**UNIVERSIDAD AUTONOMA DE MADRID**

**ESCUELA POLITECNICA SUPERIOR**



**Grado en Ingeniería Informática**

# TRABAJO FIN DE GRADO

# Interfaz automática de generación de niveles en videojuegos para la prueba de algoritmo de hormigas.

# AUTOR: Juan Arano García
# TUTOR: Carlos Aguirre Maeso


**ENERO 2016**


**Dpto. de Ingeniería Informática**
**Escuela Politécnica Superior**
**Universidad Autónoma de Madrid**

## Resumen

El juego de los Lemmings fue muy popular en la década de los 90. La temática del juego consiste en superar los diferentes obstáculos de un nivel partiendo desde un punto inicial y alcanzar un punto de salida. Para lograr este objetivo, los Lemmings disponen de distintas habilidades que puedan usar un determinado número de veces, pudiendo modificar el entorno en el que se encuentran, para así poder alcanzar el punto de salida.

Distintos métodos de inteligencia artificial son comúnmente usados para la resolución de pantallas. En este trabajo de fin de grado se propone un análisis de distintos algoritmos aplicables a la resolución de los niveles y una batería de pruebas del algoritmo de colonia de hormigas comparando sus resultados con un algoritmo genético. De igual modo, también se va a elaborar un método para la generación aleatoria de pantallas mediante un algoritmo codificado en Python.

---

The videogame “Lemmings” was so popular in 90’s. The game consist of passing different obstacles from a start point and reach an exit point. To achieve this goal, Lemmings have different skills that they can use a determinate number of times with the possibility of change the environment in which they are in order to reach the exit point.

Different artificial intelligence methods are commonly used in screen solving. In this paper, different algorithms for screen solving and a battery of tests of ant colony algorithm, comparing their results with a genetic algorithm, are proposed. Similarly, a method for random screen generation is going to be developed in Python.



## Palabras Clave

Lemmings, ACO, GA, Q-Learning, Inteligencia computacional

# Agradecimientos

Agradezco a Antonio González que me haya facilitado el código de su algoritmo para realizar este trabajo. También quiero agradecer a Carlos Aguirre por aceptarme este trabajo después de haberme quedado sin tutor.

# INDICE DE CONTENIDOS


*1 Introducción* 5

**1.1 Motivación** 5

**1.2 Objetivos** 5

*2 Estado del arte* 6

**2.1 Tipos de videojuegos** 6

**2.2 Inteligencia Computacional (CI)** 7

**2.3 Videojuegos adaptables a inteligencia artificial** 7

2.3.1 Age of Conquest 8

2.3.2 Alien Arena 8

2.3.3 Alien Swarm 9

2.3.4 Dolphinity Racer 10

**2.4 Inteligencia computacional en videojuegos** 12

2.4.1 ¿Por qué el juego de Lemmings? 12

**2.5 Algoritmos de redes neuronales convolucionales y Q-Learning** 16

**2.6 Algoritmos genéticos** 18

2.6.1 Algoritmos genéticos aplicados a los Lemmings 20

**2.7 Algoritmo de colonia de hormigas (ACO)** 20

**2.8 Generación automática del nivel** 22

2.8.1 Gramática libre de contexto 22

2.8.2 Aplicación de la gramática en Python 23

*3 Pruebas* 24

**3.1 Resultados para Lemmings** 24

3.1.1 Implementación de ACO para Lemmings 24

**3.2 Resultados para la interfaz propuesta** 25

3.2.1 Generación de los niveles de prueba en Python 25

3.2.2 Implementación de ACO para la interfaz adaptada 27

*4 Conclusiones y trabajo futuro* 33

**4.1 Conclusiones** 33

**4.2 Trabajo futuro** 33

*4. Bibliografía* 35

*5. Anexos* - 1 -

# INDICE DE FIGURAS

Ilustración 1 – Age of Conquest 8
Ilustración 2 – Alien Arena 9
Ilustración 3 – Alien Swarm 10
Ilustración 4 – Dolphin Racer 1 11
Ilustración 5 - Dolphin Racer 2 11
Ilustración 6 - Lemmings 12
Ilustración 7 – Mapa Lemmings 13
Ilustración 8 – Mapa Fácil 14
Ilustración 9 – Mapa Medio 14
Ilustración 10 – Mapa Difícil 15
Ilustración 11 – Algoritmo Deep Q-learning 17
Ilustración 12 – ACO Pseudocódigo 20
Ilustración 13 – Gráfico de generación del nivel 26
Ilustración 14 – Simulación mediante ACO 28
Ilustración 15 – Simulación mediante GA 29
Ilustración 16 – Paths totales 30
Ilustración 17 – Path Óptimo 30
Ilustración 18 – Simulación del nivel generado 31
Ilustración 19 – Final del nivel 32

# INDICE DE TABLAS

Tabla 1 – Resultados de DQN vs métodos de aprendizaje 18
Tabla 2 - Posibles Tiles 23
Tabla 3 - Configuración para Algoritmo Genético 24
Tabla 4 - Configuración para Algoritmo de Colonia de Hormigas 24
Tabla 5 - Resultados de la simulación 25
Tabla 6 - Configuración para Algoritmo de Colonia de Hormigas 27
Tabla 7 - Configuración para Algoritmo Genético 28
Tabla 8 - Resultados de la simulación 32
Tabla 9 – Tiempos de ejecución 33

# 1 Introducción

## *1.1 Motivación*

El diseño de mi interfaz adaptada de Lemmings que entregué en la asignatura de Videojuegos, me motivó para seguir un paso más allá. Mi profesor me convenció para aplicar inteligencia artificial a mi juego y realizar un estudio sobre la generación automática de niveles y la resolución de los mismos. Creo que es un objeto de interés porque se pueden enfrentar varios algoritmos entre sí y determinar cuál puede llegar a ser más eficiente a la hora de resolver un nivel. Del mismo modo, la generación automática de niveles también es interesante de cara a diseñar niveles con cierto parecido para diferentes videojuegos y aliviar así esta tarea.

## *1.2 Objetivos*

El proyecto consistirá en el diseño e implementación una interfaz que permita la creación automática de niveles (o pantallas) de un juego. El interfaz tendrá en cuenta tanto el nivel completo que debe ser resuelto, como los métodos y procedimientos adecuados que permitan tanto la captura de interacciones, como la prueba y validación de algoritmos de búsqueda y optimización que puedan ser empleados para la resolución (automática) del nivel generado.

El proyecto realizará dos estudios independientes del estado del arte, por un lado del tipo de videojuegos que podrían ser empleados para la resolución automática de pantallas, y por otro se realizará un breve análisis de los algoritmos basados en Inteligencia Artificial que pueden utilizarse para conocer si existe, o no, solución a un determinado nivel, o aquellos algoritmos que pueden optimizar la resolución del mismo (por ejemplo utilizando el menor número de acciones posibles).

Como segundo objetivo del proyecto, se estudiará y analizará el funcionamiento básico de los algoritmos de optimización basados en hormigas (o ACO, del inglés Ant Colony Optimization), se adaptará uno de estos algoritmos sobre el interfaz creado y se validará experimentalmente el funcionamiento del mismo sobre un conjunto de niveles generados.

# 2 Estado del arte

## 2.1 Tipos de videojuegos

A lo largo de la historia de los videojuegos, varios elementos que han compartido los mismos han servido para clasificarlos en distintos géneros de la misma manera que con el cine. Estos géneros se han conformado en torno a factores como la representación gráfica, interacción entre jugador y máquina, ambientación y su sistema de juego, siendo este el criterio más habitual a la hora de valorar.

Cada vez se hace más difícil clasificar un juego por sus nuevos elementos diversos. Las principales categorías son:

- *Acción*: El jugador hace uso de sus reflejos, puntería y habilidad superando obstáculos.
- *Disparos*: El jugador hace uso continuado de armas de juego para abrirse paso.
- *Estrategia*: Se manipula un grupo de personajes para lograr distintos objetivos.
- *Simulación*: Recrea situaciones dejando al jugador tomar el control de lo que ocurre.
- *Deporte*: Simulan deportes del mundo real.
- *Carreras*: Sitúan al jugador en un recorrido en el que debe llegar a una meta.
- *Aventura*: El jugador avanza por una trama interactuando con personajes y objetos.
- *Rol*: Interacción con el personaje en una historia profunda y una evolución del personaje a medida que avanza la historia.
- *Otros géneros*: Géneros secundarios como Educativos, Musical, Agilidad mental, puzzle…

Nuestra interfaz adaptada de los Lemmings encajaría en el apartado de *Otros géneros*. Se le puede identificar dentro de la categoría de *puzzle.*

## 2.2 Inteligencia Computacional (CI)

Es un conjunto de algoritmos que se han utilizado ampliamente para resolver problemas en áreas como planificación o problemas de satisfacción de restricciones (CSP). Esta última área tiene mucha importancia por su aplicabilidad a problemas reales. A continuación se van a explicar dos algoritmos particulares de CI: optimización basada en colonias de hormigas (ACO) y algoritmos genéticos (GA) en problemas con los videojuegos. Como ejemplo de investigación se ha elegido el juego de los **Lemmings** en el que se debe encontrar un punto de salida en cada nivel. Para conseguir alcanzar dicho punto, los Lemmings deben sortear obstáculos realizando distintos movimientos dentro de la pantalla. El objetivo de estos algoritmos es asignar las mejores habilidades en cada punto para guiar a los lemmings hasta la salida. Se van a comparar como trabajan ambos algoritmos respectivamente con este juego. Ambos algoritmos trabajan con un cierto número posible de soluciones del problema modelado. Además, también necesitan un cierto ajuste en su funcionalidad para guiar al algoritmo a la solución más óptima. Estas características facilitan la comparación entre los dos algoritmos.

En el caso de los GA, cada individuo es evaluado por una función ajustada que permite la evaluación individual. Después, aquellos algoritmos con mejores valores de ajuste tendrán una probabilidad mayor de ser seleccionados para la fase de reproducción. El operador que facilita esta generación individual para tener en cuenta las características del mejor se llama *crossover*. Con este operador, las características individuales se construyen recombinando sus genotipos padres. Finalmente, el operador *mutation* cambia aleatoriamente algunos genes de los nuevos individuales. Estos operadores proveen los mecanismos de explotación y exploración del espacio del problema.

Los algoritmos de enjambre de inteligencia como por ejemplo ACO, están enfocados al comportamiento de la auto-organización de los sistemas en las iteraciones por los colectivos de conocimiento individuales basados en colonias sociales. Algunos ejemplos de este tipo de algoritmos son *particle swam optimización, bee colony optimization, bird flocking o bacterial foraging.* En todos estos algoritmos no hay generación de nuevos individuos. En vez de esto, la población viaja a través del espacio de solución para obtener la mejor solución al problema. El ajuste funcional es usualmente diseñado como parte de la meta-heurística usada por este tipo de algoritmos.

El juego de Lemmings es un juego de tipo puzzle popular en los años 90, que ha sido objetivo de bancos de prueba por los Algoritmos Computacionales. Estos algoritmos también se han aplicado a ciertos juegos como Mastermind, the Art of zen, Pac-Man, Tetris o Mario Bros.

## 2.3 Videojuegos adaptables a inteligencia artificial

En este apartado se va a realizar un análisis sobre distintos tipos de videojuegos a los cuales se les puede aplicar técnicas de inteligencia artificial (IA) para conseguir resolver un nivel de un videojuego de manera autónoma.
Cada juego tendrá una descripción en la que se detallara las características que podrán usarse para desarrollar el agente que resuelva automáticamente los niveles. De igual modo, se repasarán distintas investigaciones al respecto en las que aplican inteligencia artificial a distintos videojuegos.

### 2.3.1 Age of Conquest

Es un juego medieval de tipo Risk, que se basa en estrategia por turnos en el que se tiene que administrar un imperio, aún en formación, y luchar contra diferentes imperios por la hegemonía. El juego está adaptado para los sistemas operáticos principales como Windows, Macintosh, Linux, Android e IOS.

Posee una versión gratuita para descargar pero que está limitada a sólo un escenario. Aun con esta limitación, es suficiente para diseñar un agente que sea capaz de resolver el nivel de manera exitosa.

El motivo principal para descartar este videojuego, es que carece de una API ni guías de su implementación o funcionalidad, por lo que habría que diseñarla para interactuar con el entorno del juego.

Hasta la fecha existen numerosos trabajos realizados que aplican técnicas de IA a videojuegos de este tipo de género. Un ejemplo, es el trabajo de Sánchez-Pelegrín, Gómez-Martín y Díaz-Agudo, [6], que utiliza Razonamiento Basado en Casos (CBR) para desarrollar un módulo basado en IA. En [7] se utilizan ejemplos que han sido generados por humanos para guiar a un algoritmo de tipo neuro-evolutivo Lamackiano.

Teniendo en cuenta el comportamiento de las reglas integradas en el juego de los oponentes, se puede usar Programación Genética para crear estas reglas de forma automática mediante agentes humanos o autónomos.

**Ilustración 1 – Age of Conquest**

### 2.3.2 Alien Arena

Se trata de un juego de acción y está ambientado en la estética de las películas de los años 60. Se centra en la acción multijugador en la red, y aunque contiene partidas individuales contra robots, se puede interactuar con otro jugador para eliminar enemigos.

Alien Arena consta de 37 mapas, 10 personajes, 8 clases de armas, 5 modalidades de juego, 23 niveles y un navegador incorporado con el que se gestiona a los usuarios de la red. Tiene un motor gráfico potente con efectos visuales adversos y tiene la opción de adquirir mejoras para armadura o armas y aventajarse a otros jugadores.

Este juego se ha lanzado para Windows, Linux y FreeBSD. Este juego es gratuito pero lleva integrada publicidad dentro del juego. Se puede encontrar el código a través del sistema de control de versiones SVN.

Este título se clasifica dentro del género FPS. En la actualidad, no se utilizan prácticamente técnicas de IA para el comportamiento de los oponentes o "bots" en el juego. Tradicionalmente se usan métodos como máquinas de estado, sistemas basados en reglas y scripting.


**Ilustración 2 – Alien Arena**

Existen varios trabajos que aplican técnicas de IA en estos ámbitos como el realizado por McPartland y Gallagher, [8], que utilizan aprendizaje por refuerzo (RL) combinado con 3 métodos. El primero se combina con combate previamente entrenado y un controlador de navegación que utiliza RL Jerárquico. El segundo utiliza reglas simples para determinar en qué momento ir al combate o controlar la navegación. El tercero utiliza RL para reaprender las tareas de combate y navegación simultáneamente.
El controlador RL Jerárquico es el que mejor funciona para disparar al objetivo con precisión superando al resto de controladores. Tanto el controlador de RL Jerárquico como el controlador RL basado en reglas muestran un potencial para creación de comportamiento de los "bots" en los FPS.

En [9], se presenta un método eficiente para utilizar algoritmos genéticos que van evolucionan parámetros para los "bots" que consigue hacer que jueguen igual que los parámetros ajustados por un ser humano con conocimiento experto sobre la estrategia. Identifican 2 tipos de parámetros a optimizar. El primero es la selección de armas y el segundo la agresividad que afectará a la preferencia a la hora de seleccionar un camino.

### 2.3.3 Alien Swarm

Es un juego para el Source SDK creado por un grupo de diseñadores de la comunidad de Mods contratados por Valve Corporation que pertenece al género TPS. El source SDK es un equipo de desarrollo de software para que los jugadores que utilicen el motor Source, puedan crear sus propios mapas o ver modelos con su herramienta de Modelo Viewer y Model Poser.

El juego consiste en aniquilar bichos de forma indiscriminada. Se ofrece una combinación de juego cooperativo y táctica de escuadrones. Se permiten formar escuadrones de cuatro

clases de marines que cooperan entre ellos. Se pueden planificar ataques contra una variedad de alienígenas haciendo uso de un arsenal de armas. El objetivo principal del juego es acabar con la invasión alienígena en entornos fuera de la Tierra.

Junto con el juego, se obtiene la base completa del código de Alien Swam que cuenta con actualizaciones para el motor "Source" y del SDK. El código del juego está realizado en Visual Studio. Toda la documentación está en la web del juego.

**Ilustración 3 – Alien Swarm**

El juego es muy interesante para implementarle IA, pero tiene el inconveniente de que no tiene un API que permita aplicarla, ya que las que existen están orientadas a la creación de mapas, personajes y armas, por lo que habría que diseñar otra para obtener información del juego.

Hay una gran cantidad de investigaciones sobre la aplicación de IA en videojuegos de genero FPS. Por lo contrario, para los juegos TPS, no existen tantas debido a que hay una mayor libertad de movimientos y entornos más complejos, por lo que realizar un algoritmo eficiente se vuelve muy complicado.
En [10], hay un algoritmo de planificación de movimientos en tiempo real. Se apoyan en el concepto de "espacio CT" (espacio Configuración-Tiempo) para diseñar un algoritmo de planificación de movimientos que genere movimientos sin colisiones para la parte superior del cuerpo del avatar caminando en un entorno virtual desordenado. Es una buena aproximación para aplicar a los entornos generales de los TPS pero no a un entorno especifico en particular.

En este tipo de videojuegos, sería conveniente juntar varias técnicas de IA para guiar al personaje por el entorno de una forma eficiente, defendiéndose y atacando a los enemigos de manera simultánea. Un algoritmo que podría funcionar es el de colonia de hormigas, aunque debería acotarse las direcciones de los caminos que el personaje puede elegir. Se podrían tomar por ejemplo los puntos cardinales Norte, Sur, Este…

### 2.3.4 Dolphinity Racer

Es un proyecto de simulación de coches en multiplataforma, que utilizando física profesional, consiguen una sensación realista, y gracias a un excelente motor de renderizado unos gráficos increíbles. Los coches, pistas y escenarios están creados con relativa simplicidad. El 3D y otros formatos de archivo están documentos. Los editores y programas de apoyo también se pueden descargar para configurar un simulador. OpenGL se utiliza para la representación.

Este juego es gratis y está disponible para Windows, Linux y Mac OSX. Utiliza fórmulas de movimiento de los documentos de ingeniería actual del SAE, tiene soporte para Matlab y para Matrox Surround Gaming. Existe una gran cantidad de coches y pistas disponibles en web. Tienes la opción de crear coches y pistas personales con ZModeler, 3d Studio Max… Se soportan vehículos de 2 a 8 ruedas y los movimientos simulan la funcionalidad del volante del coche.


**Ilustración 4 – Dolphin Racer 1**

**Ilustración 5 - Dolphin Racer 2**

Actualmente hay numerosas investigaciones que aplican técnicas de IA a simuladores de coches. En [20], se examinan estrategias mediante Redes Neuronales y estrategias mediante Inteligencia Artificial Basada en Comportamiento (BBAI). Esta última fue popular en el campo de la robótica. La inteligencia se percibe como un número de componentes modulares relativamente simple. Cada componente funciona dentro de un conjunto específico de condiciones. BBAI funciona sin la búsqueda, por lo tanto tiene éxito en aplicaciones de tiempo crítico como la robótica y la realidad virtual interactiva. Por otro lado, las Redes de Neuronas ofrecen una metodología alternativa para implementar IA. Pueden desarrollar estrategias interesantes y efectivas que se pasan por alto por los diseñadores basados en comportamientos.

El controlador basado en comportamiento evolutivo superó al controlador de red de neuronas. Esto hace que el primero sea un candidato para los juegos con graficos intensos con múltiples agentes que inteactúan en el juego para asegurar la presentación continua del juego.

Otro trabajo, el [11], cuyo objetivo es desarrollar un controlador evolutivo para simuladores de carreras utilizando programación genética (PG), hace una comparación entre la neuro-evolución y la PG, con y sin estados. Dado un problema y una configuración experimental, las versiones de PG evolucionan más rapido, pero las redes de neuronas trabajan mejor a largo plazo.

## *2.4 Inteligencia computacional en videojuegos*

Tradicionalmente se utilizan técnicas de inteligencia artificial y otras sub-áreas como inteligencia computacional en juegos clásicos de tableros como ajedrez, othello… El alto impacto de los videojuegos ha generado un amplio número de conferencias internacionales con el propósito de competir con la utilización de inteligencia artificial y computacional para resolver problemas como: resolución de pantallas, comportamientos de los Non-Player Characters (NPC), generación automática de pantallas, etc.

La idea básica de este tipo de competiciones está basada en la utilización del videojuego que puede ser usado para integrar algoritmos de IA/IC para comprobar su comportamiento en una batalla en particular. En ellas se ponen a prueba un juego en una determinada plataforma y se hace pruebas con algoritmos de IA e IC para comprobar que objetivos han sido alcanzados en cada competición.

Algunos algoritmos que se utilizan en el juego están explicados en la web, por lo que puede ser interesante para crear tu propio simulador de coches.

Se puede modificar la IA de los coches con ficheros ".ini", los cuales son fácilmente modificables. El único inconveniente de este juego, es que hay que desarrollar una API propia para interactuar con el entorno del juego.

### 2.4.1 ¿Por qué el juego de Lemmings?

El juego de los lemmings se ha presentado como un nuevo "Drosophila" para investigación de inteligencia artificial. Los scripts están desarrollados usando un algoritmo genético. Los resultados muestran una aproximación posible para resolver incrementalmente los mapas complejos.

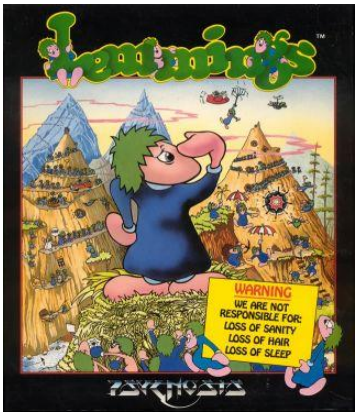



**Ilustración 6 - Lemmings**

El juego de los Lemmings es un videojuego de 1 jugador que se muestra en una pantalla en un mapa bidimensional. El objetivo consiste en entidades llamadas lemmings que se mueven alrededor de un entorno. Este entorno contiene una entrada por la que llegan los lemmings y una salida que garantiza su salvación. El peligro principal está en el fuego, agua o cuchillas rotadoras que matan a los lemmings al tocarlos.
Los lemmings caminan por defecto. Un lemming caminante se mueve por defecto a la derecha y no puede quedarse parado. El lemming cambiará de dirección si su camino es bloqueado. Si un caminante se aproxima al final de la plataforma, continuará y caerá. Un lemming que caiga desde demasiada altura morirá cuando entre en contacto con el suelo.
Hay 8 movimientos especiales que se les pueden asignar a los lemmings:

1. **Basher**: hace un agujero en una pared vertical y vuelve a andar en cuanto la atraviese

2. **Blocker**: se queda parado y bloquea el camino de los demás lemmings obligándoles a cambiar de destino. Este lemming está condenado a morir pues no podrá moverse más.

3. **Bomber**: cuenta 5 segundos y después explota y muere destruyendo parte del entorno que le rodea.

4. **Builder**: construye diagonalmente un puente hasta que un bloque solido entra en contacto con el puente. Luego vuelve a andar

5. **Climber**: Se mueve igual que un Walker pero escala hacia arriba las paredes. Escalará todas las paredes que se encuentre por el camino

6. **Digger**: hace un agujero en el suelo y vuelve a ser un caminante en cuanto lo atraviese

7. **Floater**: se mueve de la misma manera que un Walker pero sobrevive cuando cae desde cualquier altura.

8. **Miner**: Escava diagonalmente y vuelve a ser un Walker cucando atraviesa el obstáculo


**Ilustración 7 – Mapa Lemmings**

Los lemmings que sigan activos en el mapa pueden ser explotados. Esta opción es necesaria para acabar con los lemmings que bloquean o son necesarios de eliminar para acabar el nivel.

Los lemmings se comportan de una manera que hacen aprender al jugador a través de la experiencia de juego. Este tipo de comportamientos son los que hace que el juego sea más atractivo y competitivo.
Existen paredes y plataformas que no pueden ser destruidas con las habilidades y paredes que sólo pueden atravesarse en una dirección. Dependiendo de los tipos de materiales con los que se conforman los niveles, se pueden dividir los niveles en 3 grados de complejidad distintos:

- **Fácil**: Los niveles se componen de paredes y suelos que pueden ser destruidos o permanecer inamovibles. El número de acciones que los lemmings necesitan para alcanzar la salida es pequeño. Al encontrarse con materiales con los que no pueden interactuar el espacio de búsqueda queda reducido.


**Ilustración 8 – Mapa Fácil**

- **Medio**: Los niveles se componen de ambos tipos de paredes y suelos. La diferencia radica en que ambos tipos pueden ser usados y hay zonas donde se posible encontrarse una variedad mayor de acciones posibles para encontrar la solución del problema.


**Ilustración 9 – Mapa Medio**

- **Difícil**: Los niveles se componen únicamente de paredes y suelos que pueden ser modificadas. Para alcanzar la solución pueden ser necesarias un gran número de acciones y pueden elegirse varias habilidades por lo que el problema se hace más grande y el espacio de soluciones aumenta considerablemente.


**Ilustración 10 – Mapa Difícil**

Cada nivel dispone de un número máximo de veces que puede ser utilizada cierta habilidad, aunque no es necesario usarlas todas. Eliminando ciertas habilidades en cada nivel se podría aumentar la complejidad del problema por lo que la búsqueda de la solución será más costosa.

El objetivo del juego es que para un número específico de lemmings, se deben salvar el máximo número de ellos antes de un tiempo determinado. Si el jugador forma una estrategia que facilita la superación del nivel, luego avanzará a otro nivel que representa mayor complejidad y requerirá un mayor número de lemmings salvados.

John McCarthy en [12], a pesar de su introducción al juego como un problema, el autor no tiene conocimientos de cómo crear un jugador automático para el juego.

Los Lemmings es una plataforma interesante por la complejidad de sus reglas. El puzle es difícil de entender y describir. Por ello se ha desarrollado un simulador que deriva del videojuego de los lemmings para facilitarnos el análisis. Esta complejidad hace que el problema se aproxime mucho a la vida real conservando restricciones y objetivos claramente definidos en otros puzles. Es por esta razón, junto con la clasificación de los lemmings como un puzle auto actualizado puzle, que se propone Lemmings como un dominio de investigación para inteligencia artificial.

## 2.5 Algoritmos de redes neuronales convolucionales y Q-Learning

Una red neuronal convolucional es un tipo de red neuronal artificial donde las neuronas corresponden a campos receptivos de manera similar a las neuronas de un cerebro. Este tipo de red es una variación de un perceptrón multicapa, pero por su funcionamiento las hace más efectiva para tareas de visión artificial.

La red es entrenada con una variante del algoritmo Q-Learning con descenso de gradiente estocástico para actualizar los pesos. Para aliviar los problemas de los datos correlacionados y distribuciones no estacionarias, se usa un mecanismo que repetición de experiencia que muestra al azar transiciones previas y suaviza la distribución durante comportamientos pasados. Esta aproximación se ha aplicado a juegos de Atari 2600 implementados con aprendizaje del entorno.

Un agente interactúa con un entorno ε en una secuencia de acciones, observaciones y recompensas. En cada paso el agente selecciona una acción $\alpha_t$ de un conjunto de acciones $A = \{1, \ldots., K\}$. La acción es pasada al emulador y modifica el estado interno y la puntuación del juego. En general ε puede ser estocástico. El estado interno del emulador no es observado por el agente sino que observa una imagen $x_t \in \mathbb{R}^d$ del emulador, que es un vector del valor de un pixel que representa la pantalla actual. Así recibe una recompensa $r_t$ representando el cambio en la puntuación del juego.

Como el agente sólo observa imágenes de la pantalla actual, es imposible entender la situación actual a través del pixel actual, por lo que se consideran un conjunto de acciones y observaciones y se aprenden estrategias de juego que dependen de estas secuencias. Todas las secuencias se asume que deben terminar en un número finito de pasos. Esto lo convierte en un proceso de decisión de Markov (MDP) donde cada secuencia es un estado.

El objetivo del agente es interactuar con el emulador seleccionando acciones que incrementen las futuras recompensas. Estas futuras recompensas son descontadas por un factor de γ por step, y se define el retorno a tiempo t como $R_t = \sum_{t'=t}^{T} \gamma^{t'-t}\, t_{t'}$, donde T es el paso en el que el juego acaba. Se define una óptima función de acción-valor $Q*(s,a)$ como el retorno máximo alcanzable siguiendo cualquier estrategia después de seguir alguna secuencia $s$ y tomar alguna acción $a$, $Q*(s,a) = max_\pi \mathbb{E}[\, R_t \mid s_t = s,\ a_t = a, \pi]$, donde $\pi$ es una secuencia de política de acciones.

La función óptima de acción-valor obedece a una importante identidad conocida como ecuación de Bellman, que está basada en la intuición: si el valor óptimo $Q*(s',a')$ de las secuencias $s'$ en el siguiente step fue conocida por todas las posibles acciones $a'$, entonces la estrategia óptima es seleccionar la acción $a'$ maximizando el valor esperado de $r + \gamma Q*(s',a')$

$$Q*(s,a) = \mathbb{E}_{s'\sim\varepsilon}\left[\, r + \gamma \max(a')\, Q*(s',a') \mid s,a\right]$$

La idea básica detrás de muchos algoritmos de aprendizaje es estimar la función de acción-valor usando la ecuación de Bellman como una actualización interactiva $Q_{i+1}*(s,a) = \mathbb{E}_{s'\sim\varepsilon}\left[\, r + \gamma \max(a')\, Q*(s',a') \mid s,a\right]$. En la práctica, esta aproximación es impracticable porque la función acción-valor es estimada cada secuencia de manera separada sin ninguna

generalidad. Es común usar una función de aproximación para estimar la función de acción-valor $Q(s,a;\theta) \approx Q*(s,a)$. Nos referimos a una función de aproximación de una red neuronal con peso θ como una Q-network. Ésta puede ser entrenada minimizando una secuencia de funciones de pérdida $L_i(\theta_i)$ que cambia en cada iteración i.

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim p(.)}[(y_i h - Q(s,a;\theta_i))^2]$$

$y_i = \mathbb{E}_{s'\sim\varepsilon}[r + \gamma \max(a') Q*(s',a';)\theta_{i-1}|s,a]$ corresponde al objetivo de cada iteración y $p(s,a)$ es una probable distribución sobre las secuencias (s) y las acciones (a) que se refieren a la distribución de comportamiento. Los parámetros de la anterior iteración se ajustan cuando se optimiza la función de perdida Li (θi).

$$\nabla_\theta L_i(\theta_i) = \mathbb{E}_{s,a\sim p(.);s'\sim\varepsilon}[(r + \gamma \max(a') Q(s',a';\theta_i - 1) - Q(s,a;\theta_i))\nabla_{\theta i} Q(s,a;\theta_i)]$$

Antes que computar todas las expectaciones en el gradiente superior, es mejor optimizar la función de pérdida a través del gradiente descendente estocástico. Si el peso es actualizado cada paso y las expectativas son reemplazadas por ejemplos de comportamiento de la distribución de comportamiento y el emulador respectivamente, llegamos al familiar algoritmo de Q-learning.

A través de esto, se pretende conectar el algoritmo de aprendizaje profundo con las redes neuronales que operan directamente sobre imágenes RGB y procesan datos usando actualizaciones estocásticas de los gradientes. Este tipo de aproximación tiene algunas ventajas sobre los estándares Q-learning. Primero, cada paso de la experiencia es potencialmente usado en muchas de las actualizaciones, lo cual genera una eficiencia en los datos recopilados.

**Algorithm 1** Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  **for** $t = 1, T$ **do**
    With probability $\epsilon$ select a random action $a_t$
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  **end for**
**end for**

**Ilustración 11 – Algoritmo Deep Q-learning**

El algoritmo sólo almacena las últimas tuplas de experiencias N en la memoria de replay y los samples, uniformemente al azar, cuando se realizan actualizaciones. Esta aproximación está limitada cuando el buffer no diferencia transiciones importantes y siempre sobrescribe con las nuevas transiciones debido al tamaño de la memoria. Una estrategia mejor, puede ser empatizar con las transiciones para aprender lo máximo posible.

De [13], podemos extraer la tabla donde se compara esta aproximación del algoritmo Deep Q-Learning contra otros.

| | B. Rider | Breakout | Enduro | Pong | Q*bert | Seaquest | S. Invaders |
|---|---|---|---|---|---|---|---|
| **Random** | 354 | 1.2 | 0 | −20.4 | 157 | 110 | 179 |
| **Sarsa** [3] | 996 | 5.2 | 129 | −19 | 614 | 665 | 271 |
| **Contingency** [4] | 1743 | 6 | 159 | −17 | 960 | 723 | 268 |
| **DQN** | **4092** | **168** | **470** | **20** | **1952** | **1705** | **581** |
| **Human** | 7456 | 31 | 368 | −3 | 18900 | 28010 | 3690 |
| **HNeat Best** [8] | 3616 | 52 | 106 | 19 | 1800 | 920 | **1720** |
| **HNeat Pixel** [8] | 1332 | 4 | 91 | −16 | 1325 | 800 | 1145 |
| **DQN Best** | **5184** | **225** | **661** | **21** | **4500** | **1740** | 1075 |

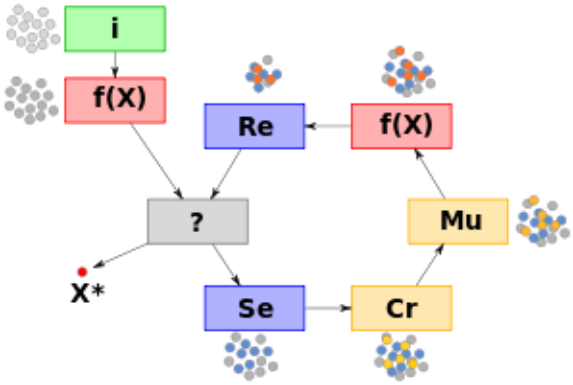
**Tabla 1 – Resultados de DQN vs métodos de aprendizaje**

La conclusión que saca [13] es que se demuestra la habilidad para resolver los problemas de los juegos de Atari 2600 usando sólo pixeles de imágenes como input.

## *2.6 Algoritmos genéticos*

En los años 50 se comenzó a aplicar los principios de Darwin en la resolución de problemas. Durante los años 60 y 70 varias corrientes de investigación comenzaron a formar parte lo que ahora se conoce como computación evolutiva. Los algoritmos genéticos fueron propuestos por John. H. Holland y su motivación inicial fue la de proponer un modelo general de proceso adaptable. Estos algoritmos hacen evolucionar una población de individuos sometiéndola a acciones aleatorias semejantes a las que actúan en la evolución biológica. De igual modo, se les somete a una selección de acuerdo con algún criterio por la que se determina cuáles son los individuos más adaptados, que sobreviven, y cuáles los menos aptos, que son descartados.
Los algoritmos genéticos funcionan entre el conjunto de soluciones de un problema (fenotipo) y el conjunto de individuos de una población natural codificando la información de cada solución en una cadena usualmente binaria (cromosoma). Los símbolos de la cadena son los llamados genes. Los cromosomas evolucionan a través de iteraciones. En cada una de ellas, son evaluados y las siguientes iteraciones son generadas aplicando los operadores genéticos repetidamente.

En general el algoritmo genético consiste en los siguientes pasos:




1- **Inicialización**: Se genera de manera aleatoria la población inicial que representan las posibles soluciones del problema. En caso de no generarse de manera aleatoria es importante hacer que se tenga la diversidad estructural de estas soluciones para tener una representación de la mayor parte de la población posible.

2- **Evaluación**: A cada uno de los cromosomas de la población se le aplica la función de aptitud para saber cómo de buena es la solución que se codifica.

3- **Condición del término**: El algoritmo se debe detener cuando se alcance la solución óptima usando criterios de detención. Normalmente se procede a correr el algoritmo genético un número máximo de iteraciones o detenerlo cuando no haya cambios. Mientras la condición de termino no se cumpla, se hace lo siguiente:

   - **Selección**: Sabiendo la aptitud de cada cromosoma se elige a los que serán cruzados en la siguiente iteración. Los mejores tienen mayor probabilidad de ser cruzados.

   - **Recombinación o Cruzamiento:** Este es el principal operador genético que representa la reproducción sexual y opera para generar dos descendientes que combinan características de ambos cromosomas padres.

   - **Mutación:** Modifica al azar parte del cromosoma y permite alcanzar espacios de búsqueda que no estaban cubiertos por la población actual.

   - **Reemplazo:** Se seleccionan los mejores individuos para conformar la siguiente población.

### 2.6.1 Algoritmos genéticos aplicados a los Lemmings

Estos algoritmos aplicados a este juego inicializan individuos con una longitud de genotipo aleatoria. La longitud máxima depende del tiempo para resolver el nivel o de la máxima longitud del genotipo permitida. Este genotipo es una lista de genes donde cada uno (< T, S >) contiene una habilidad o skill (S) que va a ser ejecutada en el siguiente step (T). Ambos valores (Step y Skill) son seleccionados aleatoriamente dependiendo del máximo tiempo dado para resolver el nivel y el número restante de habilidades. El genotipo representa las diferentes decisiones que el jugador puede tomar. El lemming empieza su ejecución aplicando las skills especificadas en los steps dados.

El objetivo es maximizar la siguiente función fitness:

$$F(Ind) = \frac{\mathrm{T(Ind)} + \mathrm{A(Ind)} + \mathrm{S(Ind)}}{\mathrm{MaxTime} + \mathrm{TotalActionsAvailable} + \mathrm{TotalLemmings}}$$

$$\mathrm{S(Ind)} = \mathrm{TotalLemmings} - \mathrm{BlockersUsed(Ind)} - \mathrm{ExplodedLemmings(Ind)}$$
$$\mathrm{T(Ind)} = \mathrm{MaxTime} - \mathrm{Time(Ind)} * \mathrm{S(Ind)}$$
$$\mathrm{A(Ind)} = \mathrm{TotalActionsAvailable} - \mathrm{ActionUsed(Ind)}$$

Aunque el ACO usara la misma función para evaluar el mejor de los path, sólo GA puede producir valores negativos para la función fitness. Este valor negativo es obtenido en el caso de que el individuo produzca un path inválido (el lemming no alcanza el punto de salida o muere intentándolo). En este caso, el valor de la función es -1. Los paths de la solución más óptima tendrán valores próximos a 0, mientras que los peores tendrán valores muy próximos a -1.

## *2.7 Algoritmo de colonia de hormigas (ACO)*

En este caso, el nido de la colonia se encuentra en el nodo que representa el punto de partida del nivel y la comida está localizada en el nodo que representa el punto de salida del nivel. Desde el nido, las hormigas empiezan construyendo su solución local mientras que viajan a través del grafo. Para conseguirlo, cada una ejecuta el comportamiento descrito en el siguiente Algoritmo:

**Algorithm 1:** Ants behaviour.

```
1  S ← possibleSkills(currentPosition)
2  P ← pheromoneInformation(currentPosition)
3  skill ← selectSkill(S,P)
4  if (skill ≠ null)) then
5  |  if canBeExecuted(skill,currentPositions) then
6  |  |  updateRemainingSkill()
7  |  |  putPheromone()
8  |  |  updateCurrentAction(skill)
9  |  end
10 end
11 execute(currentAction)
```

**Ilustración 12 – ACO Pseudocódigo**

La primera línea en este comportamiento corresponde con la heurística del problema. Existen dos heurísticas diferentes que se han llamado *Heurística Aleatoria* y *Heurística de sentido común*. Usando la primera, cada hormiga puede ejecutar cualquier habilidad en cualquier posición. En la segunda heurística, las hormigas pueden percibir el entorno en el que se encuentran y filtrar las habilidades que pueden usar dependiendo de este. Por ejemplo, si una hormiga está rodeada única y exclusivamente de aire, aplicándolo al caso de los Lemmings, esta hormiga, entiende que está cayendo y que debe aplicar la habilidad de **Floater** y no la de **Builder.**

Una vez que la hormiga disponga de los valores de las diferentes habilidades correspondientes con la función heurística y las feromonas, la decisión de escoger una es computada usando la siguiente ecuación.

$$p_{ij} = \begin{cases} \dfrac{\tau_{ij}^{\alpha}\eta_{ij}^{\beta}}{\sum_{u \in \mathcal{N}_i^k} \tau_{iu}^{\alpha}\eta_{iu}^{\beta}} & \text{if } j \in \mathcal{N}_i^k \\ 0 & \text{if } j \notin \mathcal{N}_i^k \end{cases}$$

Donde $N_i^k$ es el set de nodos factible conectados al nodo i, con respecto a la hormiga k. $T_{ij}$ representa el valor de la feromona de viajar desde el nodo i al nodo j y $\eta_{ij}$ representa el valor de la heurística de moverse desde el nodo i al nodo j. Finalmente α y β son dos parámetros que controlan la influencia de las feromonas y la heurística de la función empleada en el comportamiento de las hormigas. Por otra parte, si β >> α, las hormigas serán guiadas básicamente por la heurística de la función. Si pasa lo contrario, que β << α, entonces las hormigas seguirán los primeros caminos encontrados y por eso el algoritmo mostrará una rápida convergencia a los caminos subóptimos.

Finalmente, cuando cualquier hormiga termina su camino, bien porque ha alcanzado la comida, se ha quedado atrapada o el camino escogido le cause la muerte, la hormiga deshace el camino seguido actualizando las correspondientes feromonas con un valor que representa el grado de éxito de ese camino. Este valor es obtenido por la siguiente función:

$$F(Ind) = \frac{T(Ind) + A(Ind) + S(Ind)}{MaxTiempo + TotalAccionesDisponibles + TotalLemmings}$$

$\mathrm{T(Ind)} = \mathrm{MaxTiempo} - \mathrm{Tiempo(Ind)} * \mathrm{S(Ind)}$
$\mathrm{A(Ind)} = \mathrm{TotalAcccionesDisponibles} - \mathrm{AccionUsada(Ind)}$

Cuando la hormiga alcanza el nido, se olvidan del camino seguido y empiezan una nueva búsqueda.

## 2.8 Generación automática del nivel

### 2.8.1 Gramática libre de contexto

El juego adaptado crea los mapas a través de Tiles, por lo que para generar cada tile, se ha desarrollado una gramática libre de contexto para ello. En lenguaje formar, una gramática libre de contexto es una gramática formar en donde cada regla producida es de la forma V → w donde V es un símbolo no terminal y W una cadena de terminales y/o no terminales. El símbolo V puede siempre ser sustituido por W sin importar el contexto. Este tipo de gramáticas permiten diseñar eficientes algoritmos de análisis sintáctico que, para una cadena de caracteres dada, determinen como se puede generar la gramática.

Como cualquier gramática formal, una gramática libre de contexto se puede definir mediante la 4-tupla:

$G = (V_t, V_n, P, S)$ donde

- $V_t$ es un conjunto finito de terminales
- $V_n$ es un conjunto finito de no terminales
- $P$ es un conjunto finito de producciones
- $S \in V_n$ el denominado Símbolo Inicial
- los elementos de $P$ son de la forma

$$V_n \longrightarrow (V_t \cup V_n)^*$$

En este experimento se establecen las siguientes reglas:

***S*** → *Tile*

***Tile*** → *Suelos* | *Huecos* | *Destinos*

***Suelos*** → *Aire Suelo* | Aire Tierra | Pared Suelo | Pared Tierra | Caja Suelo | Caja Tierra | Destinos | Aire Aire

***Huecos*** → *Aire Aire*

***Destinos*** → *Destino Suelo*

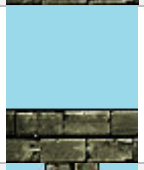
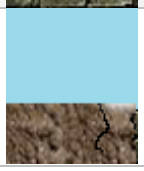
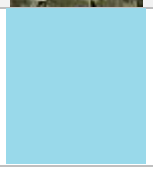
| Combinación de Tiles | | | |
|---|---|---|---|
| **Destino Suelo** |  | **Pared Tierra** |  |
| **Aire Suelo** |  | **Caja Tierra** |  |
| **Pared Suelo** |  | **Caja Suelo** |  |
| **Aire Tierra** |  | **Aire Aire** |  |

**Tabla 2 - Posibles Tiles**

### 2.8.2 Aplicación de la gramática en Python

Para aplicar esta gramática se ha descrito un algoritmo en Python que hace uso de esta gramática libre de contexto para generar aleatoriamente los tiles que constituyen el mapa. De igual modo, se establecen unas reglas para generar un mapa con cierto sentido común para evitar prácticamente el 100% de las veces se generen mapas que no pueden ser resueltos. El mapa consta de una dimensión de 20 tiles de ancho x 14 tiles de largo.

**Pseudocódigo**

```
nivel(x, y)=definirNivel(20,14)

“Iniciamos el cielo inicial por donde caen los lemmings”
para i=0 hasta 20
      tile(Arriba, Abajo) = generaTile (Huecos)
      nivel(0,i) = Tile(Arriba)
      nivel(1,i) = Tile(Abajo)

“Rellenamos el resto de Tiles del mapa”
para i=2 hasta 14 con incremento +2
      para j=0 hasta 20
            tile(Arriba, Abajo) = generaTile (Tile)
            nivel(j, i) = Tile(Arriba)
            nivel(j, i+1) = Tile(Abajo)
```

Aplicando este algoritmo se consigue generar un mapa de tiles 20 x 14 guardándose en el array nivel(x,y). Una vez generado este array, se itera sobre él para recorrer cada Tile y generar dos xml para poder almacenar los datos del mapa de forma legible. Más tarde se utilizan en la interfaz desarrollada como input para elaborar el nuevo mapa y en el algoritmo diseñado para buscar el mejor path. En el anexo se incluye el código Python con el que se realiza la generación aleatoria del mapa y transformación en XML.

# 3 Pruebas

## *3.1 Resultados para Lemmings*

### 3.1.1 Implementación de ACO para Lemmings

En esta sección se expondrá como se ha implementado el algoritmo de colonia de hormigas orientado a la interfaz adaptada del juego de los Lemmings. Tal como se muestra en el estudio [4], donde se realiza un experimento para medir la eficiencia de los algoritmos GA y ACO, podemos observar la configuración inicial para ambos algoritmos.

| Parámetros | Valor |
|---|---|
| **Tamaño de población** | 100 |
| **Max. Longitud del genotipo** | 20 |
| **Generaciones** | 500 |
| **# Offsprings** | 1 |
| **Crossover Rate** | 90% |
| **Crossover Type** | One point |
| **Mutation Rate** | 1% |
| **Elitism** | No |

**Tabla 3 - Configuración para Algoritmo Genético**

| Parámetros | Valor |
|---|---|
| **# Hormigas** | 100 |
| **# Pasos** | 500 |
| **Evaporation Rate** | 1% |
| **α** | 1 |
| **β** | 1 |

**Tabla 4 - Configuración para Algoritmo de Colonia de Hormigas**

El experimento se repitió 50 veces y se han analizado ambos algoritmos con heurística aleatoria y heurística de sentido común. Los resultados para los diferentes paths encontrados utilizando ambos algoritmos se refleja en esta tabla.

| Nivel | Complejidad | Algoritmo Genético | ACO (Hormiga aleatoria) | ACO (Sentido común) |
|---|---|---|---|---|
| **1** | Fácil | 3219 | 3868 | 2516 |
| **2** | Fácil | 12629 | 4463 | 4042 |
| **3** | Fácil | 2364 | 446 | 1330 |
| **4** | Fácil | 649 | 2128 | 2560 |
| **5** | Media | 370 | 1130 | 2487 |
| **6** | Media | 162 | 575 | 2520 |
| **7** | Media | 54 | 35 | 157 |
| **8** | Media | 1 | 12 | 228 |
| **9** | Media | 7 | 348 | 326 |
| **10** | Difícil | 2 | 15 | 32 |
| **11** | Difícil | 0 | 3 | 7 |
| **12** | Difícil | 2 | 18 | 62 |
| **13** | Difícil | 164 | 26 | 35 |
| **14** | Difícil | 1 | 4 | 23 |

**Tabla 5 - Resultados de la simulación**

Estos resultados experimentales concluyen que ambos algoritmos pueden ser aplicados para buscar posibles soluciones. Para los niveles con dificultad media-avanzada se observa que el mejor método es la solución por algoritmo genético dado que no utiliza la complejidad del nivel para generar individuos, al contrario que el algoritmo de colonia de hormigas.

Por otro lado, ACO utiliza la información del terreno para aplicar diferentes habilidades por lo que los resultados son mejores. El hecho de conocer información sobre el nivel es muy importante para crear métodos automáticos para solución de niveles.

## *3.2 Resultados para la interfaz propuesta*

Para elaborar los posibles resultados hemos de realizar primero los pasos previos como son la generación de los mapas de manera aleatoria, el análisis de dichos algoritmos sobre los mapas generados y las conclusiones finales acerca de los resultados.

### 3.2.1 Generación de los niveles de prueba en Python

Para interpretar el código Python que se ha diseñado (incluido en el anexo), se ha escogido la Shell interactiva de IPython, el cual ha ayudado a la detección de errores con colores y resaltados de línea.

Una vez pegado y ejecutado el código en la Shell, se genera automáticamente 2 ficheros en XML. Uno para ser posteriormente analizado por el Algoritmo propuesto y otro para ser traducido por la interfaz desarrollada.

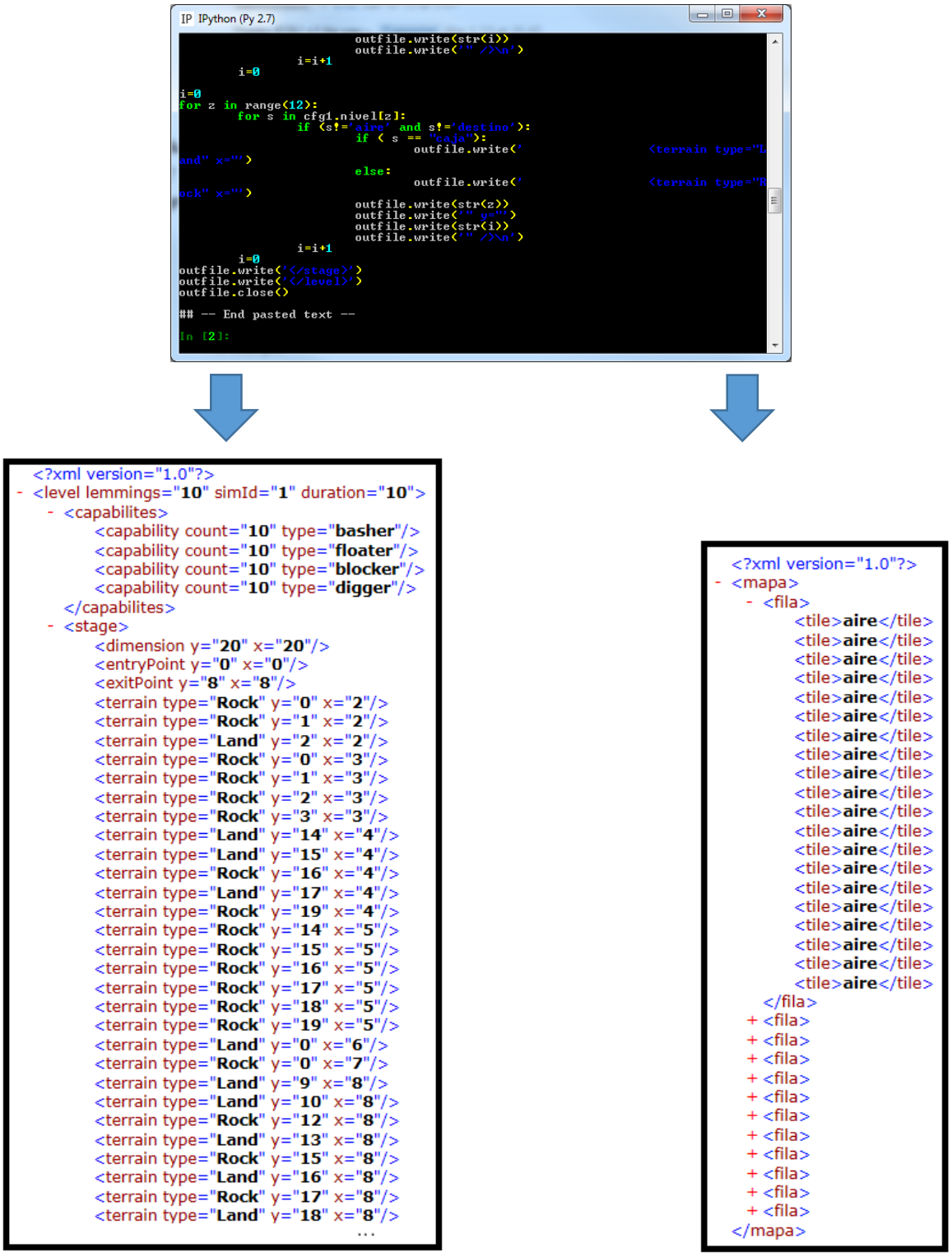



**Ilustración 13 – Gráfico de generación del nivel**

Una vez generados ambos XML, se analiza el primero con el Algoritmo propuesto para encontrar el mejor path y generar otro XML que sirva como input para la interfaz desarrollada y poder realizar una simulación del mapa generado.

### 3.2.2 Implementación de ACO para la interfaz adaptada

#### *3.2.2.1.1 Aplicación de ACO vs GA sobre la interfaz*

Se ha tomado como base un algoritmo desarrollado por *Antonio González* para la resolución de niveles para el juego de Lemmings. Se ha adaptado para la interfaz en la que vamos a realizar el experimento. Se han eliminado algunas habilidades de los Lemmings y se han dejado para el experimento las correspondientes a basher, digger, floater y blocker. Una vez se ejecuta el algoritmo, entre los paths guardados como posibles soluciones se elige el más eficiente y se traducen los pasos a un archivo XML para poder usarlo de input en la interfaz del juego.

**Pseudocódigo**

```
"Lemming Step"
cont=0
pos(x,y)=calculaPosLemming()
acciones=calculaAccionesPosibles(pos)
accion=eligeAccionAleatoria(acciones)

Si (ejecutaAccion(acción) == OK)
      planDeAcciones.add(acción)
Sino
      Mientras
            Si (ejecutaAccion(acción_actual) == FAIL)
                  cont+1
            Si (cont == 3)
                  resetHormiga()
                  break

"Se comprueba si el Lemming ha llegado al destino o superado los límites"
pos(x,y)=incrementaPosLemming()
Si (pos(x,y) == final) o (pos(x,y) == fuera_limites)
      siguienteHormiga()
```

Como variables iniciales se han utilizado las mismas que en el experimento mencionado antes.

| Parámetros | Valor |
|---|---|
| **# Hormigas** | 100 |
| **# Pasos** | 500 |
| **Evaporation Rate** | 1% |
| **α** | 1 |
| **β** | 1 |

**Tabla 6 - Configuración para Algoritmo de Colonia de Hormigas**

| Parámetros | Valor |
|---|---|
| **Tamaño de población** | 100 |
| **Max. Longitud del genotipo** | 20 |
| **Generaciones** | 500 |
| **# Offsprings** | 1 |
| **Crossover Rate** | 90% |
| **Crossover Type** | One point |
| **Mutation Rate** | 1% |
| **Elitism** | No |

**Tabla 7 - Configuración para Algoritmo Genético**

Como input del mapa a analizar se ha tomado el XML generado en el paso anterior. Después de lanzar la ejecución sobre el mapa generado para ambos algoritmos, se han obtenido diferentes posibles paths.

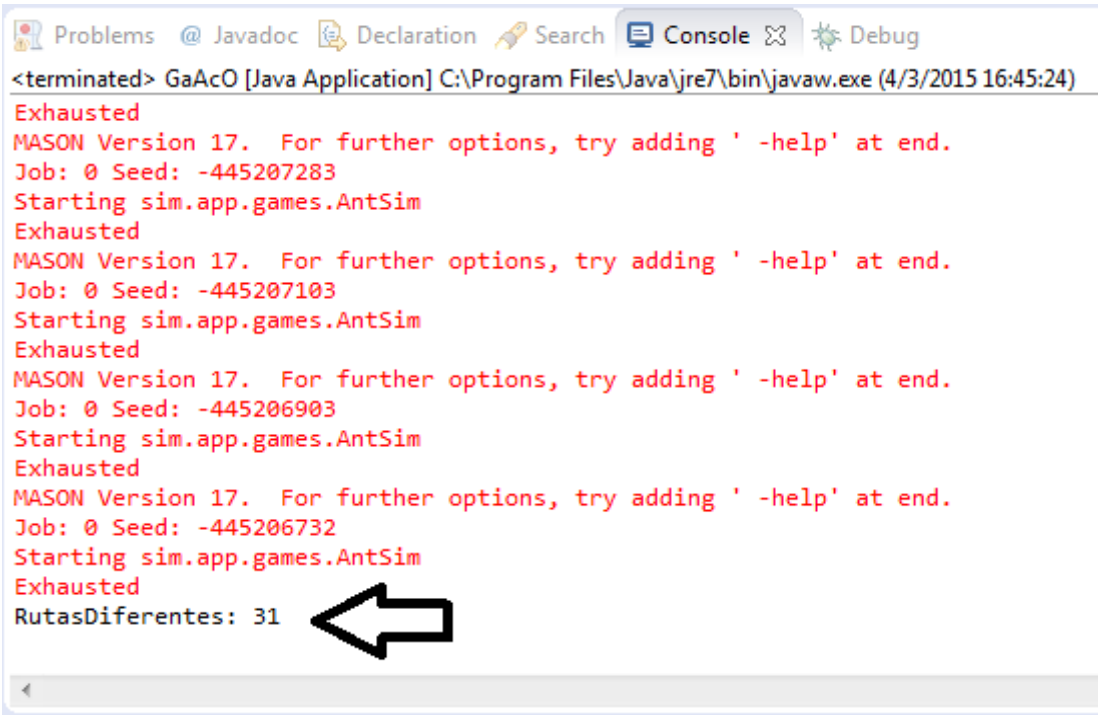



**Ilustración 14 – Simulación mediante ACO**

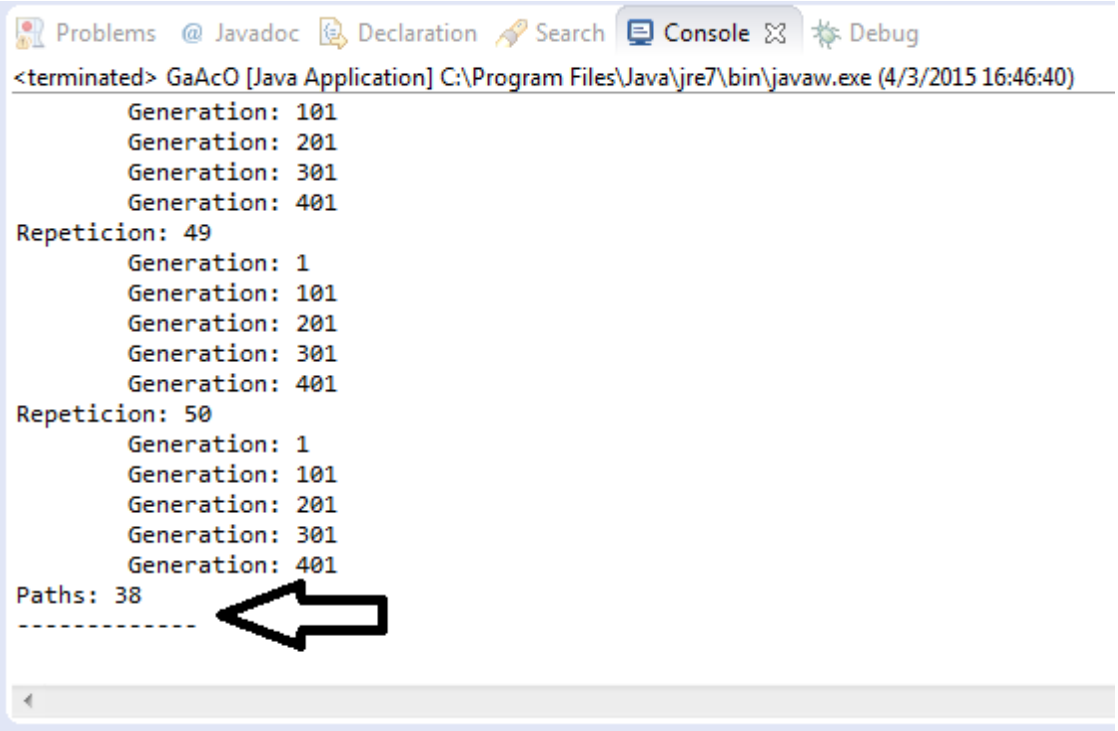



**Ilustración 15 – Simulación mediante GA**

A la hora de analizar el mejor path nos centramos en el algoritmo de ACO. Este algoritmo genera dos archivos. Un archivo de texto con todas las rutas posibles hasta la salida y otro archivo traducido a XML con la ruta más óptima para insertarlo en la interfaz de juego.

Para decidir que Path es el más óptimo se ha escogido el mejor en base al menor número de acciones necesarias para llegar del punto de inicio al punto de salida.

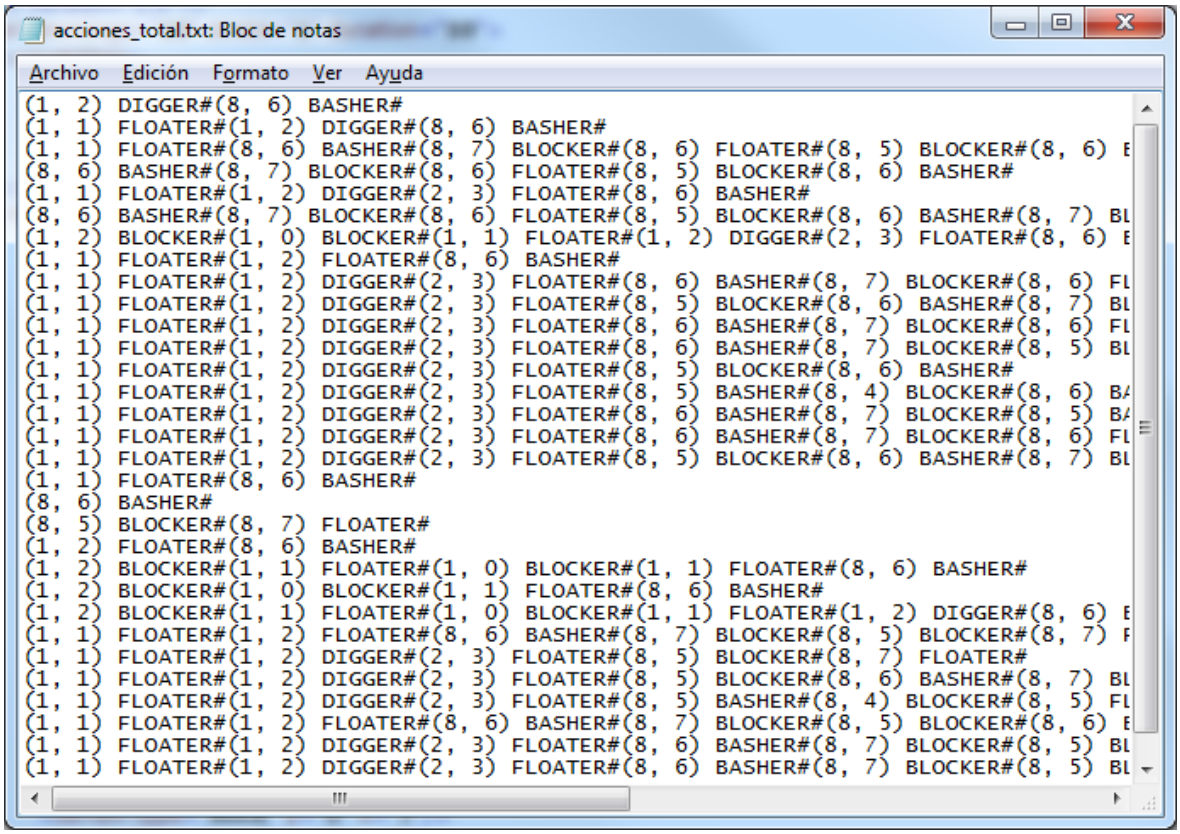



**Ilustración 16 – Paths totales**

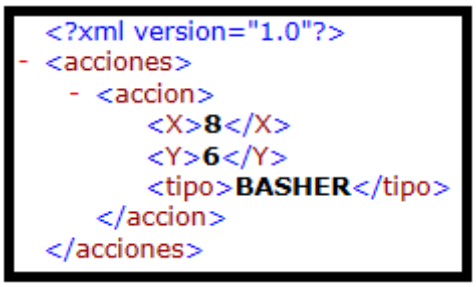



**Ilustración 17 – Path Óptimo**

Si usamos como input el mapa generado y las acciones a tomar, podemos hacer una simulación de la resolución del nivel.

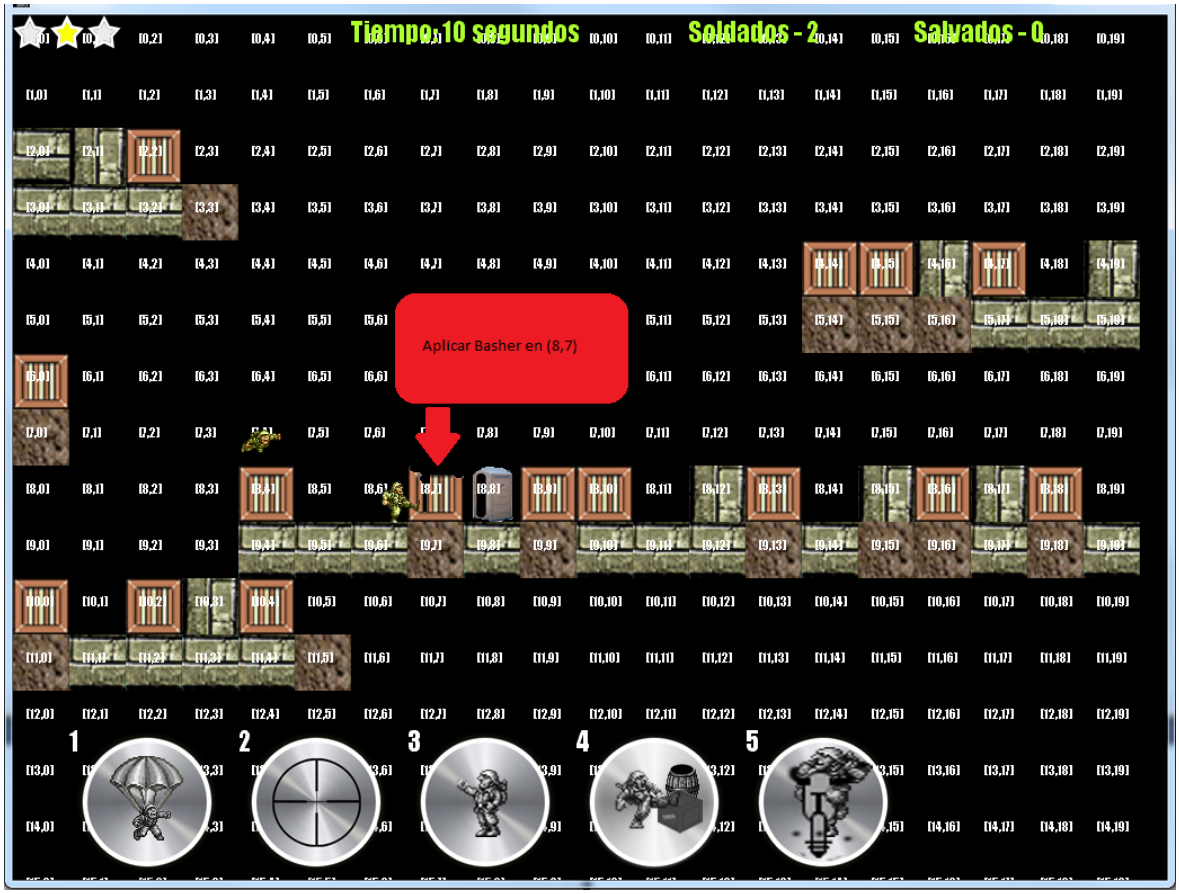



**Ilustración 18 – Simulación del nivel generado**

Como se puede observar, el mapa se ha generado correctamente y las acciones a realizar por los individuos les llevan a completar el nivel satisfactoriamente.


**Ilustración 19 – Final del nivel**

El experimento se va a repetir 50 veces para cada nivel y se han analizado ambos algoritmos. Los resultados para los diferentes paths encontrados se refleja en esta tabla.

| Nivel | Complejidad | Algoritmo Genético | ACO |
|---|---|---|---|
| **1** | Aleatoria | 38 | 31 |
| **2** | Aleatoria | Sin solución | Sin solución |
| **3** | Aleatoria | Sin solución | Sin solución |
| **4** | Aleatoria | Sin solución | Sin solución |
| **5** | Aleatoria | Sin solución | Sin solución |
| **6** | Aleatoria | Sin solución | Sin solución |
| **7** | Aleatoria | Sin solución | Sin solución |
| **8** | Aleatoria | Sin solución | Sin solución |
| **9** | Aleatoria | 72 | 23 |
| **10** | Aleatoria | Sin solución | Sin solución |
| **11** | Aleatoria | Sin solución | Sin solución |
| **12** | Aleatoria | Sin solución | Sin solución |
| **13** | Aleatoria | 26 | 21 |

**Tabla 8 - Resultados de la simulación**

No tiene sentido realizar muchas más simulaciones debido a que la complejidad de los mapas generados es aleatoria y no está categorizado según una complejidad como pasaba en el anterior experimento. Esto hace que muchas ejecuciones fallen a la hora de generar un mapa

que tenga solución. De hecho, para evitar tener que ejecutar el algoritmo un número elevado de veces, se han modificado manualmente algunos mapas generados para conseguir que tuvieran una solución plausible.

En los resultados positivos, ambos algoritmos pueden ser aplicados para buscar posibles soluciones. Por otro lado, ACO, ha demostrado obtener mejores resultados que algoritmos genéticos porque utiliza la información del terreno y saca una ligera ventaja. Además en la ejecución manual del algoritmo, ACO es mucho más rápido.

| Tiempos de ejecución (50 iteraciones) | |
|---|---|
| **ACO** | 20 segundos |
| **GA** | 90 segundos |

**Tabla 9 – Tiempos de ejecución**

# 4 Conclusiones y trabajo futuro

## *4.1 Conclusiones*

El objetivo de este trabajo es implementar un algoritmo para generar automáticamente niveles aleatorios e implementar un algoritmo ACO para conseguir resolverlos en la interfaz propuesta adaptada del juego de Lemmings. El experimento realizado muestra que, para un mapa generado automáticamente es muy probable que se genere un mapa sin solución aun aplicando ciertas reglas de sentido común para reducir la aleatoriedad del mapa. Para los mapas generados con posibles soluciones se ha demostrado que usando ACO se encuentra solución en prácticamente todos los niveles. La cantidad de soluciones depende directamente de la complejidad del mapa y las variables utilizadas.

La comparación de ACO y GA nos ha llevado a la conclusión de que ambos algoritmos pueden ser aplicados aunque según los datos podemos observar que ACO obtiene mejores resultados.

## *4.2 Trabajo futuro*

Como trabajo futuro se propone un análisis de ACO contra diferentes algoritmos de resolución de pantallas para comprobar la influencia de los parámetros usados y fijando diferentes recursos como puede ser el tiempo de resolución máximo, espacio del mapa… y así poder comparar que algoritmo es más eficiente a la hora de encontrar una solución a un mapa generado automáticamente.

# 4. Bibliografía


[1] http://es.wikipedia.org/wiki/Algoritmo_genético

[2] http://es.wikipedia.org/wiki/Computación_evolutiva

[3] http://es.wikipedia.org/wiki/Género_de_videojuegos

[4] A. Gonzalez-Pardo, F. Palero, and D. Camacho. An empirical study on collective intelligence algorithms for video games problem-solving. Computing and Informatics, In press, 2014

[5] http://es.wikipedia.org/wiki/Gramática_libre_de_contexto

[6] R. Sánchez-Pelegrín, M. A. Gómez-Martín y B. Díaz-Agudo, «A CBR module for a strategy videogame,» de Workshop on Computer Gaming and Simulation International Conference on Case-Based Reasoning, 2005.

[7] . B. D. Bryant y R. Miikulainen, «Acquiring visibly intelligent behavior with example-guided neuroevolution,» de Proceedings of the Twenty-Second National Conference on Artificial Intelligence, Menlo Park, CA, 2007.

[8] M. McPartland y M. Gallagher, «Creating a Multi-Purpose First Person Shooter Bot with,» de IEEE Symposium on Computational Intelligence and Games (CIG08), 2008.

[9] N. Cole, S. J. L. Miles y C. Miles, «Using a Genetic Algorithm to Tune First-Person Shooter Bots,» IEEE, pp. 139-145, 2004.

[10] C. Blum and D. Merkle. Swarm Intelligence. Introduction and Applications. Springer Publishing Company, Incorporated, 1 edition, 2008.

[11] J. Coldridge and M. Amos. Genetic algorithms and the art of zen. Technical report, Manchester Metropolitan University, 2010.

[12] J. McCarthy. The computer game Lemmings can serve as a new Drosophila for AI research

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver ,Alex Graves ,Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. Playing Atari with Deep Reinforcement Learning

[14] http://es.wikipedia.org/wiki/Redes_neuronales_convolucionales

# 5. Anexos

## Algoritmo Python para generar el mapa y XML

```
import random
import os
from collections import defaultdict
class GeneraNiveles(object):
    def __init__(self):
        self.tokens = defaultdict(list)
      self.destino=0
      self.aire=0
      self.nivel=defaultdict(list)

    def addRegla(self, token, valor):
        reglas = valor.split('|')
        for tokens in reglas:
            self.tokens[token].append(tuple(tokens.split()))

    def genRandom(self, tok, num):
        aux = ''

      if self.aire==1:
            tok_aleat = random.choice(self.tokens['Huecos'])
      else:
            tok_aleat = random.choice(self.tokens[tok])

      if "destino" in tok_aleat and self.destino==1:
            while True:
                  tok_aleat = random.choice(self.tokens[tok])
                  if "destino" not in tok_aleat:
                        break
      if "destino" in tok_aleat and self.destino==0:
            self.destino=1
      if tok_aleat[0]=='aire' and tok_aleat[1]=='aire' in tok_aleat and
self.aire==0 and num > 1:
            self.aire=1

      # De esta manera sólo permitimos que se genere un destino por nivel

        for simbolo in tok_aleat:
            if simbolo in self.tokens:
                aux += self.genRandom(simbolo, num)
            else:
                aux += simbolo + ' '

        return aux


cfg1 = GeneraNiveles()
cfg1.addRegla('Tile', 'Suelos')
cfg1.addRegla('Suelos', 'aire suelo | aire suelo_taladro | pared suelo |
pared suelo_taladro | caja suelo | caja suelo_taladro | Destinos | aire
aire')
cfg1.addRegla('Huecos', 'aire aire')
```

```
cfg1.addRegla('Destinos', 'destino suelo | Suelos')

while True:
      cont=0
      cfg1.destino=0
      cfg1.aire=0
      cfg1.nivel=defaultdict(list)
      """Inicializamos el cielo inicial """
      for i in range(20):
            res = cfg1.genRandom('Huecos', 1)
            aux = res.split(" ")
            cfg1.nivel[0].append(aux[0])
            cfg1.nivel[1].append(aux[1])

      """ Seguimos con el resto de Tiles del mapa"""
      for j in range(2,14,4):
            for i in range(20):
                  res = cfg1.genRandom('Tile', j)
                  aux = res.split(" ")
                  cfg1.nivel[j].append(aux[0])
                  cfg1.nivel[j+1].append(aux[1])
            cfg1.aire=0

      # Obligamos a que el primer pixel sea suelo
      cfg1.nivel[2].pop(0)
      cfg1.nivel[2].insert(0, 'suelo')

      for j in range(4,12,4):
            for i in range(20):
                  res = cfg1.genRandom('Tile', j)
                  aux = res.split(" ")
                  cfg1.nivel[j].insert(0,aux[0])
                  cfg1.nivel[j+1].insert(0,aux[1])
            cfg1.aire=0

      for i in range(20):
            if "destino" in cfg1.nivel[i]:
                  cont = 1
      if cont==1:
            break

# Tranformacion en lenguage XML

dir = 'C:\Mapa'
try:
    os.makedirs(dir)
except OSError:
    pass # ya existe
outfile = open(os.path.join(dir, 'nivel.xml'), 'w')

outfile.write('<mapa>\n')
for z in range(12):
      outfile.write('   <fila>\n')
      for s in cfg1.nivel[z]:
            outfile.write('         <tile>')
            outfile.write(s)
            outfile.write('</tile>\n')
      outfile.write('   </fila>\n')
outfile.write('</mapa>')
outfile.close()
```

```
outfile = open(os.path.join(dir, 'nivel2.xml'), 'w')
outfile.write('<level duration="10" simId="1" lemmings="10">\n')
outfile.write('   <capabilites>\n')
outfile.write('         <capability type="basher" count="10" />\n')
outfile.write('         <capability type="floater" count="10" />\n')
outfile.write('         <capability type="blocker" count="10" />\n')
outfile.write('         <capability type="digger" count="10" />\n')
outfile.write('         </capabilites>\n')
outfile.write('   <stage>\n')
outfile.write('         <dimension x="20" y="20" />\n')
outfile.write('         <entryPoint x="0" y="0" />\n')
i=0
for z in range(12):
      for s in cfg1.nivel[z]:
            if (s=='destino'):
                  outfile.write('          <exitPoint x="')
                  outfile.write(str(z))
                  outfile.write('" y="')
                  outfile.write(str(i))
                  outfile.write('" />\n')
            i=i+1
      i=0

i=0
for z in range(12):
      for s in cfg1.nivel[z]:
            if (s!='aire' and s!='destino'):
                  if ( s == "caja"):
                        outfile.write('               <terrain type="Land"
x="')
                  else:
                        outfile.write('               <terrain type="Rock"
x="')
                  outfile.write(str(z))
                  outfile.write('" y="')
                  outfile.write(str(i))
                  outfile.write('" />\n')
            i=i+1
      i=0
outfile.write('</stage>')
outfile.write('</level>')
outfile.close()
```