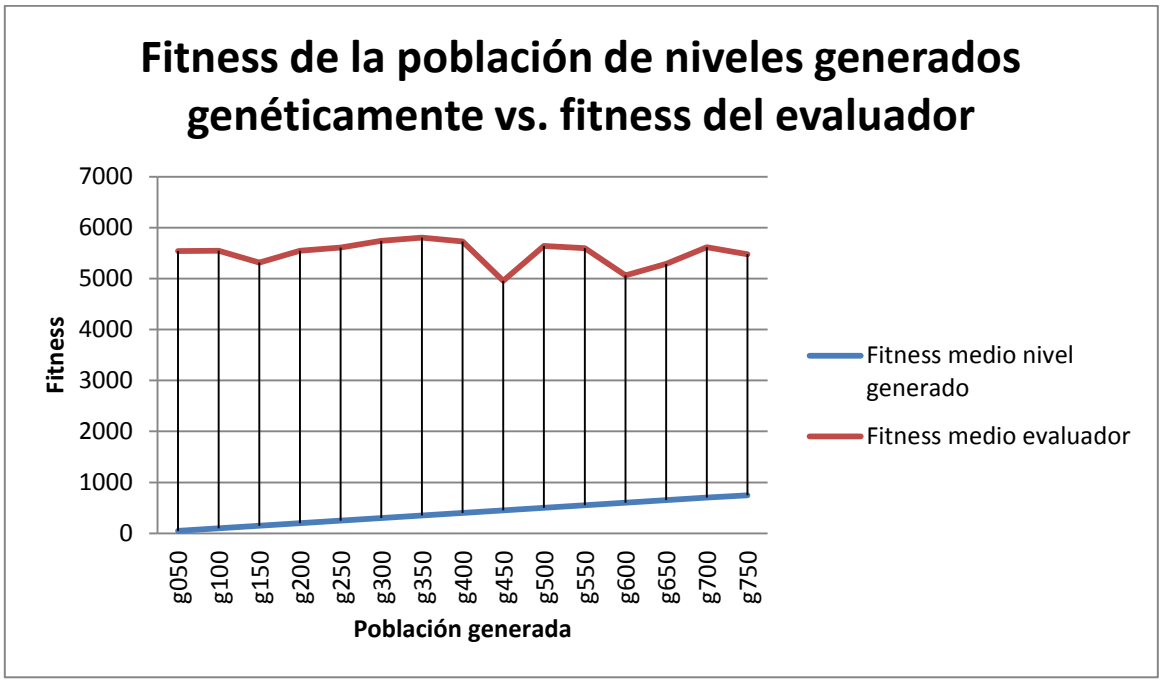


Figura 22 – Comparativa del fitness de los niveles respecto al fitness de la evaluación

Finalmente podemos analizar qué ha ocurrido en las evaluaciones.

Población	Fitness medio	Nivel completado	Tiempo medio de juego	Colisiones con criaturas	Criaturas eliminadas
g050	5540	Sí	38	2	7
g100	5549	Sí	59	2	12
g150	5317	Sí	37	2	10
g200	5546	Sí	46	2	8
g250	5609	Sí	52	1	11
g300	5742	Sí	58	2	11
g350	5806	Sí	37	0	9
g400	5732	Sí	37	1	11
g450	4957	Sí	122	2	10
g500	5641	Sí	54	2	13
g550	5595	Sí	51	2	11
g600	5066	Sí	57	2	3
g650	5290	Sí	73	2	15
g700	5619	Sí	42	2	7
g750	5476	Sí	67	2	14

Tabla 3. Información estadística media relativa a la evaluación de los niveles de las distintas poblaciones

Que visto gráficamente:

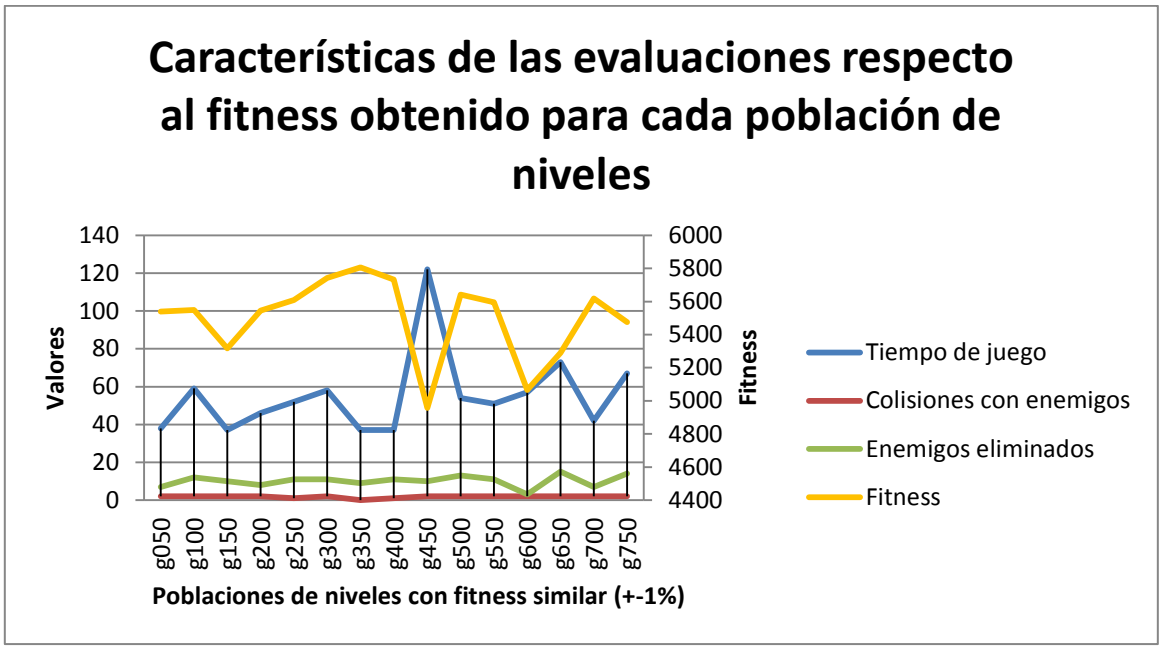


Figura 23 – Características de las evaluaciones respecto al fitness obtenido para cada población de niveles

Por lo que se puede observar en la gráfica, los factores que más afectan a la obtención de una baja puntuación son el tiempo que *Mario* tarda en completar el nivel y el número de enemigos eliminados. Así pues, un nivel en el que haya un gran número de obstáculos que dificulten el avance, o enemigos difíciles de matar (como los *spikeys*) que ralenticen el avance del agente y reduzcan drásticamente el número de enemigos eliminados repercutirá en una baja puntuación y en consecuencia, en un indicio de que el nivel es más difícil de superar que los demás. Por el contrario, cuanto menor sea el tiempo empleado en completar el nivel y mayor el número de enemigos eliminados, mejor el *fitness* resultante. Otro factor que parece tener bastante que ver, aunque en la gráfica no se aprecie tanto (debido a su escaso rango de valores) es el número de colisiones con enemigos. Una partida perfecta (esto es, sin colisiones) tendrá mucho mayor *fitness* de base que otra partida en la que *Mario* se haya chocado con algún enemigo, aunque haya tardado más tiempo en superar el nivel.

4.4. MEJORA DE LA PROPUESTA RESPECTO A OTRAS POSIBLES ALTERNATIVAS

Como se ha podido comprobar tras el estudio, la propuesta genética consigue producir una cantidad razonablemente alta de niveles genéticamente diversos en un tiempo muy aceptable (del orden de segundos).

Aparentemente, la mejor solución consiste en generar niveles mediante el enfoque genético, ya que produce niveles de una dificultad acorde a su contenido aunque como inconveniente se puede dar el caso de niveles con diferente *fitness* acaben teniendo una dificultad similar a la hora de ser jugados (evaluados). También se ha podido comprobar cómo el número de enemigos y elementos que componen el nivel es mucho más variado

en los niveles generados genéticamente que en niveles de dificultad similar producidos mediante el generador aleatorio.

Igualmente, gracias a la evaluación mediante el agente automático de los niveles generados por el algoritmo genético, se puede tener una idea de la dificultad de la jugabilidad de cualquier nivel presentado.

4.5. CONCLUSIONES

En este capítulo, se han comentado los tres métodos propuestos para la evaluación de resultados. Por un lado, la evaluación teórica de la complejidad (básicamente comprobar que se cumplen las restricciones respecto al *fitness*/dificultad y/o diversidad genética de los niveles generados), por otro lado una evaluación personal subjetiva de tres niveles diferentes presentados a cada persona, y finalmente se ha realizado una comparativa entre el generador genético y los resultados de los análisis del agente automático empleado en el coevolutivo.

Para el primer método se comprobó mediante trazas del programa que los niveles presentaban características coherentes respecto a la forma en la que fueron generados. En el caso de la evaluación personal, cada persona dio su versión subjetiva, siendo bastante diferentes entre sí para los mismos niveles (cada persona tiene una percepción distinta de la dificultad de los elementos en base a su familiarización con el juego). Finalmente con el tercer método, se generaron distintas poblaciones de 50 niveles con distintos valores de *fitness* por población. Una vez generados, se evaluaron dichos niveles empleando el agente automático y se relacionaron los datos estadísticos de los niveles con los datos estadísticos de las evaluaciones. Se pudo observar cómo aparentemente el *fitness* del enfoque genético no está directamente relacionado con el del enfoque coevolutivo. Parece coherente ya que la forma en la que se evalúan los niveles es distinta (basada en contenido respecto a basada en experiencia de juego).

En el siguiente capítulo (quinto), se presentan los distintos aspectos económicos y legales que deberán ser tenidos en cuenta a la hora de afrontar este proyecto.

5. ASPECTOS ECONÓMICOS Y LEGALES

5.1. INTRODUCCIÓN

En este capítulo se explican los aspectos económicos y legales que afectan al proyecto. Para lograr este fin, se da comienzo con el presupuesto (punto clave en este apartado), se prosigue con la explicación del entorno socio-económico actual, sobre el que se han basado gran cantidad de las decisiones tomadas. Se finaliza con el marco regulador (legal) del proyecto, para tomar las medidas adecuadas en relación al mismo.

5.2. PRESUPUESTO

Como se puede apreciar en el capítulo siguiente (que aborda el tema de la planificación), la duración del proyecto ha sido de 98 días, con una media de trabajo de cuatro horas diarias (total: **404 horas**). A lo largo de este tiempo han surgido una serie de gastos que ha sido necesario cubrir. Las estimaciones sobre los mismos se exponen a continuación.

5.2.1. Personal

Para la realización del proyecto se ha necesitado cubrir las tareas específicas realizadas por un analista de sistemas, un arquitecto de *software*, un diseñador, un programador y un experto en calidad y pruebas. Además, ha sido necesario un tutor que ha realizado gran parte del trabajo de jefe de proyecto, proporcionando asistencia en los momentos necesarios.

A continuación, se muestra una estimación de los costes que generan cada uno de esos roles por hora de trabajo:

Personal	
Cargo	**Coste (por hora)**
Analista	25,00 €
Arquitecto	20,00 €
Diseñador	20,00 €
Gestión de calidad y pruebas	25,00 €
Jefe de proyecto	35,00 €
Programador	15,00 €

Tabla 4. Estimación de coste por cada tipo de trabajo / hora

Aplicando estos costes al desglose de horas mostrado en la siguiente tabla:

Actividades y recursos							
Actividad	Personal						Totales
	Jefe de proyecto	Analista	Arquitecto	Diseñador	Pruebas	Programador	
Motivación y objetivos	10	10	10	6	0	0	36
Planteamiento del problema	22	72	18	0	0	0	112
Análisis del estado del arte	18	58	16	0	0	0	92
Requisitos	2	7	1	0	0	0	10
Restricciones	2	7	1	0	0	0	10
Diseño de la solución técnica	6	0	8	38	0	72	124
Desarrollo del diseño inicial	1	0	2	23	0	30	56
Desarrollo de alternativas del diseño	1	0	2	8	0	25	36
Ventajas e inconvenientes de los diseños	1	0	1	5	0	13	20
Elección y mejora del diseño final	3	0	3	2	0	4	12
Resultados y evaluación	8	10	0	8	30	0	56
Explicación del método de evaluación	2	6	0	1	11	0	20
Comparativa de los distintos métodos empleados y análisis de resultados	2	3	0	4	11	0	20
Mejora de la propuesta respecto a otras alternativas existentes	4	1	0	3	8	0	16

Actividades y recursos							
Actividad	**Personal**						**Totales**
	Jefe de proyecto	**Analista**	**Arquitecto**	**Diseñador**	**Pruebas**	**Programador**	
Aspectos económicos y legales	7	13	0	0	0	0	20
Presupuesto	4	10	0	0	0	0	14
Entorno socio-económico	2	2	0	0	0	0	4
Marco regulador	1	1	0	0	0	0	2
Planificación del trabajo	18	6	2	2	0	0	28
Planificación inicial	8	2	1	1	0	0	12
Planificación final	5	2	1	0	0	0	8
Comparativa del trabajo estimado y realizado	5	2	0	1	0	0	8
Conclusiones	2	2	1	2	1	0	8
Revisión del proyecto	14	6	0	0	0	0	20
Revisión del contenido	8	4	0	0	0	0	12
Revisión de la organización y estructura	3	1	0	0	0	0	4
Revisión de la presentación	3	1	0	0	0	0	4
Horas totales	**87**	**119**	**39**	**56**	**31**	**72**	**404**
Coste	**3.045,00 €**	**2.975,00 €**	**780,00 €**	**1.120,00 €**	**775,00 €**	**1.080,00 €**	**9.775,00 €**

Tabla 5. Coste asociado a las actividades y recursos empleados

El coste derivado del personal asciende a la cantidad de 9.775 euros.

5.2.2. Material

El coste derivado del uso del material puede ser dividido en tres apartados:

- Dispositivos *hardware* empleados a lo largo del proyecto.
- Licencias de *software* necesarias para llevarlo a cabo.
- Materiales fungibles, entre los que podemos encontrar los discos vírgenes empleados para grabar información, memorias flash y material de oficina (impresiones, papel, bolígrafos...).

5.2.2.1. Equipos

Para estimar el coste de los equipos se ha realizado el cálculo de la amortización de su precio respecto al tiempo que han sido usados en base al coste supuesto en el momento de adquisición. Se ha estimado una vida útil de unos tres años.

Como no tiene sentido realizar una amortización en función del número de días, se aproximarán los meses trabajados a tres y medio (101 días ≈ 3,37 meses)

En la siguiente tabla se muestra el coste derivado del uso de los equipos:

Equipos					
Nombre	**Precio unitario**	**Unidades**	**Coste(ud.)/mes**	**Nº meses**	**Total**
PC Intel core i7, 12 GB DDR3	1.250,00 €	1	34,72 €	2.5	86,80 €
Portátil HP DV7 T6400	790,00 €	1	21,94 €	3.5	76,79 €
Router Comtrend ct 536	30,00 €	1	0,83 €	2	1,66 €
Total					**165,25 €**

Tabla 6. Coste asociado al equipamiento hardware empleado

El coste derivado del uso de equipos asciende a la cantidad de 165,25 euros.

5.2.2.2. Software

Se ha intentado abaratar al máximo en cuanto a coste por *software* se refiere. Para ello se han empleado el máximo posible de aplicaciones gratuitas bajo *Windows*. Los costes por licencia del *software* más significativo se desglosan en la siguiente tabla:

Software			
Concepto	**Licencias**	**Coste/licencia**	**Coste**
Microsoft Windows 7 Professional **(*)**	2	0,00 €	0,00 €
Microsoft Project 2010 **(*)**	2	0,00 €	0,00 €
Microsoft Office 2010	2	155,00 €	310,00 €
NetBeans IDE	1	0,00 €	0,00 €
Eclipse IDE	1	0,00 €	0,00 €
Total			**310,00 €**

Tabla 7. Coste asociado al software empleado

(*) Las licencias de estos productos Microsoft se obtuvieron gracias al acuerdo Microsoft Dreamspark entre Microsoft Corp. y la Universidad Carlos III de Madrid.

Las aplicaciones que no se muestran en la Tabla 7 se proporcionan bajo licencia gratuita de uso o han sido utilizadas en sus periodos de prueba.

El coste derivado del uso de *software* asciende a la cantidad de 310 euros.

5.2.2.3. Fungible

En este apartado se incluyen los costes por memorias flash, discos duros externos, y CD / DVD vírgenes utilizados para transportar datos, así como los costes originados por el uso de material de oficina. Esto es, impresiones, fotocopias, encuadernaciones, bolígrafos, etc.

En la siguiente tabla se muestra una estimación de este tipo de costes a lo largo del proyecto:

Material fungible			
Concepto	**Cantidad**	**Coste unitario**	**Total**
Memorias USB	1	5,90 €	5,90 €
Discos vírgenes *Lightscribe* (CD)	5	1,32 €	6,60 €
Disco duro portátil 500 GB	1	79,90 €	79,90 €
Material de oficina	1	30,00 €	30,00 €
Total			**122,40 €**

Tabla 8. Coste asociado al material fungible

El coste derivado del uso de material fungible asciende a la cantidad de 122,40 euros.

5.2.3. Transporte

Para acudir al lugar de trabajo se empleó un vehículo propio (Chevrolet Aveo 90 c.v.), se estima que el coste de carburante al mes es de unos 75 euros.

Transporte	
Tipo	**Coste**
Privado (vehículo propio)	262,50 €
Total	**262,50 €**

Tabla 9. Coste asociado al transporte durante el proyecto

El coste derivado del transporte a lo largo de los tres meses y medio de duración del proyecto asciende a la cantidad de 262,50 euros.

5.2.4. Costes indirectos

En esta categoría se incluyen los costes por alquiler del local completamente acondicionado para el trabajo, durante los tres meses y medio de duración del proyecto, el consumo de luz y agua, el servicio de limpieza, la red de telefonía y comunicaciones necesaria, así como un colchón económico para cubrir posibles bajas del personal y un seguro a todo riesgo.

Costes indirectos	
Concepto	**Coste**
Alquiler de local	700,00 €
Luz, agua	210,00 €
Servicio de limpieza	210,00 €
Teléfono fijo y red de comunicaciones	55,83 €
Cobertura por bajas **(*[1])**	977,50 €
Seguro a todo riesgo **(*[2])**	28,00 €
Total	**2.181,33 €**

Tabla 10. Costes indirectos

(*1) En la cobertura por bajas se aplica en caso de tener que contratar personal para cubrir bajas por enfermedad o incapacidad.

(*2) El seguro a todo riesgo incluye cobertura de personal y bienes materiales.

La cantidad económica asociada a los costes indirectos a lo largo del proyecto asciende a la cifra de 2.181,33 euros.

5.2.5. Resumen de costes

Calculando la suma económica necesaria para cubrir los costes derivados de los apartados anteriores, se necesitan 73.527,60 euros para realizar el proyecto. Sin embargo, esta cantidad no incluye el impuesto sobre valor añadido ni los márgenes de beneficio y riesgo asociados a este tipo de proyectos.

Resumen de costes	
Concepto	**Cantidad**
Mano de obra	9.775,00 €
Equipos	165,25 €
Software	310,00 €
Fungible	122,40 €
Transporte	262,50 €
Costes indirectos	2.181,33 €
Total	**12.816,48 €**

Tabla 11. Resumen de costes asociados al proyecto

5.2.6. Totales

Llegados a este punto se puede obtener el precio final necesario para llevar a cabo este proyecto. Se muestra en la siguiente tabla:

Presupuesto del PFC	
Concepto	**Cantidad**
Coste total	12.816,48 €
Riesgo (5% del coste total estimado)	640,82 €
Beneficio (20% del coste total estimado)	2.563,30 €
Total sin I.V.A.	**16.020,60 €**
I.V.A. (18%)	2.883,71 €
Total	**18.904,31 €**

Tabla 12. Presupuesto del TFG

Teniendo en cuenta los costes desglosados en los apartados anteriores las ganancias del proyecto ascienden a un total de 2.563,30 €.

El presupuesto total de este proyecto asciende a la cantidad de:

18.904,31 €

Dieciocho mil novecientos cuatro euros con treinta y un céntimos.

Leganés, a 7 de Junio de 2012

La ingeniera proyectista

Fdo. Beatriz Puerta Hoyas

5.3. ENTORNO SOCIO-ECONÓMICO

Actualmente, la situación existente en gran parte del mundo es de recesión económica y, como es de esperar, esto tiene consecuencias negativas para muchos sectores, entre ellos, el que es interesante para el proyecto: la industria de los videojuegos.

Aunque los orígenes de los videojuegos son bastante recientes: en 1958 William Higinbotham desarrolló el primer videojuego (*Tennis for two*) en un osciloscopio [18]. Poco a poco, gracias a ese paso, comenzó a surgir una industria que se dedicaba a crear y comercializar videojuegos, siendo cada vez estos más complejos gracias al aumento de la capacidad de cómputo de las máquinas en las que se ejecutaban.

En la situación presente, se puede observar el aumento de empresas relacionadas con este campo y que cada vez el producto es más variado para llegar a los gustos de más tipos de personas, ofreciéndoles variedad de elección. Es por esto, que se buscan utilizar nuevas técnicas para innovar y atraer a una mayor cantidad de público, puesto que la industria del entretenimiento es una de las más rentables (siempre se invierte en cine, música…). Gracias a este objetivo, una técnica que se está empezando a emplear desde hace algunos años es la de distintos tipos de ***AI* en videojuegos**, puesto que se podrían generar enemigos con un comportamiento más complejo, agentes con una cierta libertad de decisión y niveles que variasen en función de distintos parámetros (la pericia del jugador, ciertas características aleatorias, el nivel de dificultad…).

Puesto que en los juegos de estrategia sí que se ha empleado en algunas ocasiones, en los juegos de plataforma (como *Infinite Mario Bros*) todavía no se ha desarrollado una *AI* lo suficientemente eficaz y eficiente como para poder resultar rentable en el aspecto comercial.

También, hay que tener en cuenta un importante concepto para las empresas pertenecientes a esta industria del videojuego: ¿Hasta qué punto interesa realizar una *AI* que sea capaz de generar distintos tipos de niveles? O lo que es lo mismo, si se vende un juego con capacidad de volver a ser jugado una vez finalizado (gracias a la *AI* que tiene implementado que le permite crear nuevas situaciones) ¿evitaría eso vender otros juegos porque el usuario seguiría jugando al mismo? ¿Supondría esto una **pérdida económica** para las empresas?

Por todas estas preguntas, las empresas involucradas en este sector son muy cuidadosas con sus innovaciones, puesto que sigue siendo más **rentable** para ellas que el usuario juegue una única vez al juego del principio al final y luego se compre otro, y no que una vez finalizada la primera partida puedo seguir jugando al mismo.

Esto no quiere decir que se deba abandonar esta idea, sino todo lo contrario: cuando se genere una *AI* lo suficientemente **sencilla** de desarrollar y que sea **atractiva** para el usuario sin llegar a poder producir pérdidas para las empresas, entonces se habrá alcanzado otro hito más en el **progreso de este sector**.

5.4. MARCO REGULADOR

En este proyecto se ha tenido en cuenta tanto el **marco regulador técnico** como el legal; el primero se encuentra en el capítulo 2 (apartados 2.3 y 2.4 en los que se desarrollan los requisitos y las restricciones gracias al análisis presentado para que la solución propuesta sea la más adecuada).

En cuanto al segundo (el **marco regulador legal**) es el que se expone en dicho capítulo al estar estrechamente relacionado con los aspectos económicos y legales que se desarrollan en el proyecto.

La única normativa legal que pudiera existir en el desarrollo de este proyecto es por la procedencia del banco de pruebas *Mario AI*. No obstante, como se puede apreciar en los documentos referenciados, especialmente en los que explican las normas de utilización del mismo [11] [12], este banco de pruebas (*software*) puede ser usado libremente para motivos de investigación o enseñanza. Es por ello que se puede hacer un desarrollo de algoritmos sobre el mismo sin restricciones.

Sin embargo, como la autoría de la competición y el código fuente es de la propiedad de tres autores (J. Togelius, S. Karakovskiy y N. Shaker), todo desarrollo producido sobre la herramienta que quiera ser publicado es adecuado contar con la participación de estos tres autores, especialmente, si se participa en la competición propuesta [1]. En ese caso, aquellos generadores de niveles que hayan sido elegidos para participar en la competición (de entre todos los presentados), participarán también en la redacción de un artículo de investigación con los tres creadores del juego y los demás competidores.

En cuanto al resto de herramientas utilizadas o desarrolladas, no hay restricciones legales puesto que se deban tener en cuenta. El *hardware* es de uso personal y el resto de las herramientas *software* tienen las licencias correspondientes. No existirían más términos legales destacables para presentar este proyecto.

5.5. CONCLUSIONES

En este capítulo, se pretende aclarar todos los posibles **conceptos económicos y legales** (especialmente los primeros), para que este proyecto sea viable y factible, y evitar con ello que el proyecto se quedase en una **mera idea** sin desarrollar.

Por este motivo, se han tratado temas como: el desarrollo de un **presupuesto** ajustado (uno de los puntos más importantes de este capítulo), el estudio del entorno socio-económico (para la comprensión de la situación actual y aseguración de que el tema elegido genera productos vendibles en el mercado) y la reflexión del marco regulador (en este caso, legal), que indica las pautas y normas a seguir para que el proyecto se desarrolle sin percances de tipo legal.

En el siguiente capítulo (sexto), se podrá reflexionar acerca de la evolución temporal del proyecto; comparando para ello la planificación inicial con respecto a la final. Por tanto, previamente a este paso, se desarrollarán con detalle ambas **planificaciones**.

6. PLANIFICACIÓN DEL TRABAJO

6.1. INTRODUCCIÓN

En este capítulo se muestra la planificación realizada del proyecto (explicándose para ello tanto la **planificación inicial** como la **planificación final**), así como el método que se sigue para realizar el seguimiento del proyecto y evitar desviaciones. Se realiza una **comparación** entre ambas planificaciones (estimada y real).

Para elaborar la planificación del proyecto se ha utilizado la herramienta *Microsoft Project 2010*; el diagrama de Gantt ha sido generado mediante dicha herramienta

6.2. PLANIFICACIÓN INICIAL

6.2.1. Cronograma de actividades y control para la planificación inicial

Para un correcto seguimiento y control del proyecto se convocarán reuniones de manera asidua, fijándose el número óptimo de las mismas en una por semana, para comprobar el correcto avance del proyecto, así como de que se estén alcanzando los objetivos fijados.

El control sobre el proyecto se realizará mediante el siguiente cronograma que muestra las distintas etapas del proyecto, con sus respectivas actividades y el tiempo que deberá ocupar cada una de estas.

Puesto que en un principio la entrega de artículos de investigación para participar el *The 2012 Mario AI Championship: Level Generation Track* finalizaba el 15 de mayo de 2012, se intentó ajustar lo más posible el tiempo del que se disponía para el proyecto poder realizar el mencionado artículo y enviarlo a la competición. Puesto que si no te tenía finalizado el proyecto, no se habría llegado a las conclusiones buscadas, y por tanto, no tendría sentido escribir el artículo. Al ser esta una actividad "*extra*" no aparece en la tabla de tareas.

Id	Actividad	Duración de la tarea (Inicio - Fin)	Tiempo estimado dedicado	Inicio	Fin
1	**Motivación y objetivos**	**13 días**	**9 días**	**08/02/12**	**20/02/12**
2	**Planteamiento del problema**	**36 días**	**26 días**	**21/02/12**	**27/03/12**
3	Análisis del estado del arte	30 días	22 días	21/02/12	21/03/12
4	Requisitos	6 días	2 días	22/03/12	27/03/12
5	Restricciones	6 días	2 días	22/03/12	27/03/12
6	**Diseño de la solución técnica**	**30 días**	**22 días**	**28/03/12**	**26/04/12**
7	Desarrollo del diseño inicial	16 días	12 días	28/03/12	12/04/12
8	Desarrollo de alternativas del diseño	8 días	6 días	13/04/12	20/04/12
9	Ventajas e inconvenientes de los diseños	4 días	2 días	23/04/12	26/04/12
10	Elección y mejora del diseño final	4 días	2 días	23/04/12	26/04/12
11	**Resultados y evaluación**	**16 días**	**12 días**	**18/04/12**	**03/05/12**
12	Explicación del método de evaluación	7 días	5 días	18/04/12	24/04/12
13	Comparativa de los distintos métodos empleados y análisis de resultados	6 días	4 días	25/04/12	30/04/12
14	Mejora de la propuesta respecto a otras alternativas existentes	3 días	3 días	01/05/12	03/05/12
15	**Aspectos económicos y legales**	**5 días**	**3 días**	**04/05/12**	**08/05/12**
16	Presupuesto	4 días	1,5 día	04/05/12	07/05/12
17	Entorno socio-económico	2 días	1 día	07/05/12	08/05/12
18	Marco regulador	1 día	0,5 días	08/05/12	08/05/12
19	**Planificación del trabajo**	**55 días**	**6 días**	**16/03/12**	**10/05/12**
20	Planificación inicial	5 días	3 días	16/03/12	20/03/12
21	Planificación final	4 días	1,5 días	07/05/12	10/05/12

Id	Actividad	Duración de la tarea (Inicio - Fin)	Tiempo estimado dedicado	Inicio	Fin
22	Comparativa del trabajo estimado y realizado	4 días	1,5 días	07/05/12	10/05/12
23	**Conclusiones**	**2 día**	**1 día**	**09/05/12**	**10/05/12**
24	**Revisión del proyecto**	**4 días**	**2 días**	**11/05/12**	**14/05/12**
25	Revisión del contenido	1 día	1 día	11/05/12	11/05/12
26	Revisión de la organización y estructura	1 día	0,5 días	14/05/12	14/05/12
27	Revisión de la presentación	1 día	0,5 días	14/05/12	14/05/12
28	**Entrega del proyecto**	**0 días**	**0 días**	**15/06/12**	**15/06/12**
	TOTAL DE TIEMPO INVERTIDO	**129 días**	**81 días**	**08/02/12**	**15/06/12**

Tabla 13. Desarrollo de la planificación inicial

(*1) El sumatorio de los días por tareas se hace en función del tiempo trabajado en una subtarea. Si dos o más subtareas comparten el mismo día, se les asigna a cada una medidas de tiempo inferior a un día, de tal forma que la suma del tiempo invertido sea la correcta.

(*2) Como el sumatorio total de tiempo invertido conllevará que distintas tareas puedan superponerse espacialmente y dado que como máximo en un mismo día se pueden llegar a superponer dos tareas, se sumarán los días totales y, en cuanto al horario de trabajo estipulado para ese día, se dedicará el doble de tiempo al proyecto para cumplir con el cronograma establecido. Es decir, si dos tareas coinciden en el mismo día, se hacen las dos aunque lleve el doble de tiempo que el dedicado en un día "normal".

Se puede extraer como conclusión lo siguiente: Se estima que el proyecto durará 97 días, y se aproxima que **81 días** se habrán dedicado al trabajo en el proyecto. Se estima una media de cuatro horas por día.

6.2.2. Planificación inicial (Gantt)

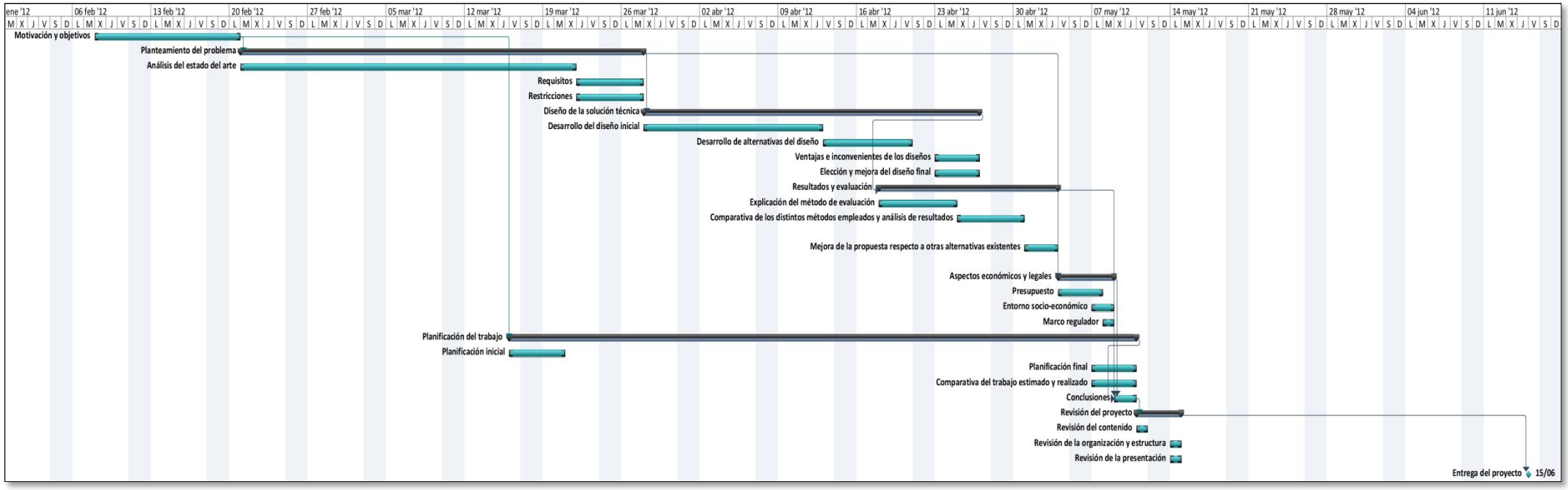


Figura 24 – Planificación final expuesta mediante un diagrama de Gantt

6.3. PLANIFICACIÓN FINAL

6.3.1. Cronograma de actividades y control para la planificación final

Para un correcto seguimiento y control del proyecto, por lo general, se convocaron reuniones de manera asidua, procurando realizarse una por semana o una por cada dos semanas. De esta forma, se pudo realizar un seguimiento sobre el correcto avance del proyecto, así como si se estaban alcanzando los objetivos fijados.

El control sobre el proyecto se realizó mediante el siguiente cronograma que muestra las distintas etapas del proyecto, con sus respectivas actividades y el tiempo que necesitó cada una de estas.

Debido a que fue prolongado el plazo de participación en el *The 2012 Mario AI Championship: Level Generation Track* y se podía realizar la entrega de artículos hasta el 25 de agosto de 2012, se alargó la duración de las tareas para el proyecto con el objetivo de poderle dedicar un mayor tiempo. La única fecha crítica que se debía tener en cuenta es la entrega del trabajo, como máximo, el día 15 de junio de 2012.

Id	Actividad	Duración de la tarea (Inicio - Fin)	Tiempo real dedicado	Inicio	Fin
1	**Motivación y objetivos**	**13 días**	**9 días**	**08/02/12**	**20/02/12**
2	**Planteamiento del problema**	**36 días**	**28 días**	**21/02/12**	**29/03/12**
3	Análisis del estado del arte	31 días	23 días	21/02/12	22/03/12
4	Requisitos	6 días	2,5 días	23/03/12	29/03/12
5	Restricciones	6 días	2,5 días	23/03/12	29/03/12
6	**Diseño de la solución técnica**	**42 días**	**31 días**	**30/03/12**	**11/05/12**
7	Desarrollo del diseño inicial	20 días	14 días	30/03/12	18/04/12
8	Desarrollo de alternativas del diseño	13 días	9 días	19/04/12	01/05/12
9	Ventajas e inconvenientes de los diseños	10 días	5 días	02/05/12	11/05/12
10	Elección y mejora del diseño final	8 días	3 días	04/05/12	11/05/12
11	**Resultados y evaluación**	**18 días**	**14 días**	**14/05/12**	**31/05/12**
12	Explicación del método de evaluación	5 días	5 días	14/05/12	18/05/12
13	Comparativa de los distintos métodos empleados y análisis de resultados	11 días	5 días	21/05/12	31/05/12

Id	Actividad	Duración de la tarea (Inicio - Fin)	Tiempo real dedicado	Inicio	Fin
14	Mejora de la propuesta respecto a otras alternativas existentes	11 días	4 días	21/05/12	31/05/12
15	**Aspectos económicos y legales**	**5 días**	**5 días**	**01/06/12**	**07/06/12**
16	Presupuesto	7 días	3,5 días	04/06/12	07/06/12
17	Entorno socio-económico	1 días	1 día	01/06/12	01/06/12
18	Marco regulador	1 día	0,5 días	04/06/12	04/06/12
19	**Planificación del trabajo**	**55 días**	**7 días**	**16/03/12**	**07/06/12**
20	Planificación inicial	5 días	3 días	16/03/12	20/03/12
21	Planificación final	4 días	2 días	04/06/12	07/06/12
22	Comparativa del trabajo estimado y realizado	4 días	2 días	04/06/12	07/06/12
23	**Conclusiones**	**2 días**	**2 días**	**05/06/12**	**07/06/12**
24	**Revisión del proyecto**	**7 días**	**5 días**	**08/06/12**	**14/06/12**
25	Revisión del contenido	3 días	3 días	08/06/12	12/06/12
26	Revisión de la organización y estructura	2 días	1 día	13/06/12	14/06/12
27	Revisión de la presentación	2 días	1 día	13/06/12	14/06/12
28	**Entrega del proyecto**	**0 días**	**0 días**	**15/06/12**	**15/06/12**
	TOTAL DE TIEMPO INVERTIDO	**129 días**	**101 días**	**08/02/12**	**15/06/12**

Tabla 14. Desarrollo de la planificación final

(*1) El sumatorio de los días por tareas se hace en función del tiempo trabajado en una subtarea. Si dos o más subtareas comparten el mismo día, se les asigna a cada una medidas de tiempo inferior a un día, de tal forma que la suma del tiempo invertido sea la correcta.

(*2) Como el sumatorio total de tiempo invertido conllevará que distintas tareas puedan superponerse espacialmente y dado que como máximo en un mismo día se pueden llegar a superponer dos tareas, se sumarán los días totales y, en cuanto al horario de trabajo estipulado para ese día, se dedicará el doble de tiempo al proyecto para cumplir con el cronograma establecido.

Se puede extraer como conclusión lo siguiente: Se estima que el proyecto durará 178 días, y se aproxima que **101 días** se habrán dedicado al trabajo en el proyecto. Se ha trabajado aproximadamente unas cuatro horas por día (de media).

6.3.2. Planificación final (Gantt)

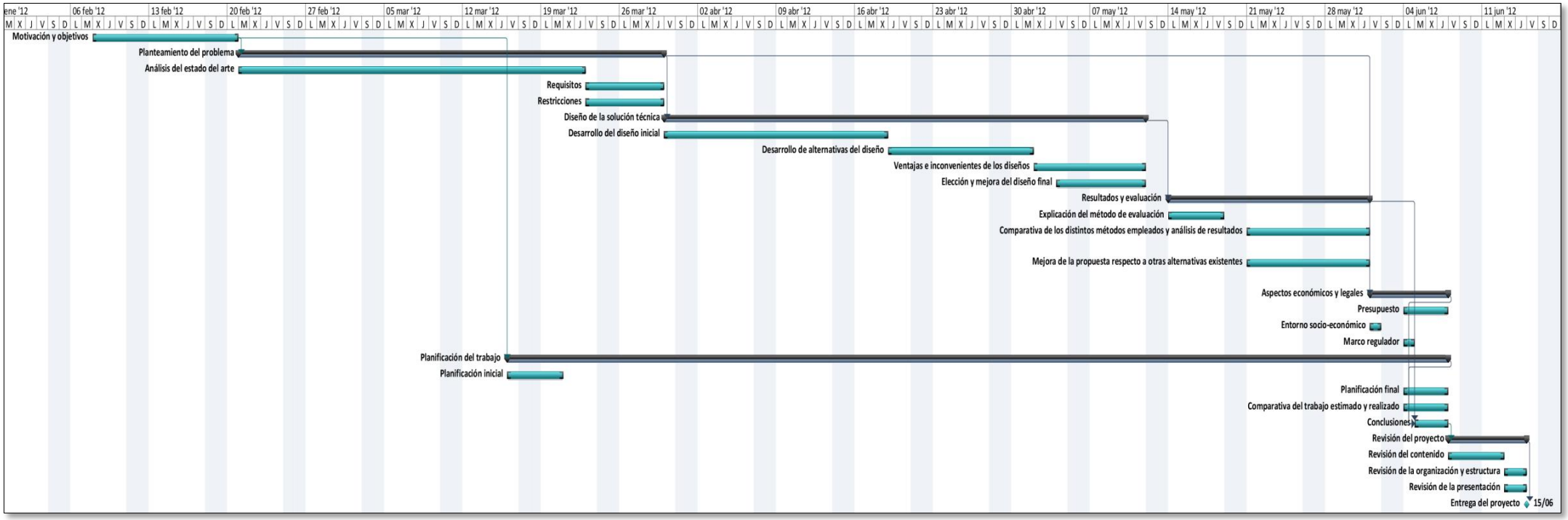


Figura 25 – Planificación final expuesta mediante un diagrama de Gantt

6.4. COMPARATIVA DEL TRABAJO ESTIMADO Y REALIZADO

Se puede observar que la diferencia fundamental entre ambas planificaciones (inicial y final) radica en la ampliación del tiempo de trabajo durante un mes (se estimó haber finalizado el proyecto para el 14 de mayo de 2012 pero esta acción se realizó el 14 de junio de 2012 –un mes más tarde–). Esto no se debe a una mala planificación, sino a que (como se razonó anteriormente) se decidió alargar la duración del proyecto cuando se tuvo constancia de la extensión de plazo para la presentación del mismo en la *Level Generation Competition* (parte de la *IEEE Computational Intelligence and AI in Games*, patrocinada por *2010 Mario AI Championship*).

Puesto que **el plazo se había incrementado** en más de tres meses (el 15 de mayo de 2012 era la fecha límite inicial para la presentación de artículos), se decidió aumentar el tiempo dedicado al proyecto para así obtener mejores resultados y más avances sobre el mismo (diseñar otras alternativas como solución, ajustar mejor los valores para llegar a conocer mejores resultados, etc.).

Por tanto, se aprovechó prácticamente todo el tiempo del que se disponía hasta la entrega del proyecto para realizar más avances de los estimados y efectuar una investigación más exhaustiva sobre el tema aquí expuesto.

No existe ninguna diferencia significativa entre ambas planificaciones además de la ya descrita. Sólo aumentó el tiempo dedicado a las tareas de una forma proporcional al tiempo de la nueva cantidad de tiempo disponible. Tampoco se debe olvidar que al tomar esta decisión, fue necesario encontrar un equilibrio entre la cantidad de trabajo que se puede realizar en el proyecto y el tiempo que es necesario para cumplir con éxito ese trabajo planteado. Sin embargo, este hecho influyó en el coste del proyecto, puesto que al haber invertido un mayor número de horas en el mismo, el coste de personal es mayor (el presupuesto resultante es más alto) que si no se hubiera tomado esta decisión.

6.5. CONCLUSIONES

En este capítulo se ha desarrollado la **planificación**, mostrándose tanto la inicial como la final (destacando en ambas las tareas críticas mediante los diagramas de Gantt, marcando hitos y estructurando una clara secuenciación y temporización de las tareas).

Con el desarrollo de la planificación inicial y de la final, se busca comprender la **evolución** que ha sufrido el proyecto a lo largo del tiempo, y descubrir si esta planificación ha sido correcta para poder terminar exitosamente un proyecto de esta envergadura con el tiempo necesario.

Además, se han esclarecido las posibles diferencias existentes entre la planificación inicial y la final gracias a una **comparativa** en la que, a su vez, se justifican estas diferencias.

En el próximo capítulo (el séptimo), se desvelarán las **conclusiones** alcanzadas, mostrando los problemas encontrados más relevantes durante el transcurso de la búsqueda de la solución y, lo más destacable de este capítulo, será ver qué objetivos se han cumplido y cuáles son las líneas de trabajo futuras que surgen a partir de estos.

7. CONCLUSIONES

7.1. INTRODUCCIÓN

En este capítulo se explican las conclusiones a las que se han llegado mediante la realización del proyecto. Se ha considerado conveniente dividir este capítulo en tres partes: **objetivos**, **problemas** y **líneas futuras** a fin de conseguir una visión más general sobre lo que se ha logrado con este proyecto.

7.2. OBJETIVOS CUMPLIDOS

A lo largo del proyecto se han desarrollado una serie de alternativas para generar niveles de forma automática empleando enfoques evolutivos. Dichas alternativas se han comparado entre sí y con una primera aproximación basada en un enfoque aleatorio dirigido por reglas fijas. Se puede afirmar que los dos objetivos presentados en el punto 1.2 han sido cumplidos.

Como se ha visto anteriormente, el generador de niveles genético permite producir niveles distintos entre sí para una dificultad dada, con la ventaja principal respecto al aleatorio de que no hace monótono el número y tipo de elementos presentes en niveles a partir de una determinada dificultad. A efectos prácticos esto se traduce en **niveles más divertidos** que los producidos mediante el generador aleatorio.

Respecto al objetivo de conseguir grupos de niveles basados en la dificultad, se ha podido determinar que existen dos tipos de dificultad. Aquella determinada por los tipos de elementos, tipo de enemigos y número de estos últimos en el nivel, y aquella basada en la experiencia de juego. La ventaja principal del generador de niveles basado en coevolución es que al evaluar jugando los niveles, permite precisar la dificultad en base a parámetros relacionados con la experiencia de juego (tiempo empleado en completar el nivel, enemigos eliminados, *power-ups* conseguidos, etc.), así pues, podemos clasificar los niveles generados en base a dos **tipos distintos de dificultad** (**contenido** y **experiencia de juego**) que no tienen por qué estar relacionados entre ellos.

7.3. PROBLEMAS ENCONTRADOS

Se pueden clasificar en dos tipos el número de problemas encontrados. **Problemas relacionados con el entorno** y **problemas relacionados con la solución**.

En los problemas relacionados con el entorno podemos citar por ejemplo la gran cantidad de tiempo empleado en elaborar un buen estado del arte y buscar información relacionada con el proyecto, que por otra parte ha permitido enfrentarse directamente a la solución, ahorrando tiempo al no tener que repetir trabajo que ya estaba hecho (utilizar un agente inteligente previamente implementado y probado en lugar de tener que implementar uno nuevo desde cero). También cabe destacar el inmenso esfuerzo dedicado a estudiar y comprender el código de la plataforma *Mario AI*, así como la del agente inteligente empleado para la estrategia coevolutiva, y el esfuerzo de adaptar

dicho código para convertirlo en una librería de evaluación reutilizable y usable en nuestro proyecto.

Respecto a los problemas relacionados con la solución, se puede citar el haber encontrado niveles de una determinada dificultad generados mediante un enfoque genético que sin embargo no eran resolubles (por ejemplo, por presentar saltos imposibles). Dichos niveles deberían tener asociada una complejidad infinita al no poder ser resueltos, pero como se trata de casos difíciles de evaluar incluso para una función global, se decidió que la mejor forma de encontrar solución a este problema era mediante la utilización de la coevolución (a la hora de evaluar el nivel, el agente automático no encontraría solución, penalizando al nivel en cuestión). Otro problema no menos grave es el gran coste computacional asociado al enfoque coevolutivo que conlleva una gran cantidad de tiempo necesario para la convergencia en una solución. Dicho problema quizá podría reducirse bastante (a nivel temporal) si se paralelizara convenientemente el código adaptándolo a plataformas *hardware* adecuadas, o adaptándolo a sistemas de computación distribuida, pero dichas mejoras quedan fuera del marco del presente proyecto.

7.4. LÍNEAS FUTURAS DE TRABAJO

Como se acaba de comentar, una posible mejora sería intentar reducir el tiempo de computación del enfoque coevolutivo mediante la paralelización. Igualmente otras posibles mejoras serían las correspondientes a aplicar distintos algoritmos de la *AI*, como la cuarta alternativa planteada (Aprendizaje Automático) y probar a emplear otros agentes de evaluación en lugar del agente *Mario* genético, ya que se ha dejado preparada la plataforma de evaluación para admitir el uso de cualquier otro agente automático de evaluación.

En una línea paralela, quizá para mejorar la usabilidad de la aplicación, sería conveniente desarrollar una interfaz gráfica para lanzar las distintas funcionalidades proporcionadas por la aplicación desde una ventana. Dicha interfaz gráfica podría ir integrada en el código del proyecto o bien comportarse como un lanzador que le pase parámetros al .jar correspondiente a la aplicación del proyecto.

7.5. CONCLUSIONES

En este capítulo se ha desarrollado las **conclusiones del proyecto**. Por tanto, para lograr unas conclusiones completas que reuniesen todo lo trabajado, ha sido conveniente separar estas conclusiones en tres apartados:

Objetivos cumplidos: Donde se puede observar que, efectivamente, se ha logrado el objetivo mencionado en el principio del proyecto.

Problemas encontrados: Donde se distingue qué contrariedades han aparecido a lo largo del proyecto; sin embargo, también se puede advertir que existen soluciones a dichos problemas.

Líneas futuras de trabajo: Donde se propone qué siguientes pasos se puede dar para mejorar el proyecto realizado y qué sería conveniente investigar haciendo llegado a este punto.

A partir de la finalización de este capítulo, se concluirá dicho proyecto con las referencias bibliográficas sobre las que se ha trabajado y con los cuatro anexos de los que requiere este proyecto para su correcto entendimiento.

REFERENCIAS BIBLIOGRÁFICAS

[1] J. Togelius, S. Karakovskiy y N. Shaker, «2012 Mario AI Championship,» 2012. [En línea]. Available: http://www.marioai.org. [Último acceso: Febrero 2012].

[2] H. Valero Capataz, «Computación evolutiva aplicada al desarrollo de videojuegos: Mario AI,» Madrid, Spain, 2011.

[3] T. W. Malone, «What makes things fun to learn?: Heuristics for designing instructional computer games,» de *ACM*, 1980.

[4] T. W. Malone, «Heuristics for Designing Enjoyable User Interfaces: Lessons from Computer Games,» de *ACM*, 1981.

[5] J. G. Hogle, «Considering Games as Cognitive Tools: In Search of Effective "Edutainment",» *ERIC Document Reproduction Service No. ED 425 737*, 1996.

[6] K. Compton y M. Mateas, «Procedural Level Design for Platform Games,» de *AAAI Artificial Intelligence and Interactive Digital Entertainment*, 2006.

[7] C. Pedersen, J. Togelius y G. N. Yannakakis, «Modeling player experience in Super Mario Bros,» de *IEEE Symposium on CIG*, 2009.

[8] G. Smith y J. Whitehead, «Analyzing the Expressive Range of a Level Generator,» de *PCGames*, Monterey, CA, USA, 2010.

[9] N. Shaker, G. Yannakakis y J. Togelius, «Towards Automatic Personalized Content Generation for Platform Games,» de *AAAI Artificial Intelligence and Interactive Digital Entertainment*, 2010.

[10] J. Togelius, E. Kastbjerg, D. Schedl y G. N. Yannakakis, «What is Procedural Content Generation? Mario on the borderline,» de *PCGames*, Bordeaux, France, 2011.

[11] N. Shaker, J. Togelius, G. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith y R. Baumgarten, «The 2010 Mario AI Championship: Level Generation Track,» *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, nº 4, pp. 332 - 347, Dec. 2011.

[12] S. Karakovskiy y J. Togelius, «The Mario AI Benchmark and Competitions,» *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, nº 1, pp. 55 - 67, March 2012.

[13] J. Togelius, S. Karakovskiy, J. Koutnik y J. Schmidhuber, «Super Mario Evolution,» *IEEE Symposium on CIG*, pp. 156 - 161, 2009.

[14] J. Togelius, S. Karakovskiy y R. Baumgarten, «The 2009 Mario AI Competition,» de *IEEE Evolutionary Computation*, Barcelona, Spain, 2010.

[15] N. Sorenson, P. Pasquier y S. DiPaola, «A Generic Approach to Challenge Modeling for the Procedural Creation of Video Game Levels,» *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, nº 3, pp. 229 - 244, Sept. 2011.

[16] C. H. Tan, K. C. Tan y A. Tay, «Dynamic Game Difficulty Scaling Using Adaptive Behavior-Based AI,» *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, nº 4, pp. 289 - 301, Dec. 2011.

[17] Universidad Carlos III de Madrid, «Tema 3: Análisis,» de *Asignatura de Diseño de Sistemas Interactivos (Grad. Ing. Inf.)*, Curso 2010-2011.

[18] Brookhaven National Laboratory - U.S. Department of Energy, «First Video Game?,» 2012. [En línea]. Available: http://www.bnl.gov/video/index.php?v=127. [Último acceso: Junio 2012].

[19] J. Togelius, M. Preuss y G. N. Yannakakis, «Towards multiobjective procedural map generation,» de *PCGames*, Monterey, CA, USA, 2010.

[20] L. Johnson, G. N. Yannakakis y J. Togelius, «Cellular automata for real-time generation of infinite cave levels,» de *PCGames*, Monterey, CA, USA, 2010.

[21] J. Whitehead, «Toward Procedural Decorative Ornamentation in Games,» de *PCGames*, Monterey, CA, USA, 2010.

[22] C. Pedersen, J. Togelius y G. Yannakakis, «Optimization of platform game levels for player experience,» de *AAAI Artificial Intelligence and Interactive Digital Entertainment*, 2009.

[23] J. Togelius y S. M. Lucas, «Arms races and car races,» de *Parallel Problem Solving from Nature*, 2006.

[24] J. Togelius, G. Yannakakis, K. O. Stanley y C. Browne, «Search-based Procedural Content Generation,» de *EvoApplications*, 2010.

[25] R. Koster, A Theory of Fun for Game Design, Scottsdale, Arizona, USA: Paraglyph Press, 2004.

[26] J. Togelius, R. De Nardi y S. Lucas, «Towards automatic personalised content creation for racing games,» *IEEE Symposium on CIG*, pp. 252-259, 2007.

[27] J. Togelius y J. Schmidhuber, «An experiment in automatic game design,» *IEEE Symposium on CIG*, pp. 111 - 118, 2008.

[28] G. Yannakakis y J. Hallam, «Real-Time Game Adaptation for Optimizing,» *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 1, nº 2, pp. 121 - 133, Jun. 2009.

[29] E. Hastings, R. Guha y K. Stanley, «Automatic Content Generation in the Galactic Arms Race Video Game,» *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 1, nº 4, pp. 245 - 263, Dec. 2009.

[30] C. Pedersen, J. Togelius y G. Yannakakis, «Modeling Player Experience for Content Creation,» *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 2, nº 1, pp. 54 - 67, Mar. 2010.

[31] J. Doran y I. Parberry, «Controlled Procedural Terrain Generation Using Software Agents,» *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 2, nº 2, pp. 111 - 119, Jun. 2010.

[32] D. Loiacono, P. Lanzi, J. Togelius, E. Onieva, D. Pelta, M. Butz, T. Lönneker, L. Cardamone, D. Perez, Y. Sáez, M. Preuss y J. Quadflieg, «The 2009 Simulated Car Racing Championship,» *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 2, nº 2, pp. 131 - 147, Jun. 2010.

[33] G. Smith, J. Whitehead, M. Mateas, M. Treanor, J. March y M. Cha, «Launchpad: A Rhythm-Based Level Generator for 2-D Platformers,» *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 3, nº 1, pp. 1 - 16, Mar. 2011.

[34] G. Yannakakis y J. Togelius, «Experience-Driven Procedural Content Generation,» *IEEE Transactions on Affective Computing,* vol. 2, nº 3, pp. 147 - 161, July - Sept. 2011.

[35] A. Smith y M. Mateas, «Answer Set Programming for Procedural Content Generation: A Design Space Approach,» *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 3, nº 3, pp. 187 - 200, Sept. 2011.

[36] D. Loiacono, L. Cardamone y P. Lanzi, «Automatic Track Generation for High-End Racing Games Using Evolutionary Computation,» *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 3, nº 3, pp. 245 - 259, Sept. 2011.

[37] J. Togelius, G. Yannakakis, K. Stanley y C. Browne, «Search-Based Procedural Content Generation: A Taxonomy and Survey,» *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 3, nº 3, pp. 172 - 186, Sept. 2011.

ANEXO 1: ACRÓNIMOS

AI	*Atificial Intelligence* (Inteligencia Artificial)
GA	*Genetic Algorithm* (Algoritmo Genético)
LGT	*Level Generation Track*
PCG	*Procedural Content Generation*
PFC	Proyecto Fin de Carrera
SLP	*Single Layer Perceptron* (Perceptrón Simple)
MLP	*Multi Layer Perceptron* (Perceptrón Multicapa)
TFG	Trabajo Fin de Grado
UML	Unified Modeling Language (Lenguaje Unificado de Modelado)

ANEXO 2: GLOSARIO

Agente inteligente: Es un tipo de agente que sigue razonamientos (se entiende que de *AI*) para llegar a una solución sobre un problema planteado.

Algoritmo genético: Es una técnica de la computación evolutiva que resuelve problemas de búsqueda en los que se requiere hallar el resultado óptimo. Para ello, se emplean individuos que forman poblaciones, y sobre dichas poblaciones se aplican operadores genéticos. El proceso se repetirá tantas veces (cada vez que este proceso se lleva a cabo se denomina "generación") como sea necesario para alcanzar el objetivo deseado (hasta alcanzar el criterio de parada establecido).

Coevolución (competitiva): Relación entre dos sistemas en la que deben mejorar tanto uno como el otro para que el entorno se mantenga en equilibrio. O en otras palabras, el éxito de una parte es el fracaso de la opuesta, que debe desarrollar respuestas para mantener sus posibilidades de supervivencia. Gracias a esta técnica, perteneciente a la computación evolutiva, se generan adaptaciones muy complejas.

Computación evolutiva: Es una rama de la Inteligencia Artificial que se basa en la evolución biológica para el desarrollo de sus técnicas.

Criterio de parada: Reglas que al cumplirse, provocan la detección en el desarrollo de nuevas generaciones del algoritmo, y se muestra el resultado obtenido en ese momento.

Cromosoma: Codificación de una solución.

Cruce y mutación: Operadores genéticos de búsqueda.

Determinista: Es aquel sistema que no depende del azar, sino que todo estado A cuando reciba una entrada concreta (unos datos, unas variables del sistema), siempre llegará a otro estado B. Por lo que estando en A y recibiendo en otra ocasión la mista entrada, volverá a llegar a B.

Evolución biológica: Es el conjunto de transformaciones que han sufrido los seres vivos a través del tiempo, que permiten que de un antepasado común existan variedad de nuevos seres distintos. Algunas teorías sobre este concepto fueron las de Lamarck, Darwin y Wallace, Mendel…

Fitness: Calidad de la solución.

***Fitness*, función de**: Criterio de evaluación de las soluciones.

Gen: Parte de la codificación.

Individuo: Solución a un problema.

Inteligencia Artificial: Perteneciente a las ciencias de la computación, "es la ciencia e ingeniería de hacer máquinas inteligentes, especialmente programas de cómputo inteligentes", tal y como la definió John McCarthy en 1956.

Infinite Mario Bros: Una versión de dominio público del clásico juego *Super Mario Bros*, que se ha usado con fines experimentales en la evolución de los

videojuegos mediante técnicas de Inteligencia Artificial. En esta versión está basada la competición anual de *Mario AI Championship*. A Junio de 2012, esta versión se encuentra disponible en: http://www.mojang.com/notch/mario/

Lightscribe: "*LightScribe es una tecnología de etiquetación de discos que le permite grabar una etiqueta láser directamente sobre el disco.*" (http://www.lightscribe.com/gslanding/es/)

Mario: Personaje protagonista de la saga de juegos *Super Mario Bros*, que ha protagonizado gran cantidad de juegos y versiones de los mismos.

Mario AI: Juego basado en *Infinite Mario Bros*, que consiste en la aplicación de *AI* para crear un agente inteligente de *Mario*, generar niveles de forma automática...

Perceptrón multicapa: Combinación de varios perceptrones simples, puesto que dicha combinación podía resolver ciertos problemas no lineales. Es un aproximador universal.

Perceptrón simple: Sistema capaz de realizar tareas de clasificación de forma automática. A partir de un número de ejemplos etiquetados, el sistema determina la ecuación del hiperplano discriminante.

Población: Conjunto de soluciones

Requisito: Es algo que el producto debe hacer o una cualidad que debe tener.

Supervivencia: Cuando un individuo es la suficientemente bueno, se dice que este sobrevive, al menos hasta la siguiente generación (dentro de un algoritmo), pero si la combinación de sus genes no es la adecuada, o bien se ha determinado algún criterio por el que ya no sea tan adecuado como otros, no sobreviviría, y por tanto, se eliminaría de la población para la siguiente generación.

Volere: Una de las múltiples plantillas que existen para la realización de requisitos. Dicha plantilla se puede encontrar en: http://www.volere.co.uk/

ANEXO 3: USO DEL PROGRAMA

Para poder utilizar el programa es necesario disponer de una versión de Java™ mayor o igual a la 1.6.

Como se ha comentado anteriormente, se ha modificado el lanzador por defecto del *Mario AI* a fin de integrar una serie de parámetros de ejecución. A continuación se presentan las distintas posibilidades de invocación a la aplicación desarrollada (y distribuida en un .jar autocontenido) en base a la funcionalidad requerida.

EJECUCIÓN SIN PARÁMETROS

En caso de disponer de una versión de Java™ superior o igual a la 1.6, en caso de hacer doble clic el .jar conseguiremos el mismo resultado que en una invocación desde consola sin parámetros. En este caso, la aplicación se ocupará de generar un nivel mediante el generador aleatorio (el más rápido) utilizando como tipo de nivel y como dificultad un valor aleatorio que cambia en cada ejecución. Una vez generado, el nivel se presenta en la interfaz gráfica del *Mario AI* para poder ser jugado por el usuario.

VALORES DE LOS PARÁMETROS

A partir de este momento se van a describir funcionalidades que requieren la invocación desde consola de comandos, especificando una serie de valores concretos. En este punto se describen los tipos y rangos de dichos valores.

- **Tipo de generador**: hace referencia al generador de niveles a emplear. Puede ser *aleatorio*, *genético* o *coevolutivo*. Los posibles valores para este parámetro son
  - **0**: aleatorio
  - **1**: genético
  - **2**: coevolutivo
  - **Cualquier otro**: muestra mensaje de error.

- **Dificultad a conseguir**: hace referencia a la dificultad asociada al nivel a generar. En caso de generadores evolutivos se corresponderá a un valor de *fitness* asociado al método elegido, y el generador intentará producir uno o varios niveles cuyo *fitness* sea el especificado o un valor que diste en un 1%.
  - **Aleatorio**: *valores enteros entre 0 y 32*, siendo 0 la dificultad más baja y 32 la más alta. Nótese que es posible especificar un valor de dificultad mayor a 32, pero el nivel generado tendrá una dificultad equivalente a uno de 32.
  - **Genético**: *valor entero*. Debido a las posibles longitudes de nivel generadas, los valores tomados son del *orden de centenas*. Cuanto más bajo, nivel más sencillo. Cuanto más alto, nivel más complicado.

  - **Coevolutivo**: *valor entero*. En este caso se corresponde a la puntuación obtenida por un agente automático de evaluación. Valores de *orden de millares*. Cuanto más elevado (puntuación más alta), más sencillo es el nivel generado, mientras que en valores bajos (puntuaciones bajas) más complicados los niveles.

- **Apariencia o tipo del nivel**: determina cómo se mostrará el nivel por la interfaz gráfica de juego de *Mario AI*, tanto para jugar de forma manual como para ser jugado por un agente automático de evaluación. Puede tomar tres valores enteros:
  - **0**: aplica apariencia y música de "superficie".
  - **1**: aplica apariencia y música de "subterráneo".
  - **2**: aplica apariencia y música de "castillo".
  - **Cualquier otro**: igual que 0.

- **Mostrar o jugar**: indica si el agente de evaluación va a mostrar un resumen estadístico textual de los logros tras procesar el nivel o va a mostrar por pantalla una "demostración gráfica" de cómo completar el nivel. Los posibles valores son:
  - **0**: mostrar resumen estadístico.
  - **1**: mostrar gráficamente cómo completar el nivel.

- **Nombre del fichero**: ruta o nombre del fichero a cargar o guardar en cada caso. En caso de una operación de guardado NO SE DEBE especificar la extensión.

GENERAR NIVEL ESPECÍFICO PARA JUGARLO

La aplicación brinda la posibilidad de **generar un nivel para ser jugado** empleando uno de los tres generadores disponibles (aleatorio, genético o coevolutivo). Igualmente para cada caso, es posible especificar la dificultad del nivel que se quiere generar y su apariencia (superficie, subterráneo o castillo).

Para ejecutar en este modo se debe utilizar el parámetro **–g** o **–gen** seguido del **tipo de generador**, la **dificultad a conseguir** y finalmente el **tipo de nivel**.

Ej:

java -jar MarioAI2012_BPH-all.jar ***-g 2 5800 2***

Dicho comando genera un nivel mediante el algoritmo coevolutivo intentando alcanzar un *fitness* de 5800 y tipo "castillo". Una vez generado lo muestra en la interfaz gráfica de *Mario AI* para ser jugado

GENERAR Y GUARDAR NIVEL ESPECÍFICO

La aplicación brinda la posibilidad de **generar un nivel que será guardado** para jugarlo o evaluarlo posteriormente empleando uno de los tres generadores disponibles (aleatorio, genético o coevolutivo). Igualmente para cada caso, es posible especificar la dificultad del nivel que se quiere generar y su apariencia (superficie, subterráneo o castillo). Igualmente se va a guardar la población en el momento de la convergencia para casos en los que se utilicen generadores evolutivos, con el mismo nombre de fichero pero distinta extensión.

Para ejecutar en este modo se debe utilizar el parámetro **–gs** o **–gensav** seguido del **tipo de generador**, la **dificultad a conseguir**, el **tipo de nivel** y finalmente el **nombre** (y opcionalmente la ruta) **del fichero**. En caso de no especificar nombre se creará uno siguiendo un patrón en el que se incluye la fecha y hora del momento en que fue generado.

Ej:

java -jar MarioAI2012_BPH-all.jar ***-gs 1 378 1 pruebaSubterraneo***

Dicho comando genera un nivel mediante el algoritmo genético intentando alcanzar un *fitness* de 378 y tipo "subterráneo". Una vez generado lo guardará en el fichero "pruebaSubterraneo.mlv". Nótese que la extensión se añade automáticamente.

GENERAR Y GUARDAR LOTE DE NIVELES PARA UN FITNESS DETERMINADO

La aplicación brinda la posibilidad de **generar y guardar un lote de niveles** para ser jugados o evaluados posteriormente empleando uno de los dos generadores evolutivos disponibles (genético o coevolutivo). Igualmente para cada caso, es posible especificar la dificultad de los niveles a generar y su apariencia (superficie, subterráneo o castillo). En caso de la dificultad, cabe destacar que el algoritmo convergerá una vez que el *fitness* de todos los individuos se encuentren en un intervalo que diste el 1% del *fitness* especificado. Igualmente se va a guardar la población en el momento de la convergencia, con el mismo nombre de fichero pero distinta extensión y un fichero de texto con las estadísticas y características (número de enemigos, *fitness*, nombre, etc.) de cada uno de los niveles generados.

Para ejecutar en este modo se debe utilizar el parámetro **–gsp** o **–gensavpop** seguido del **tipo de generador**, la **dificultad a conseguir**, el **tipo de nivel** y finalmente el **nombre** (y opcionalmente la ruta) **del fichero**. En caso de no especificar nombre se creará uno siguiendo un patrón concreto e incremental. Los niveles estarán ordenados por su cercanía al *fitness* especificado y por la diversidad genética entre ellos (en caso de mismo *fitness*).

Ej:

java -jar MarioAI2012_BPH-all.jar ***-gsp 1 378 1 levelTest***

Dicho comando genera una población de niveles mediante el algoritmo genético con variación de *fitness* +-1% respecto a 378 y tipo "subterráneo". Una vez generados los guardará en ficheros cuyo nombre irá precedido de "levelTest". Igualmente se guardará

la población en el momento de la convergencia y un fichero de texto con información sobre los niveles generados.

CARGAR NIVEL ESPECÍFICO PARA JUGARLO

La aplicación brinda la posibilidad de **cargar niveles para ser jugados**. Una vez cargados muestra el tipo de generador empleado y la dificultad o *fitness* asociado a dicho nivel y lanza el nivel en la interfaz gráfica de juego de *Mario AI*.

Para ejecutar en este modo se debe utilizar el parámetro **–l** o **–load** seguido del **nombre del nivel** a cargar (con su extensión).

Ej:

java -jar MarioAI2012_BPH-all.jar ***–l z.mlv***

Dicho comando carga el nivel "z.mlv" y lo muestra en la interfaz gráfica de *Mario AI* para ser jugado

CARGAR NIVEL ESPECÍFICO PARA EVALUACIÓN AUTOMÁTICA

La aplicación brinda la posibilidad de **cargar niveles para ser evaluados por agentes inteligentes automáticos**. Una vez evaluados puede mostrar la información asociada a dicha evaluación (*fitness* alcanzado, estado de *Mario* al terminar, tiempo empleado, *power-ups* recogidos, etc.) o bien mostrar una "demostración gráfica" de cómo completar el nivel.

Para ejecutar en este modo se debe utilizar el parámetro **–ev** o **–eval** seguido del valor **mostrar o jugar** y finalmente el **nombre del nivel** a cargar (con su extensión).

Ej:

java -jar MarioAI2012_BPH-all.jar ***–ev 0 z.mlv***

Dicho comando carga el nivel "z.mlv", lo evalúa mediante un agente inteligente genético y muestra los resultados de la evaluación por consola

java -jar MarioAI2012_BPH-all.jar ***–ev 1 a.mlv***

Dicho comando carga el nivel "a.mlv", lo evalúa mediante un agente inteligente genético y una vez evaluado, lleva a cabo en la interfaz de gráfica juego una demostración de cómo completar el nivel en base al resultado de la evaluación.

EN CASO DE ERROR EN LOS PARÁMETROS

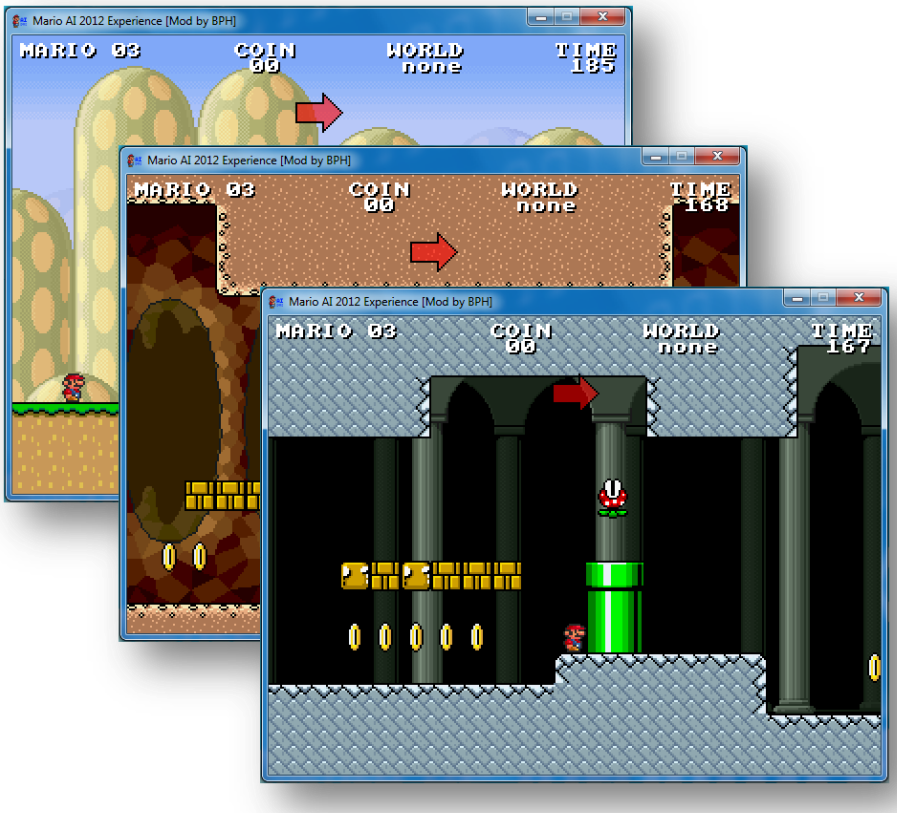
Para facilitar la usabilidad de la aplicación, si en la invocación de la misma se introduce un parámetro o valor erróneo, se mostrará el siguiente mensaje:

```
Puede ejecutar la aplicacion de las siguientes maneras:

        (sin parámetros): nivel de dificultad y tipo aleatorio.
        -g | -gen <tipoGenerador> <dificultad> <tipoNivel>
        -gs | -gensav <tipoGenerador> <dificultad> <tipoNivel> <nombreFichero>
        -gsp | -gensavpop <tipoGenerador> <dificultad> <tipoNivel> <nombreFichero>
        -l | -load <nombreFichero>
        -ev | -eval <mostrarOjugar> <nombreFichero .mlv>

Donde:
        <tipoGenerador>: es el tipo de generador. Puede tener los siguientes valores:
                0: generador aleatorio
                1: generador genético
                2: generador coevolutivo
        <dificultad>: es la dificultad del nivel a generar. Valor entero positivo.
        <tipoNivel>: es el tipo de nivel a generar. Puede tener los siguientes valores:
                0: nivel de superficie
                1: nivel subterraneo
                2: nivel castillo
        <mostrarOjugar>: valor entero [0] -> muestra est. de eval.   [1] -> juega el nivel.
        <nombreFichero>: es la ruta al fichero de nivel que se guardará o cargará.
```

Dicho mensaje puede servir como resumen de todo lo explicado en detalle anteriormente.

ANEXO 4: TECLAS DE JUEGO

Una vez ejecutado el juego (ver la siguiente figura), además de los controles básicos se han implementado una serie de funcionalidades extra accesibles mediante teclado. Se detallan a continuación



MOVIMIENTO

Los movimientos básicos de *Mario* se realizan mediante los cursores, la tecla "a" y la tecla "s":

Desplaza a *Mario* a la izquierda mientras se mantiene pulsada

Desplaza a *Mario* a la derecha mientras se mantiene pulsada

Permite que *Mario* suba por los elementos de escenario que sea posible mientras se mantenga pulsada

Permite que *Mario* se agache cuando está en un estado mejorado (tras tomar una seta o una flor). Esto mientras se encuentre pulsada

Mientras se mantiene pulsada hace que el movimiento de *Mario* sea más rápido (correr)

Hace que *Mario* dé un salto. No es necesario mantener la tecla pulsada. En caso de combinar con un movimiento a izquierda o derecha y "correr", la distancia saltada será mayor

INTERFAZ

El siguiente conjunto de teclas ejecutar funcionalidades específicas:

Finaliza la ejecución de *Mario AI*

Mute. Silencia la música y efectos sonoros de *Mario AI*.

Noisy. Restaura la música y efectos sonoros de *Mario AI*.

Guarda el nivel actual en el directorio en el que se ejecutó con nombre "*level.mlv*". Si existe otro fichero con dicho nombre se sobrescribirá sin confirmación.

En caso de existir, carga y pone disponible para ejecución el nivel "*level.mlv*"

Cabe destacar que las últimas 4 funcionalidades han sido implementadas para mejorar la experiencia de usuario con el juego, ya que tras sucesivas pruebas y casos de ejecución se observó que podrían resultar necesarias.