

Comparación del rendimiento de algoritmos genéticos en su aplicación a videojuegos



ULPGC
Universidad de
Las Palmas de
Gran Canaria



Aleksander Borysov Ravelo - Grado en Ingeniería Informática 06/2023

Tutorizado por:

- Oliverio Jesús Santana Jaria
- Jose Daniel Hernández Sosa

• ESTADO ACTUAL Y OBJETIVOS INICIALES:

Situación actual del tema relacionado con el TFT

Motivación

Los algoritmos genéticos simulan el proceso de evolución que se da en la naturaleza, haciendo competir distintas posibilidades entre sí para alcanzar una solución óptima a un determinado problema. Se trata de un amplio campo de investigación con importantes aplicaciones en la ingeniería y la tecnología, por lo que su conocimiento es de gran interés para los estudiantes que terminan la titulación y deben enfrentarse a un mercado laboral en el que cada vez es más común la necesidad de implementar aplicaciones prácticas de inteligencia artificial y ciencia de datos.

Objetivos

Como caso de estudio, se comprobará el comportamiento de distintos algoritmos genéticos aprendiendo a jugar a un videojuego. Una vez seleccionado el videojuego más apropiado para esta tarea, se seleccionarán e implementarán una serie de algoritmos genéticos en lenguaje Lua, incorporándolos como “plugin” a un emulador de juegos de consolas que se ejecutará en un ordenador. Los algoritmos comenzarán entonces a manejar el videojuego por ellos mismos y aprenderán de forma evolutiva. Tras el periodo de entrenamiento, y con los datos sobre su rendimiento extraídos, se realizará una comparación entre ellos y se estudiará la posibilidad de incluir cambios específicos que permitan mejorar su funcionamiento.

- Conocer los algoritmos genéticos y los últimos avances en el campo.
- Implementar algoritmos genéticos capaces de controlar videojuegos.
- Hacer pruebas de rendimiento sobre los prototipos desarrollados y, con espíritu crítico, saber separar los buenos resultados de los malos.

• JUSTIFICACIÓN DE LAS COMPETENCIAS ESPECÍFICAS CUBIERTAS:

CP04: Capacidad para conocer los fundamentos, paradigmas y técnicas propias de los sistemas inteligentes y analizar, diseñar y construir sistemas, servicios y aplicaciones informáticas que utilicen dichas técnicas en cualquier ámbito de aplicación

Resulta evidente que la exploración del algoritmo NEAT y sus variantes ya involucra conocimiento sobre sistemas inteligentes. Así como aprender cómo se podría aplicar en un videojuego.

• DESARROLLO:

En una fase preliminar, se seleccionarán y se justificarán los algoritmos a comparar. Además, se elegirá el videojuego más apropiado para realizar el estudio.

En una segunda fase, se implementarán los algoritmos en Lua utilizando la interfaz que ofrece el emulador de videojuegos de consola Bizhawk. La implementación accederá a la memoria reservada por el juego en tiempo de ejecución. El algoritmo reaccionará a lo que lea del estado del videojuego en cada instante y pulsará los botones que manejan al personaje de forma virtual. Se plantearán diferentes prototipos que se irán analizando para guiar el desarrollo de versiones más completas.

En la fase final, se elegirán las métricas de rendimiento y se obtendrán lecturas del rendimiento de cada algoritmo en su tarea. Se tendrán en cuenta el número de generaciones, los tiempos de ejecución y la efectividad a la hora de completar, al menos parcialmente, el/los niveles del videojuego objeto de estudio. A partir de esta información, se determinará el mejor algoritmo para este ámbito y se valorará la posibilidad de incluir alguna mejora dentro del contexto del videojuego estudiado.

Familiarizándonos con el entorno

Antes de empezar con el trabajo tenemos que familiarizarnos con el material y las herramientas que vamos a usar durante el mismo.

Primero atenderemos a la instalación de Bizhawk, el emulador que utilizaremos para ejecutar el juego en el ordenador. Seleccionamos Bizhawk porque permite la implementación de *plugins* escritos en Lua que pueden interactuar con el juego en ejecución. El *plugin* base que utilizaremos es un código del algoritmo NEAT ya escrito para el juego Super Mario Bros (NES).

Este plugin fue desarrollado por Sethbling, un creador de contenido en YouTube ^{1, 2}.

Para que el emulador funcione en Windows hay que instalar un paquete de pre-requisitos que podemos encontrar en el siguiente enlace [Releases · TASEmulators/BizHawk-Prereqs \(github.com\)](https://github.com/TASVideos/BizHawk-Prereqs).

Por otro lado, la instalación del emulador es simple. Simplemente descargamos los binarios desde la página oficial. [Bizhawk/ReleaseHistory - TASVideos](https://github.com/TASVideos/BizHawk/ReleaseHistory). En este trabajo se utiliza la versión 2.8.0. La descarga incluye todos los archivos necesarios y ya se puede ejecutar Bizhawk.

Uno de los tutores tiene el cartucho original. Pero no contamos con las herramientas para fabricar una ROM propia. Aceptando este hecho creemos que al no haber beneficio económico, quedándose todo en el plano académico podemos estar tranquilos.

El fichero fuente original NEATEvolve.lua utilizaba un nombre de la ROM diferente. Además, dichos nombres estaban *hardcodeados*. La primera tarea es refactorizar la detección de nombre para la ROM y cambiarlo al nombre de nuestra ROM que puede ser uno diferente al que usó el autor original del código.

```

if gameinfo.getromname() == "Super Mario World (USA)" then
    Filename = "DP1.state"
    ButtonNames = {
        "A",
        "B",
        "X",
        "Y",
        "Up",
        "Down",
        "Left",
        "Right",
    }
elseif gameinfo.getromname() == "Super Mario Bros." then
    Filename = "SMB1-1.state"
    ButtonNames = {
        "A",
        "B",
        "Up",
        "Down",
        "Left",
        "Right",
    }
end

```

El nombre exacto de la ROM podemos obtenerlo ejecutando la ROM en el emulador y escribiendo el comando gameinfo.getromname() en la consola.

Abrir la ROM: File -> Open ROM

Abrir la consola: Tools -> Lua Console

```

SuperMarioBrosROMName = "Super Mario Bros. (W) [!]"
SuperMarioBrosStateFile = "SMB1-1.state"

elseif gameinfo.getromname() == SuperMarioBrosROMName then
    Filename = SuperMarioBrosStateFile
    ButtonNames = {
        "A",
        "B",
        "Up",
        "Down",
        "Left",
        "Right",
    }
end

```

Extraemos el nombre en una variable global en todas las ocurrencias.

Se puede observar una variable global SuperMarioBrosStateFile. Esta variable guarda el nombre del archivo de estado del juego que utilizará el algoritmo para cargar o resetear la partida.

Este archivo de estado lo fabricamos guardando el estado del juego con el emulador. En este caso queremos guardar el estado del juego al principio del primer nivel de Mario.

Abrimos la ROM de Mario y empezamos una partida. Podemos guardar el estado en File -> Save State -> Save Named State. Es preferible guardarla como un Named State porque podemos darle nombre y localizar el fichero más fácilmente.

```
-- For SMW, make sure you have a save state named "DP1.state" at the beginning of a level,
-- and put a copy in both the Lua folder and the root directory of BizHawk.
```

Tal y como pone en las instrucciones al principio del fichero fuente original, este fichero de estado debe estar presente en el directorio del script y en el directorio raíz de BizHawk (junto con el ejecutable).

Ya estamos listos para ejecutar el juego con el script.

Abrimos la consola: Tools -> Lua Console -> Open script. Seleccionamos NEATEvolve.lua

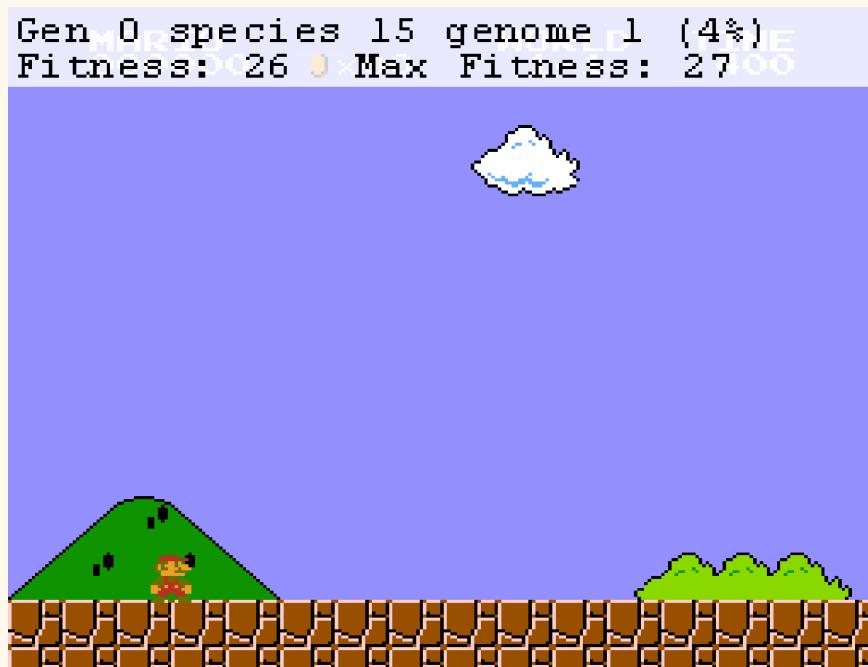
La primera tarea con el script funcionando ya es conseguir un entrenamiento que complete el nivel. Así como lograr entender lo que significan los distintos elementos de la IU. Además tenemos que averiguar si podemos guardar y cargar el estado actual del algoritmo, para poder reanudar el entrenamiento, acceder a las propiedades del entrenamiento, etc.

Un entrenamiento que complete el nivel en este juego puede tardar en terminar de uno a tres días, dependiendo de si la aleatoriedad acaba beneficiando al progreso.

Analicemos ahora la interfaz de usuario que genera el script.

Leyendo en orden:

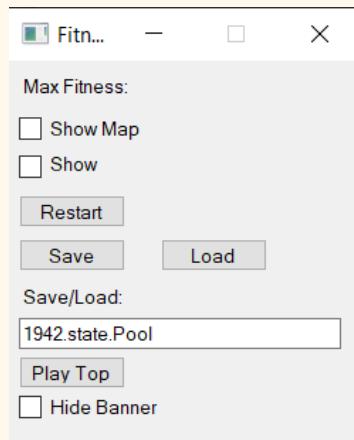
- Gen: indica la generación actual
- Species: indica la especie del individuo que se está evaluando
- Genome: indica el individuo que se está evaluando
- Progreso (%): indica el progreso de entrenamiento en la generación actual. Cuando llega al 100% se ha iterado por todos los individuos e inicia una nueva generación.
- Fitness: indica la puntuación del individuo actual
- Max Fitness: indica la puntuación máxima alcanzada por cualquier individuo durante el transcurso de las generaciones.



Por otro lado, queda explicar el proceso de guardado y de carga del estado del algoritmo.

Al inicio de cada generación se guarda en un fichero la información concerniente a cada uno de los individuos de la generación, estado del entrenamiento, etc. Estos ficheros tienen la nomenclatura siguiente: backup.GENERACION_ACTUAL.NOMBRE_ROM.state.Pool.

Para cargar estos ficheros hacemos uso de esta pequeña ventana que se abre al ejecutar el script NEATEvolve.lua en el emulador.



En ella hay varios controles que son importantes.

- Show Map: Muestra la configuración del escenario del juego codificada para mandarlo como entrada a la red.
- Show: Muestra los ratios de mutación del individuo actual (solo si Show Map está activado).
- Restart: Reinicia el entrenamiento desde la generación cero.
- Save: Guarda el estado actual del algoritmo en el fichero especificado en la entrada.
- Load: Carga el fichero de estado especificado en la entrada.
- Entrada: Se escribe la ruta relativa del fichero respecto a la ubicación del script.
- Play Top: Juega el jugador con más fitness.
- Hide Banner: Esconde el elemento de la IU principal (solo si Show Map está activado).

Con esto nos hemos terminado de familiarizar con el uso de la interfaz y los controles básicos del script.

Ahora realizamos una pequeña mejora en el código para acelerar el proceso de entrenamiento. Mientras entrenaba la primera vez, se podía observar que el algoritmo se pasaba mucho tiempo en la pantalla de muerte o *game over*. Es por eso por lo que ahora también se evalúa si el jugador está muerto como una condición para pasar al siguiente genoma.

```
local TimeoutBonus = Pool.currentFrame / 4
local hasTimedOut = Timeout + TimeoutBonus <= 0
local IsDead = IsDead()
if IsDead or hasTimedOut then
    local fitness = GetFitness()
    genome.fitness = fitness
```

Agregamos al bucle principal la comprobación de muerte.

```
function isDead()
    if gameinfo.getromname() == "Super Mario World (USA)" then
        return memory.readbyte(0x13e0) == 0xe
    elseif gameinfo.getromname() == SuperMarioBrosROMName then
        local playerState = memory.readbyte(0x000E)
        local dying = 0x0B
        local playerDies = 0x06
        return (playerState == dying or playerState == playerDies)
    end
end
```

La función accede a la memoria principal del juego y recupera de la dirección pertinente un byte que representa el estado actual del jugador. Si ese byte tiene valor 0x0B está ejecutando la animación de muerte. Si ese byte tiene valor 0x06 está muerto. Añadimos los dos por si los tiempos no dan para capturar la primera comprobación. En este caso conocemos la dirección de memoria gracias al mapa de memoria en Data Crystal³. Pero esto, como ya veremos más adelante, no será siempre así y tendremos que recurrir a procesos de ingeniería inversa para encontrar lo que nos interesa de la memoria.

Entendiendo el algoritmo

La red neuronal base consiste en tener una neurona por cada entrada y una neurona por cada salida.

Si a la red neuronal le entra una matriz 16x16, por ejemplo, en la capa de entrada habrá 256 neuronas. Esta entrada representaría la pantalla actual del juego, con la colocación correcta de los bloques, enemigos y otros elementos relevantes que queramos comunicar a la red.

En la capa de salida tenemos tantas neuronas como controles hayamos programado para el juego. En el caso de Super Mario Bros tenemos 6 controles.

```
ButtonNames = {
    "A",
    "B",
    "Up",
    "Down",
    "Left",
    "Right",
}
```

Salto, correr, arriba, abajo, izquierda, derecha. Y por lo tanto, tendremos 6 neuronas en la capa de salida.

Como evaluar el contenido de la pantalla en cada imagen del juego sería poco eficiente, el proceso de recogida de entradas de la pantalla se realiza cada 5 cuadros. La matriz de entrada se genera iterando sobre todas las direcciones de memoria de interés. Estas se pueden obtener con el mapa de memoria de Data Crystal o por medio de ingeniería inversa.

NEAT

El primer acercamiento que se tuvo para aprender el funcionamiento del algoritmo fue pobre. En el sentido de que se intentó leer el código fuente y comprender NEAT desde ahí. Se desperdicó bastante tiempo en algo que se solucionaría con leer el paper original del algoritmo. Las siguientes figuras han sido obtenidas del paper.⁴

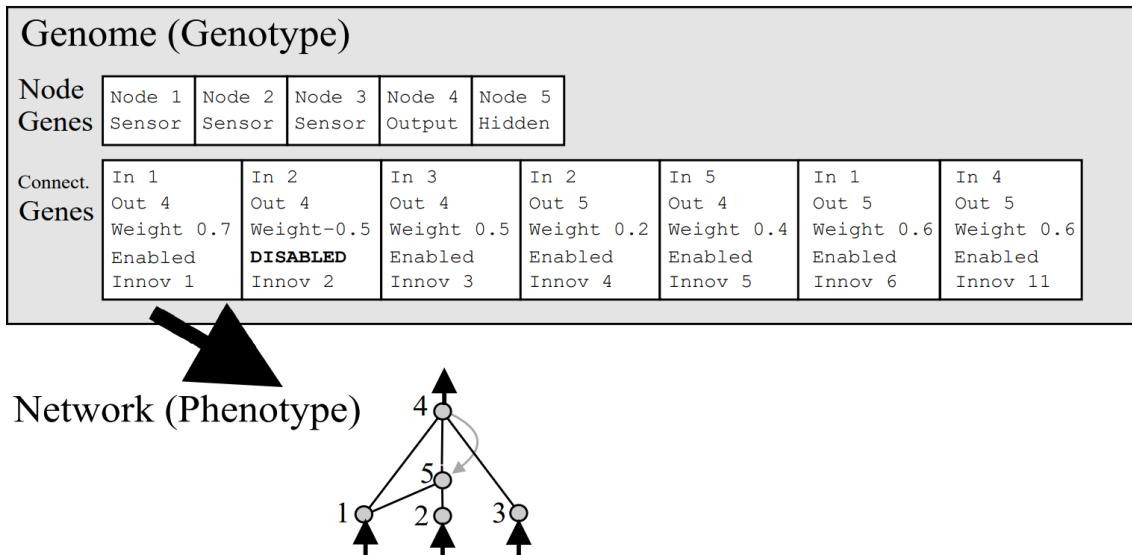


Figure 2: A genotype to phenotype mapping example. A genotype is depicted that produces the shown phenotype. There are 3 input nodes, one hidden, and one output node, and seven connection definitions, one of which is recurrent. The second gene is disabled, so the connection that it specifies (between nodes 2 and 4) is not expressed in the phenotype.

Pag 106

Para entender las siguientes explicaciones debemos definir una serie de términos.

- Fenotipo: El individuo que se somete a evolución, la red neuronal.
- Genotipo: Sinónimo de genoma. Es el conjunto de genes que codifican un individuo o fenotipo (red neuronal)

En este algoritmo definimos dos tipos de genes.

- Gen nodo: Representa una neurona de la red neuronal.
- Gen conexión: Representa una conexión en la red neuronal.

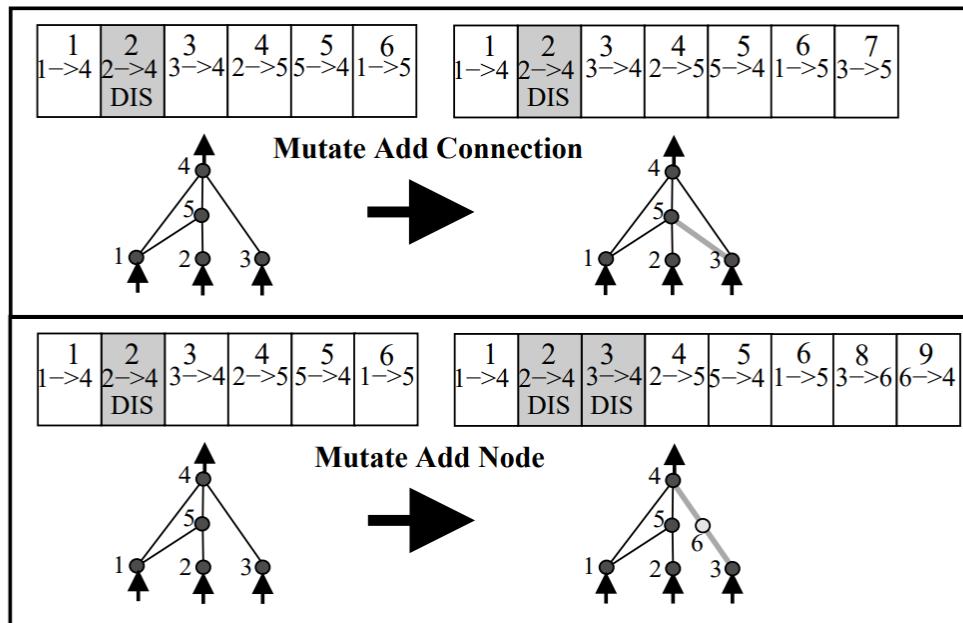


Figure 3: The two types of structural mutation in NEAT. Both types, adding a connection and adding a node, are illustrated with the connection genes of a network shown above their phenotypes. The top number in each genome is the *innovation number* of that gene. The innovation numbers are historical markers that identify the original historical ancestor of each gene. New genes are assigned new increasingly higher numbers. In adding a connection, a single new connection gene is added to the end of the genome and given the next available innovation number. In adding a new node, the connection gene being split is disabled, and two new connection genes are added to the end of the genome. The new node is between the two new connections. A new node gene (not depicted) representing this new node is added to the genome as well.

Pag 107

También tenemos dos mutaciones básicas, a las cuales el autor original añadió otras más.

- Mutación: Provocar un cambio estructural sobre el fenotipo.
- Añadir conexión: Añade una nueva conexión entre dos nodos. Su peso es aleatorio entre -2 y 2.
- Añadir nodo: Añade un nodo en lugar de una conexión y genera dos conexiones. La primera tiene peso 1 y la segunda el peso de la conexión original.
- Marcador histórico: Identifican el ancestro original de cada gen. A cada gen nuevo se le asigna un marcador histórico incremental. Este identificador es crucial para el funcionamiento del algoritmo.

Las mutaciones adicionales que el autor ha añadido son:

- Activar un enlace: Selecciona un gen conexión del genoma y lo activa.
- Desactivar un enlace: Selecciona un gen conexión del genoma y lo desactiva.
- Force bias: Muta una nueva conexión cuya neurona de entrada es forzosamente una neurona de la capa de entrada.
- Weight: Según una probabilidad, cambiamos el peso por un factor de step determinado o generamos un valor totalmente nuevo entre -2 y 2.

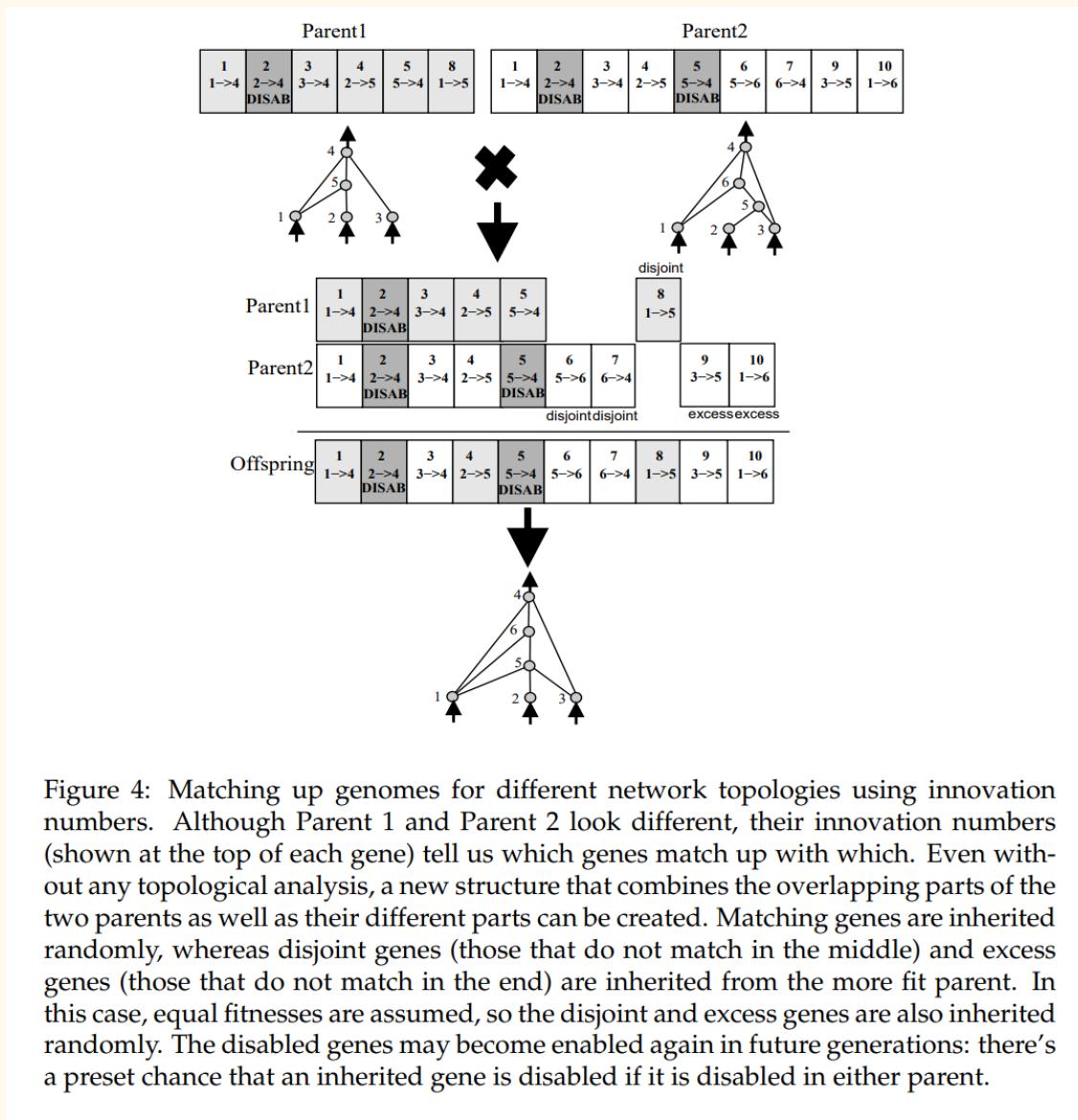


Figure 4: Matching up genomes for different network topologies using innovation numbers. Although Parent 1 and Parent 2 look different, their innovation numbers (shown at the top of each gene) tell us which genes match up with which. Even without any topological analysis, a new structure that combines the overlapping parts of the two parents as well as their different parts can be created. Matching genes are inherited randomly, whereas disjoint genes (those that do not match in the middle) and excess genes (those that do not match in the end) are inherited from the more fit parent. In this case, equal fitnesses are assumed, so the disjoint and excess genes are also inherited randomly. The disabled genes may become enabled again in future generations: there's a preset chance that an inherited gene is disabled if it is disabled in either parent.

Cuando se crea una mutación se le asigna un identificador incremental (Marcador histórico).

- Solo se pueden cruzar genes cuando el marcador histórico coincide. El cruce se realiza por defecto aleatoriamente. Es decir, se escogen de cualquiera de los dos padres.
- Si no coincide será una operación de disjoint (intercalar). Si el fitness de los dos padres es igual también se obtienen aleatoriamente. En caso contrario, prevalecen los genes del padre más performante.
- Excess (añadir al final). El criterio es igual que en la operación de intercalar. Si el fitness de los dos padres es igual será aleatorio, y si no serán heredados del padre más en forma.

Las innovaciones han de ser protegidas para que tengan el tiempo suficiente de ser optimizadas. Sin esta protección, una mutación de tipo añadir nodo podría desaparecer. Simplemente porque al momento de introducirla ya puede suponer un decremento en rendimiento de la red.

Para ello, introducimos el concepto de especie. Una especie es una división de la población en partes que comparten una topología similar y un rendimiento (fitness) similar. De este modo, la innovación se desarrolla en un entorno más pequeño sin el peligro de desaparecer al exponerse a toda la población a la vez.

Las especies se obtienen con la ayuda del coeficiente de distancia δ . Cuando la distancia es menor se asume que es de la misma especie. Si la distancia es muy lejana con todas las especies, es hora de crear una nueva especie.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}.$$

The number of excess and disjoint genes between a pair of genomes is a natural measure of their compatibility distance. The more disjoint two genomes are, the less evolutionary history they share, and thus the less compatible they are. Therefore, we can measure the compatibility distance δ of different structures in NEAT as a simple linear combination of the number of excess E and disjoint D genes, as well as the average weight differences of matching genes \bar{W} , including disabled genes:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \bar{W}. \quad (1)$$

The coefficients c_1 , c_2 , and c_3 allow us to adjust the importance of the three factors, and the factor N , the number of genes in the larger genome, normalizes for genome size (N can be set to 1 if both genomes are small, i.e., consist of fewer than 20 genes).

The distance measure δ allows us to speciate using a compatibility threshold δ_t . An ordered list of species is maintained. In each generation, genomes are sequentially placed into species. Each existing species is represented by a random genome inside the species from the *previous generation*. A given genome g in the current generation is placed in the first species in which g is compatible with the representative genome of that species. This way, species do not overlap.¹ If g is not compatible with any existing species, a new species is created with g as its representative.

As the reproduction mechanism for NEAT, we use *explicit fitness sharing* (Goldberg and Richardson, 1987), where organisms in the same species must share the fitness of their niche. Thus, a species cannot afford to become too big even if many of its organisms perform well. Therefore, any one species is unlikely to take over the entire population, which is crucial for speciated evolution to work. The adjusted fitness f'_i for organism i is calculated according to its distance δ from every other organism j in the population:

$$f'_i = \frac{f_i}{\sum_{j=1}^n \text{sh}(\delta(i, j))}. \quad (2)$$

The sharing function sh is set to 0 when distance $\delta(i, j)$ is above the threshold δ_t ; otherwise, $\text{sh}(\delta(i, j))$ is set to 1 (Spears, 1995). Thus, $\sum_{j=1}^n \text{sh}(\delta(i, j))$ reduces to the number of organisms in the same species as organism i . This reduction is natural since species are already clustered by compatibility using the threshold δ_t . Every species is assigned a potentially different number of offspring in proportion to the sum of adjusted fitnesses f'_i of its member organisms. Species then reproduce by first eliminating the lowest performing members from the population. The entire population is then replaced by the offspring of the remaining organisms in each species.²

Eligiendo otro título

A pesar de que el algoritmo funciona sobre el juego de Mario. Necesitamos tener más propiedad sobre el trabajo para justificar el TFG. Necesitamos encontrar otro juego.

La búsqueda se cierra sobre todo en el pequeño grupo de juegos que tienen disponible un mapa de RAM en la web de Data Crystal. Como criterio adicional, buscamos juegos que supongan un desmarque respecto al género de Mario. No queremos otro juego de plataformas.

Entre 1942 y Bomberman II, nos decantamos por el segundo. Porque Bomberman II era más familiar y también más desafiante. Hay que destacar que este juego es menos reactivo, requiere razonamiento y las acciones son más complejas que el salto y desplazamiento lateral de Mario.



Captura del juego en un video de Youtube

Para cambiar el foco del algoritmo tenemos que cambiar todo el proceso de obtención de inputs. Las direcciones de memoria y la disposición de los datos en la memoria cambian entre juego y juego.

Este fue el primer reto complicado, conseguir leer el mapa del nivel desde la memoria.

Tras un largo proceso de búsqueda tenemos la siguiente información.

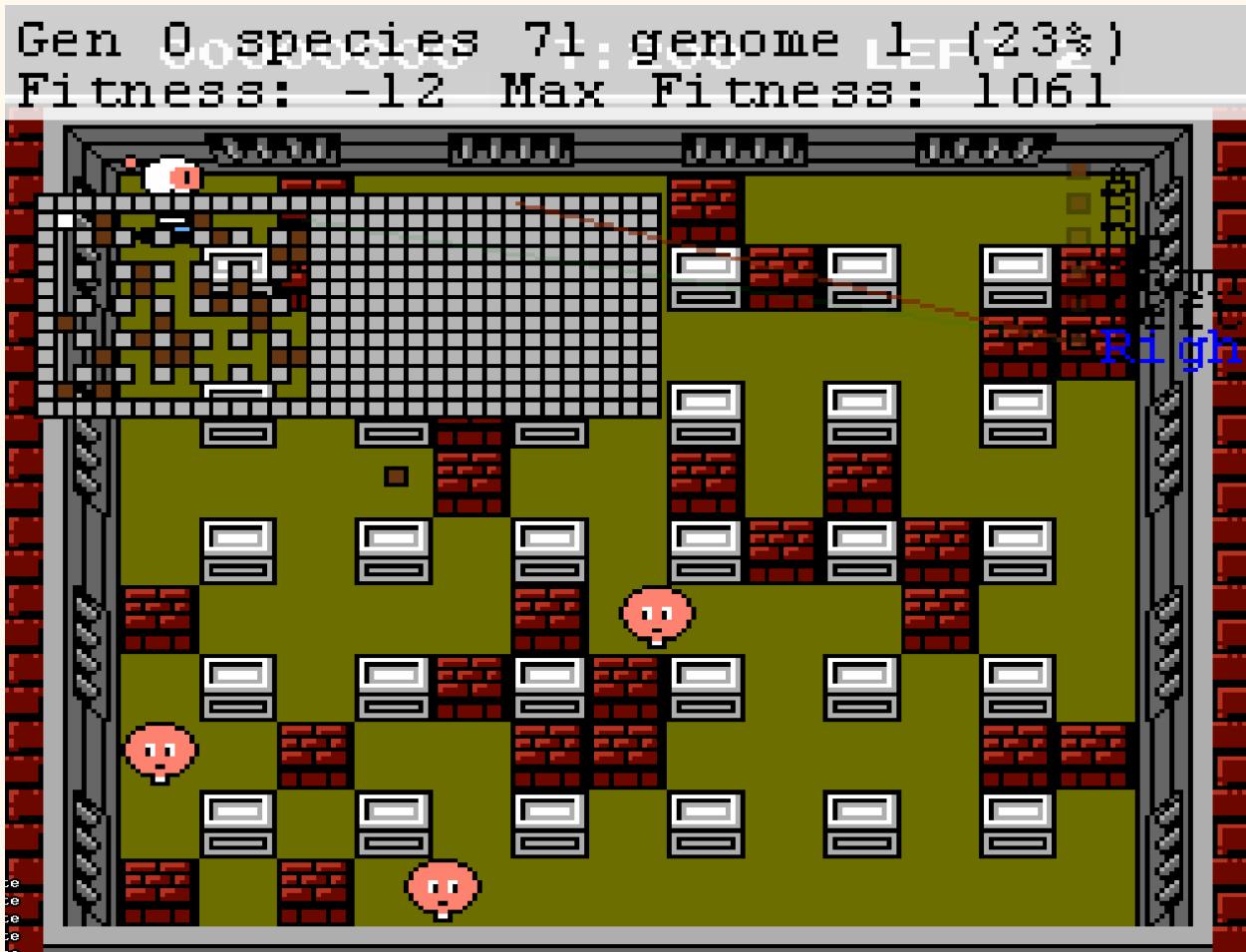
BOMBERMAN_ALIVE = 0x0069	/* Bomberman vivo o muerto */
BOMBERMAN_X = 0x0072	/* Posición de Bomberman en el eje X */
BOMBERMAN_Y = 0x0078	/* Posición de Bomberman en el eje Y */
WRAM_LEVEL_INIT = 0x62F3	/* Inicio del mapeo del nivel */
WRAM_LEVEL_END = 0x6492	/* Final del mapeo del nivel */
LEVEL_WIDTH = 31	/* Ancho del nivel (bloques) */
LEVEL_HEIGHT = 12	/* Altura del nivel (bloques) */

El contenido de la memoria entre el inicio del nivel y el final del nivel puede tener los siguientes valores -u otros que no resultan relevantes-.

AIR = 0x00	/* Aire (No hay nada) */
POWERUP = 0x01	/* Potenciador (Si se recoge aumenta las características del personaje) */
DOOR = 0x02	/* Puerta (Meta) */
BOMB = 0x10	/* Bomba colocada */
BREAKABLE_BLOCK = 0x20	/* Bloque destructible */
BREAKABLE_BLOCK_POWERUP = 0x21	/* Bloque destructible con potenciador dentro*/
BREAKABLE_BLOCK_DOOR = 0x22	/* Bloque destructible con puerta dentro */
UNBREAKABLE_BLOCK = 0x40	/* Bloque irrompible */
EXPLOSION = 0x80	/* Explosión ocurriendo en ese bloque */
BLOCK_BREAKING = 0xA0	/* Bloque rompiéndose */

NES System Bus - 0x10000 addresses																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
6210		FF														
6220		FF														
6230		FF														
6240		FF														
6250		00	00	00	00	00	00	01	01	01	FF	FF	FF	FF	FF	FF
6260		FF	00	00	00	FF	FF	FF	FF	FF	D8	4A	18	FF	FF	FF
6270		FF	FF	FF	FF	FF	00	00	00	FF	FF	FF	FF	FF	FF	95
6280		38	7B	FF	FF	FF	FF	FF	FF	OD	04	01	FF	FF	FF	FF
6290		FF	FF	FF	09	03	07	FF	FF	FF	FF	FF	FF	00	00	00
62A0		FF	FF	FF	FF	FF	FF	01	03	01	FF	FF	FF	FF	FF	FF
62B0		FF	80	80	80	FF	FF	FF	FF	FF	01	00	01	FF	FF	FF
62C0		FF	FF	FF	FF	FF	00	01	02	FF						
62D0		00	00	FF	00	00	00	01	8C	00						
62E0		00	00	70	FF	FF	FF	00	FF	FF	FF	FF	00	FF	FF	FF
62F0		FF	00	00	40	40	40	40	40	40	40	40	40	40	40	40
6300		40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
6310		40	40	40	40	00	00	00	00	00	00	00	00	00	00	00
6320		00	40	40	40	40	40	40	40	40	40	40	40	40	40	40
6330		40	40	40	40	00	40	00	40	00	40	00	40	21	40	40
6340		00	40	40	40	40	40	40	40	40	40	40	40	40	40	40
6350		40	40	40	40	00	20	00	00	00	00	00	00	00	00	00
6360		20	40	40	40	40	40	40	40	40	40	40	40	40	40	40
6370		40	40	40	40	00	40	00	40	00	40	00	40	20	40	40
6380		00	40	40	40	40	40	40	40	40	40	40	40	40	40	40
6390		40	40	40	40	20	00	00	00	00	00	00	20	00	00	20
63A0		00	40	40	40	40	40	40	40	40	40	40	40	40	40	40
63B0		40	40	40	40	00	40	00	40	20	40	20	40	00	40	40
63C0		20	40	40	40	40	40	40	40	40	40	40	40	40	40	40
63D0		40	40	40	40	00	00	20	00	00	20	00	00	00	20	40
63E0		00	40	40	40	40	40	40	40	40	40	40	40	40	40	40
63F0		40	40	40	40	00	40	00	40	00	40	00	40	22	40	20
6400		00	40	40	40	40	40	40	40	40	40	40	40	40	40	40
6410		40	40	40	40	00	00	00	20	20	20	20	20	00	00	20
6420		00	40	40	40	40	40	40	40	40	40	40	40	40	40	40
6430		40	40	40	40	20	40	00	40	00	40	20	40	00	40	40
6440		00	40	40	40	40	40	40	40	40	40	40	40	40	40	40
6450		40	40	40	40	00	00	00	00	20	00	00	00	00	00	00
6460		20	40	40	40	40	40	40	40	40	40	40	40	40	40	40
6470		40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
6480		40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
6490		40	40	40	50	1F	FF									
64A0		FF														

Vemos la disposición en memoria del nivel generado. En la vista en ascii de la derecha podemos reconocer la estructura del nivel. Las @ son bloques indestructibles (los que parecen de metal y dejan un espacio entre cada uno).



En la representación de la IU, vemos el mapa dibujado a partir de las direcciones de memoria que hemos recibido. Observamos que hay un trozo que sobresale por la derecha. Esto es memoria reservada para niveles de tamaño doble, que se empiezan a utilizar más adelante en el juego.

Explicando algunas funciones del juego.

El juego consiste en derrotar a todos los enemigos del escenario y encontrar la puerta destruyendo bloques. Esto se consigue plantando bombas como Bomberman. De vez en cuando, pueden salir potenciadores al destruir bloques. Estos potenciadores proporcionan mejoras al personaje y a las bombas que coloca el personaje.

Una vez estén derrotados todos los enemigos podremos entrar en la puerta y pasar al siguiente nivel.

Ahora hay que adaptar la función de fitness. Para ello observamos la que existe en Super Mario Bros.

$$\text{Fitness} = \text{rightmost} - \text{frameActual}/2$$

Rightmost es posición máxima hacia la derecha que ha alcanzado super mario y frameActual es cuantos frames han pasado (penalización por tiempo). Además, si llega a completar el nivel recibe un bonus de 1000 puntos.

La función es relativamente sencilla porque el objetivo de Mario es llegar hasta el extremo derecho del mapa. La penalización por tiempo es para evitar que se quede quieto en algún punto.

En el caso de Bomberman, la salida se genera en lugares distintos. Las coordenadas de la puerta se pueden obtener de la memoria. Junto con la posición actual del jugador podemos hacer una función de distancia.

La función de Mario no tiene en cuenta factores adicionales, como puntos por matar monstruos, romper bloques o conseguir potenciadores. Por ello podemos probar una versión sencilla de una función de fitness e ir aumentándola más adelante.

Lo ideal sería que la cercanía a la puerta solo entrara en juego cuando se descubriera (rompiendo el bloque que la contiene), pero como prueba inicial debería funcionar. Queremos comprobar que el algoritmo aprende a acercarse a la puerta sorteando los obstáculos que se le presenten.

Por lo tanto se propone la siguiente función de fitness para este título:

$$\text{Fitness} = 100000000 / \text{GetDistanceToDoor()} - \text{frameActual}/2$$

La inversa tiene una gran magnitud para que el valor de fitness no tenga valores decimales.

Primeros entrenamientos

Tras varios entrenamientos, Bomberman no ha aprendido a doblar una esquina. Solo avanza en línea recta vertical u horizontal. Este movimiento en línea recta mejora la función de fitness, pero no tanto como lo haría si cambiara de dirección.

La distancia actualmente se calcula con la distancia Manhattan. Así que probamos con la distancia Manhattan al cuadrado.

Conseguimos un comportamiento nuevo pero igual de insuficiente. Consiste en que el personaje se queda moviéndose rápidamente izquierda, derecha, izquierda, derecha... en un cruce.

Parecería que el algoritmo se "acomoda" cuando hace solo un movimiento vertical (porque en ese escenario en concreto es lo que más puntos da si solo puedes pulsar un botón) y "se acerca a la puerta", y las mutaciones aleatorias no logran romper ese comportamiento. Salvo cuando intenta ese comportamiento errático izq., dcha., izq., dcha....

Probamos también con la distancia euclídea sin suerte.

Cabe destacar que si hay aprendizaje, porque aunque se mueva de forma rectilínea escoge el camino recto que da más puntos más a menudo.

Un comportamiento que observamos es que si sustraemos el tiempo, el algoritmo aprende a quedarse quieto. Porque es donde menos tiempo gasta, en contraposición a moverse y hacer cosas.

Nos debatimos los motivos de que no funcione. Hay varias hipótesis:

- Saturación de información

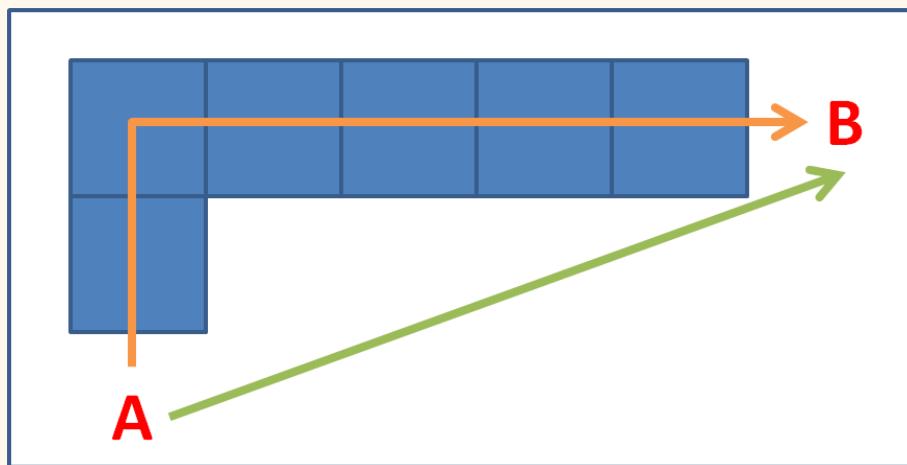
El algoritmo es muy simple para la cantidad de información que se le introduce. Si le pasamos poca información el algoritmo no será capaz de ver nada, pero si le pasamos mucha información tampoco verá nada por toda la saturación de información que se genera.

En este caso se le manda mucha información. Posición de bloques irrompibles, bloques rompibles, enemigos, posición del jugador, posición de las bombas, posición de los potenciadores, posición de *bloques de aire*...

- La distancia no se toma en cuenta de la forma correcta

El juego está organizado por casillas, la distancia quizás debería interpretarse como una magnitud discreta.

Por ejemplo, la distancia entre los puntos A y B no es 5.38 (camino verde) porque ese camino es imposible de recorrer, la distancia a todos los efectos es 6 (camino naranja).



- La red neuronal es muy simple y no es capaz de moverse cuando hay dos grados de libertad.

Aunque es una tesis prometedora, esto queda desmentido más adelante. Ponemos en práctica al algoritmo en un mapa subacuático de Mario. En esta clase de mapas, Mario debe aprender a moverse en vertical además de moverse en horizontal. Y podremos observar que lo consigue.

Aun así, la realidad es que una red neuronal tan pequeña no debe ser capaz de desarrollar comportamientos complejos como ENEMIGO CERCA: `huir_de_enemigo()`.

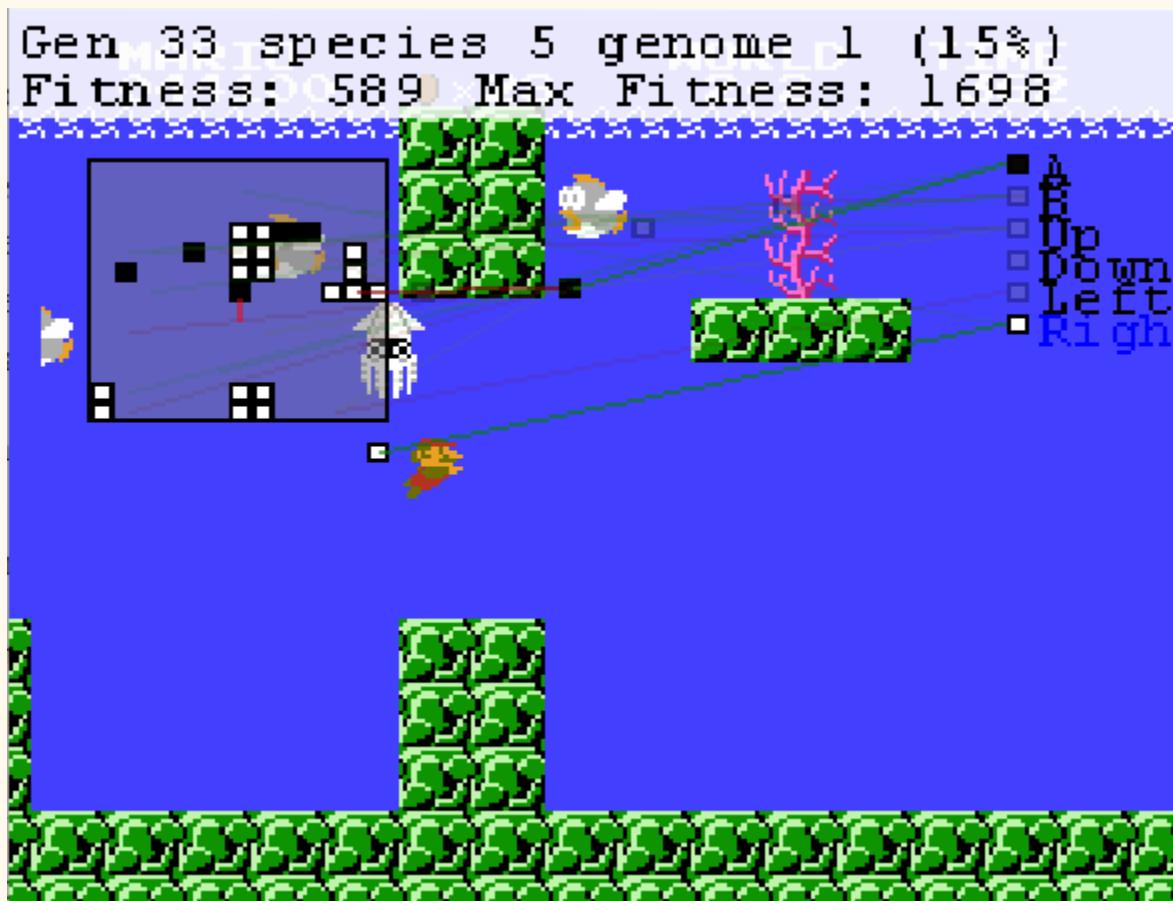
Además, teniendo en cuenta que una red neuronal con una capacidad de evolución de máximo una conexión por generación, será imposible cubrir todas las neuronas de entrada hasta bien entrados en el entrenamiento. Entendiendo que cada neurona se encarga de recibir un estímulo separado en el mapa (bloque, enemigo...).

De entre todas las hipótesis, la primera parece la más verosímil. Esto podría demostrarse desarrollando una solución al problema con un algoritmo más potente como PPO (Proximal Policy Optimization), que admite una capa de entrada mucho más grande.

Pero la tercera hipótesis (red demasiado simple) también tiene su razón. Habría que realizar una selección de inputs optimizada, descartando la información menos valiosa o irrelevante.

Entrenamiento de Mario bajo el agua

Entrenamos a Mario en un mapa acuático para descartar que se trate de un patrón y que el algoritmo está capacitado para aprender a moverse en dos grados de libertad. Además observamos que las monedas se interpretan como obstáculos, obviaremos las monedas a la hora de construir la matriz de entrada.



Mario nadando entre obstáculos y enemigos en vertical

Cambio a 1942

Dadas las circunstancias, cambiamos de juego de nuevo. Esta vez a la segunda alternativa que teníamos anteriormente. En este caso se trata de un juego *matamarcianos*, en el que el jugador controla un avión que debe destruir el resto de aviones mientras avanza. Hasta llegar al final del nivel.



La fórmula del fitness se diseña rápidamente

$\text{Fitness} = (\text{frameActual}/2 + \text{score}) * \text{nivelActual}$

Recompensamos al algoritmo por pasar más tiempo vivo, destruir aviones -la puntuación en el juego depende de los aviones destruidos y otros parámetros- y le damos una generosa recompensa por cada nivel que termina.

Planear con antelación la implementación en 1942

Analizando el problema que teníamos en Bomberman se llegó a una conclusión. La solución no funcionó por una mezcla entre saturación de información y una capa de entrada difícil de cubrir por una red básica que evoluciona despacio.

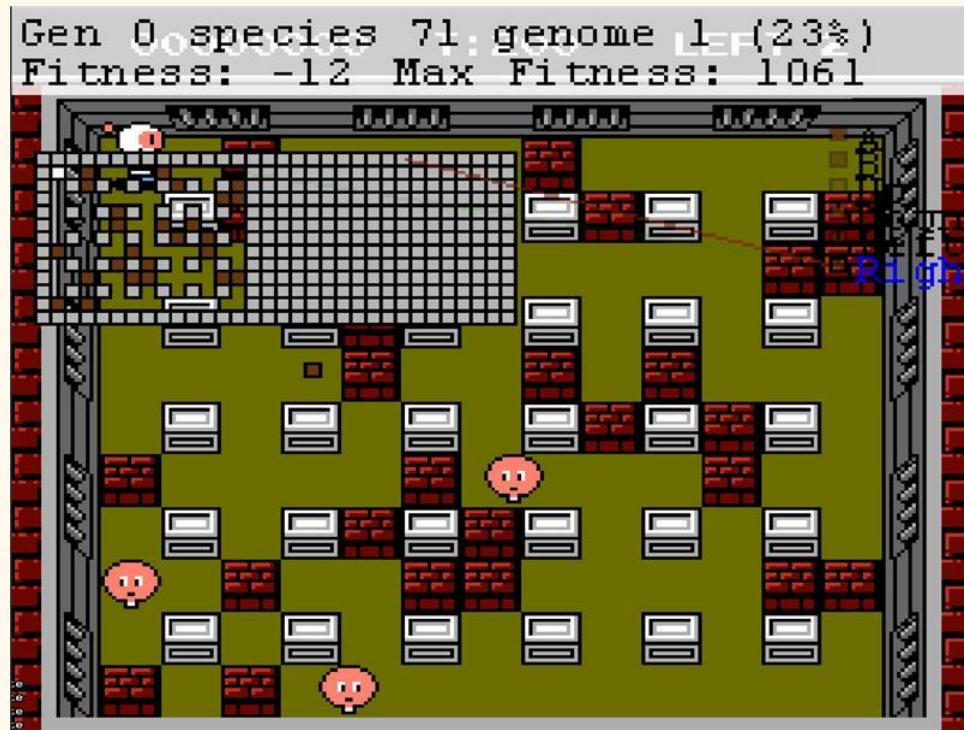


Esta es una red del principio de un entrenamiento. Como podemos ver en la representación, cuando la red lee un bloque debajo de Mario pulsa "Derecha".



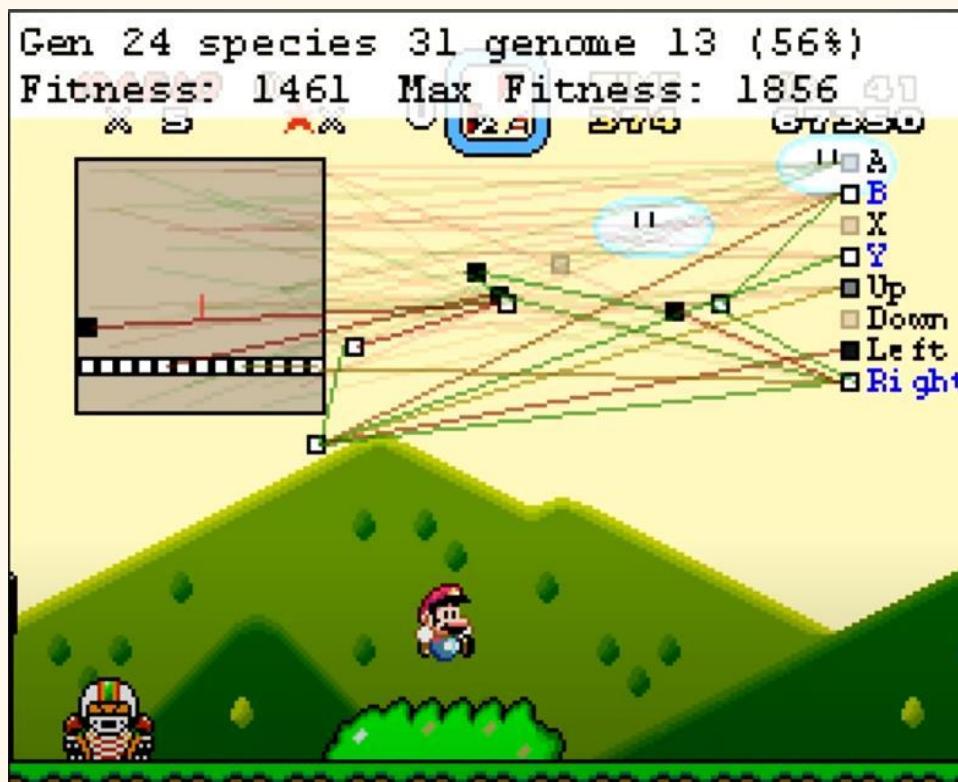
Cuando lee un enemigo pulsa la A para saltar.

Tal y como se le planteó el problema a la Inteligencia Artificial (Un input fijo que no se mueve, al contrario del input que se recibe en la de Mario), el algoritmo tendría que haber acabado encontrando una configuración con neuronas repartidas por todo el espacio para poder leer dónde hay una esquina, dónde hay un bloque, donde hay un enemigo y - lo peor de todo - donde está Bomberman.



Estaba leyendo incluso de bloques fuera de la zona de juego. Era una disposición muy complicada para el algoritmo y muchos grados de libertad.

Es decir, no es tanto una limitación de la red sino del diseño que se le dio a la solución. Porque la red puede volverse bastante concreta y complicada, superando como ya hemos visto varios niveles de Mario.



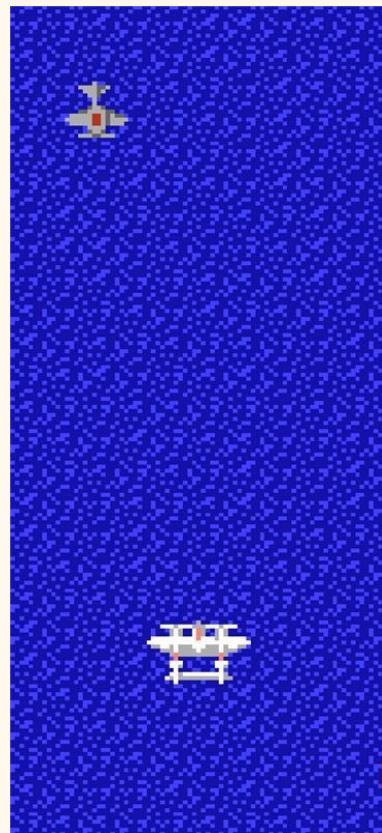
Para 1942 podemos plantear la solución de dos formas distintas:

Segmentar la imagen horizontalmente

La red neuronal no será capaz de ver lo que haya a los lados. Solo lo que tenga en frente, a lo largo del eje Y solo para el rango de X actual. En un principio el movimiento vertical estará desactivado. El input tendrá que mostrar los aviones (0 destructible) y balas (1 no destructible). La posición del avión es irrelevante, como en Mario. El escenario cambia dinámicamente y la red sólo tendrá que reaccionar a lo que vaya ocurriendo (el palito rojo en las vistas de Mario).

Primero leemos todo el sprite buffer, y luego ponemos en la matriz de entrada aquellos datos que correspondan a lo que la red debe poder ver, según la posición horizontal del avión. La matriz puede ser simplificada (agrupando píxeles como en Mario) o de asociación pixel-elemento de matriz.

Esto sería una aproximación como la que tuvo el autor original en la adaptación a Mario. Luego podríamos volver a intentar con un campo de visión más amplio también.



Abarcar toda la pantalla

Esta sería una aproximación como la que tuvimos en Bomberman y es la que menos tiempo necesita para implementarse. Aun así, en este caso sí podría justificarse. En este juego la pantalla se mueve constantemente. Cualquier neurona que la red coloque sobre la pantalla se activará si un avión enemigo pasa por encima. Además, en este caso no existe saturación de información. Solo hay dos posibles inputs, bala o enemigo. El algoritmo podría intentar encontrar una configuración que permita a la nave esquivar y sobrevivir generando una serie de neuronas que avisen a la nave con antelación si está en peligro.

Como se tarda menos, nos decantamos por abarcar toda la pantalla. Y si da tiempo podemos intentar la versión con segmentación de imagen.

Fuera cual fuese la técnica, las entradas solo se pueden obtener de una manera.

Leer de memoria

Todos los aviones están configurados en imágenes más pequeñas que forman el todo.



Así esta dividida la nave, en varios sprites mas pequeños y modulares (cambian de orientacion, forma y de color). En este ejemplo el ala izquierda ha cambiado de orientación (Se modificó el sprite cambiando el valor de la memoria en la dirección 0x0232).

El esquema que sigue una nave en memoria para su representación es el siguiente:

Address	Value	Meaning						
0x0230	0-255	player posY Ala izq						
0x0231	51 (Ala)	player sprite Ala izq						
0x0232	3 (blanco orientacion neutral)	player color + orientación Ala izq						
0x0233	0-255	player posX Ala izq						
0x0234Centro						
0x0235	.	.						
0x0236	.	.						
0x0237	.	.						
0x0238Ala dcha						
0x0239	.	.						
0x023A	.	.						
0x023B	67 (blanco orientacion espe)	.						
0x023CTrasero izq						
0x023D	.	.						
0x023E	.	.						
0x023F	.	.						
0x0240Trasero dcha						
0x0241	.	.						
0x0242	.	.						
0x0243	.	.						

Para mayor legibilidad:

En la dirección 0x0230 tenemos un valor entre 0 y 255 que representa la coordenada en Y del sprite.

En la dirección 0x0231 tenemos el valor que representa la imagen que se usa como sprite.

En la dirección 0x0232 tenemos el valor que representa el filtro de color y la orientación del sprite

En la dirección 0x0233 tenemos el valor que representa la coordenada X del sprite.

Esta secuencia de 4 bytes se repite para cada una de las piezas que componen la nave del jugador. Para los aviones más pequeños o de tres piezas, habrán tres secuencias de 4 bytes en lugar de cinco.

Los aviones enemigos se cargan en el sprite buffer. El rango del sprite buffer comienza en 0x0200 y termina en 0x02FF. Si hay un avión se incluirá ahí. Nosotros tenemos que rastrear el buffer en busca de estos, siempre filtrando lo que sea parte del jugador.

El buffer para la posición de las balas enemigas en X está entre 0x04C3 y 0x04CA. De igual forma, hay espacio para 8 balas en el buffer para la posición Y 0x04AA.

Para este caso, se ha decidido no representar los aviones con todos los trozos. La posición de un avión en la pantalla de cara a la red neuronal será la posición de su sprite central.

Aparte de la posición de los actores, también buscamos otros datos.

- Hay que buscar donde está el dato que indica si el jugador ha muerto. 0x03B1 = 1
- La puntuación (necesario para el cálculo del fitness):

Por ejemplo, 002850 puntos.

0x0427 0x00

0x0428 0x00

0x0429 0x02

0x042A 0x08

0x042B 0x05

0x042C 0x00

- El nivel actual (necesario para el cálculo del fitness): Nivel 1 sería 0x0438 = 0x01
- Hay que filtrar ciertos elementos del sprite buffer como la interfaz, potenciadores, explosiones, etc. para no pasarla como entrada de la red.
- Además durante la búsqueda se encontraron datos que se extrajeron pero nunca se usaron. Por ejemplo, las vidas restantes del jugador, la cantidad de acciones de ‘rodar’ restantes o el buffer para las balas del jugador.

Este proceso se realizó mediante ingeniería inversa. Ayudándonos de tutoriales^{5,6} y de las herramientas que ofrece Bizhawk, podemos buscar direcciones de memoria y valores en memoria colocando filtros sobre los valores aceptables, congelando direcciones de memoria, comparando con valores anteriores, añadiendo los registros deseados a una lista de vigilancia...

Podemos incluso dar vuelta atrás en el tiempo en el juego.

The screenshot shows the RAM Watch interface in Bizhawk. On the left, there's a list of watches with columns for Address, Value, Changes, Domain, and Notes. A specific entry for address 04C3 is highlighted with a value of 177 and a note 'RAM Bullet1X'. The main pane displays a memory dump with columns for Address, Value, Prev, and Changes. The 'Changes' column highlights differences from the previous value. A search interface on the right allows filtering by address, value, or changes, with comparison operators like Equal To, Less Than, Greater Than, and Greater Than or Equal To. The search bar currently shows '0'. The bottom of the screen shows a Google search results page for 'rewind bizhawk'.

Address	Value	Changes	Domain	Notes
04AA	241	0	RAM	Bullet1Y
04C3	177	0	RAM	Bullet1X
04AB	243	84	RAM	
04AC	0	46	RAM	
04AD	0	36	RAM	
04C4	183	28	RAM	
04C5	0	46	RAM	
04C6	0	36	RAM	

Address	Value	Prev	Changes
0488	0	0	0
0489	0	0	0
048A	0	0	0
048B	0	0	0
048C	0	0	0
048D	2	6	151
048E	136	136	0
048F	0	0	0
0490	0	0	0
0491	0	136	11
0492	0	136	3
0493	0	0	2
0494	0	0	2
0495	0	0	0
0496	0	0	0
0497	0	0	0
0498	0	0	0
0499	140	136	26
049D	140	136	22
049E	136	136	27
049F	135	136	30
04A0	140	0	23
04A1	136	0	23
04A2	140	136	24
04A3	136	0	24
04A4	0	0	0
04A5	0	0	0
04A6	0	0	0
04A7	166	192	318
04A8	0	0	0
04A9	0	0	0
04AA	241	234	258
04AB	243	140	84
04AC	0	0	46
04AD	0	0	0
04AE	0	0	0
04AF	0	0	0
04B0	0	0	0
04B1	0	0	0
04B2	0	0	0
04B3	0	0	0
04B4	0	0	0
04B5	111	116	710
04B6	172	42	692
04B7	59	44	640
04B8	7	44	645
04B9	163	0	777
04BA	7	0	676
04BB	177	120	576
04BC	98	0	790
04BD	0	0	0
04BE	0	0	0
04BF	0	0	0
04C0	113	153	592
04C1	0	0	0
04C2	0	0	0
04C3	177	144	121
04C4	183	137	28
04C5	0	0	46
04C6	0	0	36
04C7	0	0	0
04C8	0	0	0
04C9	0	0	0
04CA	0	0	0
04CB	0	0	0
04CC	0	0	0
04CD	0	0	0
04CE	148	126	789

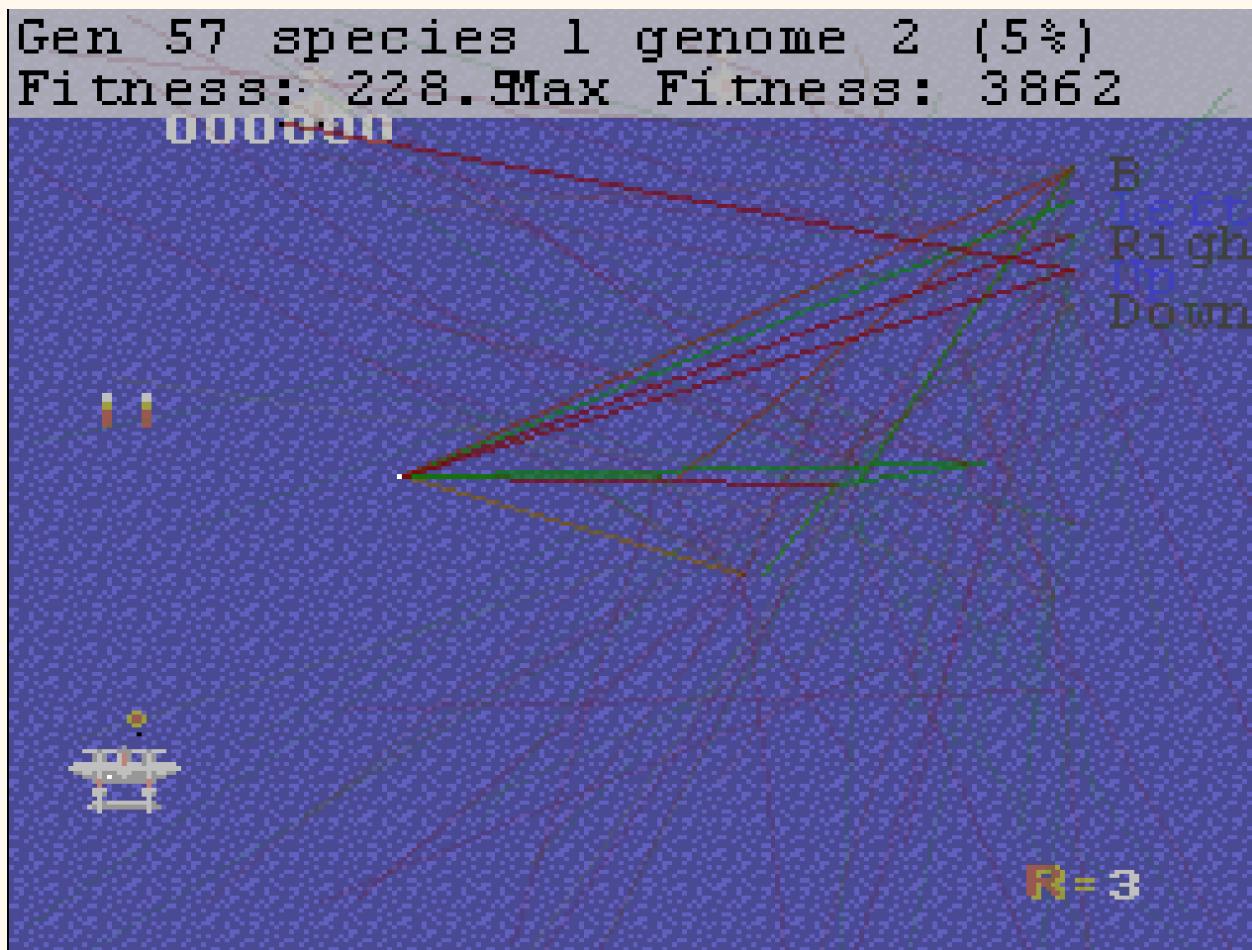
Primer entrenamiento exitoso de 1942

Este primer entrenamiento servirá como primera referencia y toma de contacto.

Al no haber condición de parada, la ejecución del algoritmo continúa hasta ser frenada manualmente. En cuanto se ven resultados interesantes, paramos la ejecución:

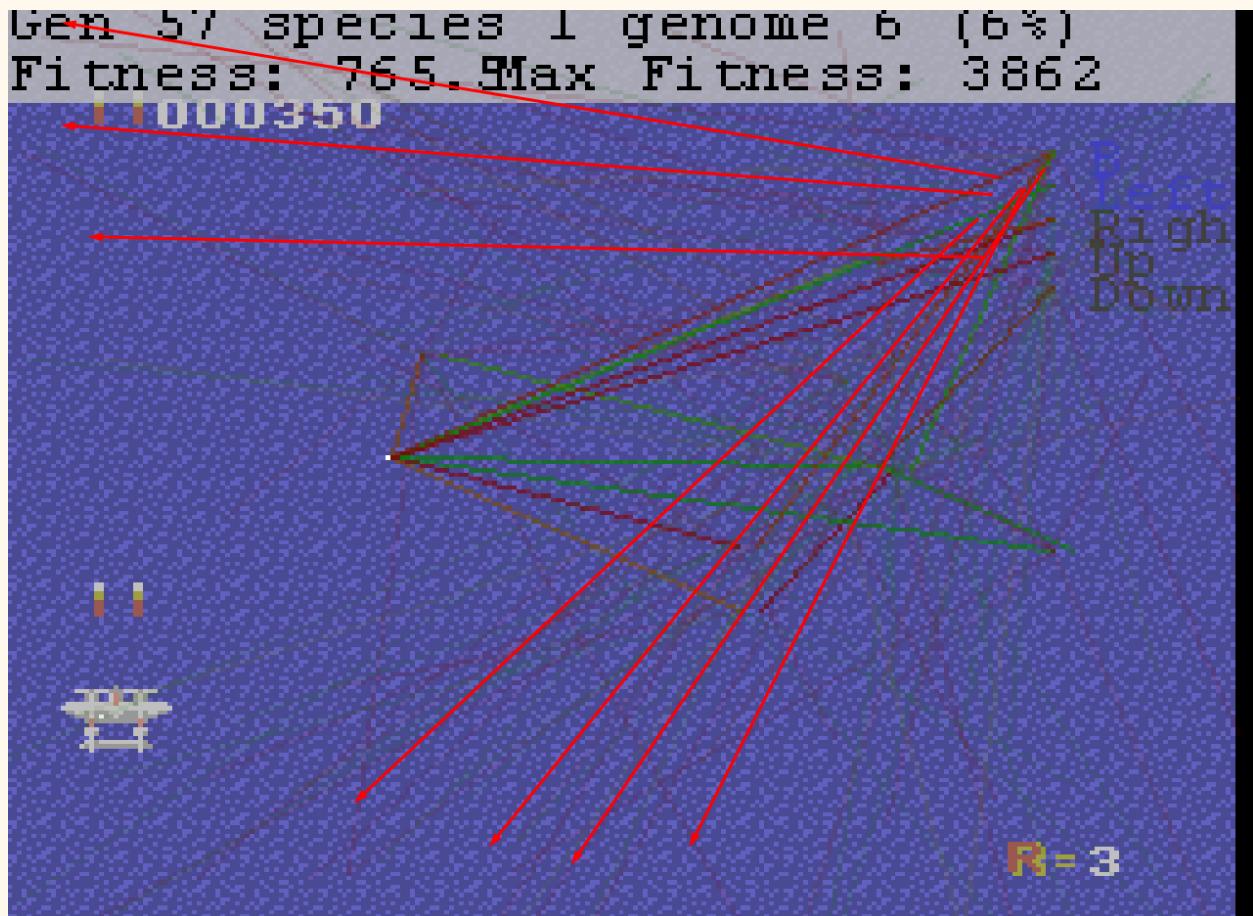
Cada entrenamiento en este punto del trabajo dura alrededor de dos días y medio. Tras sesenta horas de ejecución, los primeros resultados salen a la luz.

En general, aprendió a sentarse en la esquina de la izquierda y aguantar lo más posible. Aprendió a no parar de disparar.



Este individuo -no es el mejor que rinde pero es un poco superior a la media con 2000 puntos- tiene neuronas repartidas por todo el escenario. Tendiendo al infinito, quizá pueda incluso cubrir con neuronas todos los píxeles. Pero en este tiempo limitado hay una repartición

aleatoria y ruidosa. Aunque observando igual hay un patrón con forma de cono (que rodea a la nave y le avisa de si viene un peligro)



[Mejor partida.mp4 \(sharepoint.com\)](#)

Pruebas con diferentes parámetros

Ahora debemos entender el algoritmo genético escrito, para poder hacer modificaciones sobre el mismo. Diseccionamos el siguiente pseudocódigo:

- Selección de los individuos de cada especie por encima de la media
- Ordenación de mayor a menor puntuación
- Eliminar especies estancadas (Especies que lleve más tiempo del estipulado sin que un individuo llegue al top1 general)
- Ordenación de mayor a menor puntuación
- Calcular fitness medio por especie
- Eliminar especies débiles (Especies que tengan un fitness medio relativamente muy bajo frente al resto)
- Generar hijos para cada especie (Cada especie engendra proporcionalmente a su fitness)
- Eliminamos en todas las especies todos los individuos salvo el top 1
- Generar hijos hasta llenar la población seleccionando especies aleatorias
- Añadir a las especies todos los hijos engendrados según su especie, si no tiene una especie se crea una especie para él
- Contador de generación +1

A la hora de engendrar descendencia se utiliza un 0.75 de probabilidad que sea fruto de un cruce. Si no fuese un cruce, se haría una copia de un genoma aleatorio.

Además, a la hora de realizar mutaciones, existe un ratio que cambia aleatoriamente en cada iteración. El ratio indica a cada genoma cuantas veces debe hacerse a sí mismo una mutación de cada tipo.

También tenemos un parámetro de elitismo que nos dice cuantos individuos deben sobrevivir cuando se desechan las especies.

En las pruebas jugaremos con la mutación y el elitismo. Además probaremos con un operador de cruce distinto.

Los primeros resultados que obtenemos son prometedores. Modificamos el elitismo de uno a tres. Ahora en cada generación sobreviven los tres mejores de cada especie.

Hay movimientos evasivos mucho más complejos. Hay más variedad en cómo las diferentes especies intentan resolver el problema. Existe una que hace como en el anterior entrenamiento,

pegándose a la esquina izquierda. Ahora la especie más dominante no sólo llega más lejos sino que lo hace jugando en el centro de la pantalla, combinando movimientos horizontales y verticales. Además hay algunos movimientos que reaccionan muy bien a los enemigos en pantalla, los persigue o los esquiva.

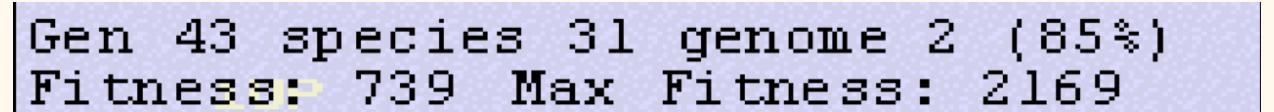
[Saludos.mp4 \(sharepoint.com\)](#)

Viendo la mejora, planteamos probar modificando el elitismo en Bomberman II, pero la idea se descarta.

Ahora probaremos modificando la probabilidad de mutación. Teníamos `random(1,2) == 1` y ahora tendremos `random() > 0.25`. Pasamos de 50% a 75%.

Tras también esperar 44 generaciones, no hemos visto comportamientos nuevos. El fitness tampoco mejora, de hecho queda en 1950. Comparando con los records del entrenamiento anterior alrededor de 5000.

Viendo que el elitismo había supuesto una mejora, pasamos de elitismo 3 a elitismo 5.



```
Gen 43 species 31 genome 2 (85%)
Fitness: 739 Max Fitness: 2169
```

El resultado empeora tras 44 generaciones. Hasta el momento nada consigue hacer frente al elitismo 3.

Ahora probaremos bajando la probabilidad de mutación. En lugar de un 75% tendremos un 25%.

Tenemos resultados como máximo de 2500 de fitness. Suponiendo una mejora frente a la configuración con el 75% pero peor frente a la configuración estándar.

Tras semanas de entrenamientos, tenemos el siguiente desglose:

Resultado en la generación 44:

Configuración estándar (elitismo a 1, mutación 50%): 1876 Max fitness -> **3862 en la gen 55**

Elitismo 3: 5190 Max fitness

Elitismo 5: 2169 Max fitness

Mutación 75%: 1927 Max fitness

Mutación 25%: 2536 Max fitness

Más tarde nos daremos cuenta de que estos resultados quizás no sean válidos, pero hasta el momento existe la creencia de que el elitismo 3 ha sido la configuración claramente vencedora.

Tenemos la intención de hacer modificaciones sobre el algoritmo, pero no queremos hacer un cambio radical, porque llevaría mucho tiempo. La intención entonces es cambiar el operador de cruce. Esto estropeará la esencia de NEAT, porque la clave en este algoritmo es que el operador de cruce tiene la capacidad de cruzar estructuras neuronales gracias al caracterizador de innovación / número histórico.

Simplemente, por curiosidad, cruzaremos usando el cruce de punto simple. Este cruce consiste en seleccionar un índice dentro de los límites del genoma de los dos padres, y utilizarlo para marcar la primera mitad de un parente y la segunda del otro parente. De esta manera, el hijo obtendrá la primera parte de un parente y la segunda del otro parente, habiendo sido divididas con este índice.

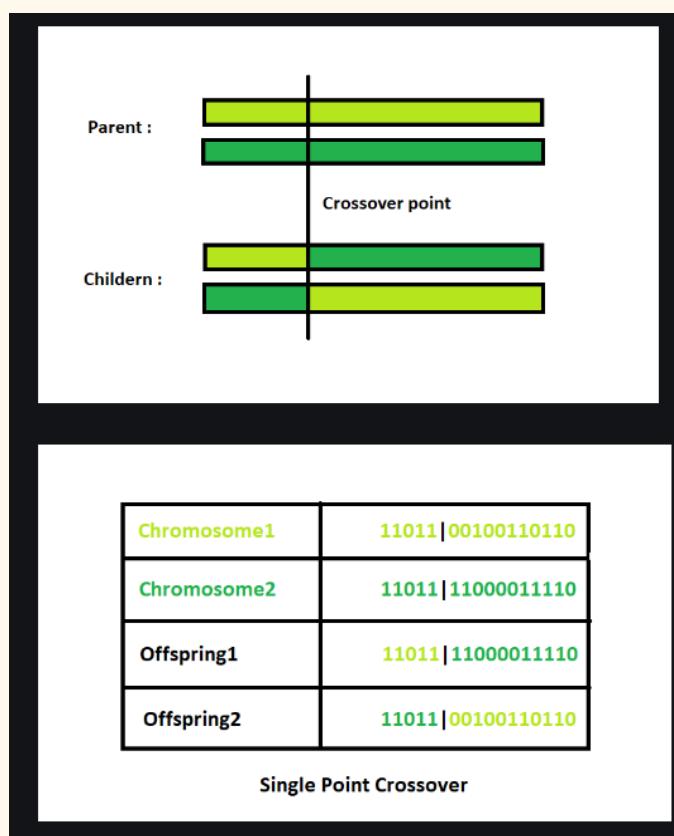


Figura obtenida de GeeksforGeeks. [Crossover in Genetic Algorithm - GeeksforGeeks](https://www.geeksforgeeks.org/crossover-in-genetic-algorithm/)

Tras el entrenamiento tenemos la siguiente matriz:

Resultado en la generación 44:

Configuración estándar (elitismo a 1, mutación 50%): 1876 Max fitness -> **3862 en la gen 55**

Elitismo 3: 5190 Max fitness

Elitismo 5: 2169 Max fitness

Mutación 75%: 1927 Max fitness

Mutación 25%: 2536 Max fitness

Cruce de punto simple, configuración estándar (elitismo a 1, mutación 50%) -> 3102 Max fitness

Antes de seguir adelante queremos completar esta matriz. Se ha comprobado el elitismo 3 con mutación al 50%, pero no se ha probado el elitismo 3 con mutación 25%, las variaciones con mejor resultado hasta el momento. Obtenemos que Elitismo 3, mutación 25%: 2431 Max fitness.

A partir de aquí podemos sacar que elitismo 3 debería juntarse con mutación 50% y con cruce punto simple.

Resultado en la generación 44:

Configuración estándar (elitismo a 1, mutación 50%): 1876 Max fitness -> **3862 en la gen 55**

Elitismo 3: 5190 Max fitness

Elitismo 5: 2169 Max fitness

Mutación 25%: 1927 Max fitness

Mutación 75%: 2536 Max fitness

Cruce de punto simple, configuración estándar (elitismo a 1, mutación 50%) -> 3102 Max fitness

Elitismo 3, mutación 75%: 2431 Max fitness

Cruce de punto simple, elitismo 3 (mutación estándar) -> 1828 Max fitness

Analizando los resultados, proponemos revisitar el elitismo 3 estándar. Hasta ahora el tutor estaba asumiendo que se trataban de valores promediados y que eran representativos de la configuración. Si los resultados no son estables y pueden cambiar de una ejecución a otra hay un problema.

Y en efecto, una vez llegamos a 44 generaciones en elitismo 3, obtenemos una puntuación máxima de 1056.

Es decir, repitiendo la prueba que mejores resultados nos había dado, hemos obtenido el peor resultado. Los dos resultados con la misma configuración no solo no convergen, son distintos.

Ahora necesitamos representar los resultados gráficamente para poder extraer conclusiones.

Los archivos .backup que explicamos al principio se han ido conservando en muchas configuraciones. Y a pesar de algunos datos perdidos o traspapelados, podemos dibujar tendencias generales y analizar cómo se comporta el algoritmo más a fondo. Pero antes, hagamos un inciso.

El fitness se calcula en cada frame de la partida para el individuo que está jugando. Al finalizar, eso queda guardado para ese individuo en la estructura de datos de la especie (que contiene todos los individuos de la especie). Cuando se ha hablado de Max Fitness, se habla de la puntuación máxima que ha obtenido cualquier individuo de cualquier especie hasta el momento.

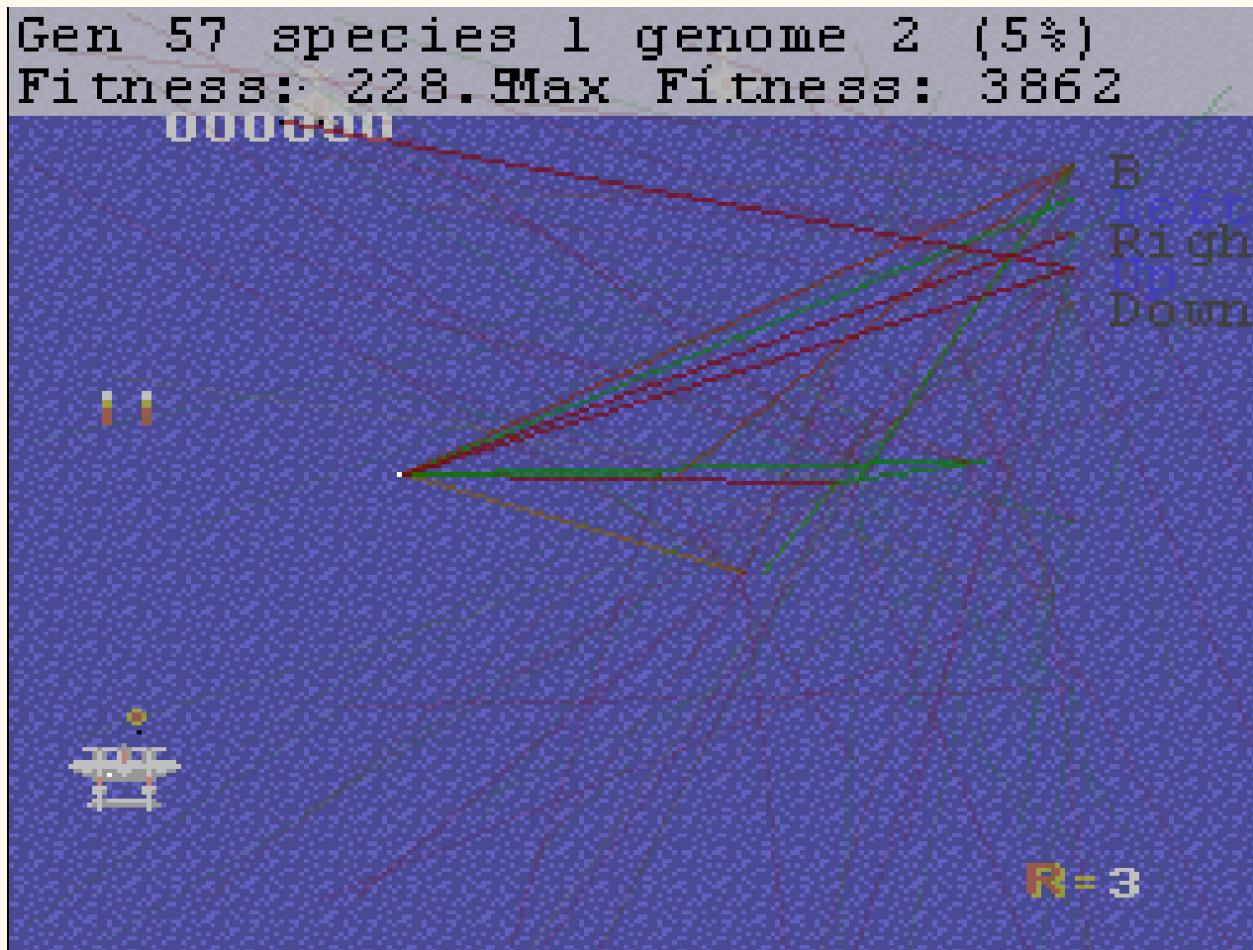
Todas las partidas además, comienzan a partir de un archivo de partida guardada. El juego carga desde ahí y los individuos juegan. El juego es determinista a partir de una partida guardada y es igual siempre, salvo por algunos aviones enemigos especiales que se generan aleatoriamente.

En definitiva, si el juego siempre es igual a partir de ese punto de guardado que tenemos, dada una red neuronal (individuo) debería dar el mismo resultado siempre.

En un principio, la variabilidad de los resultados se puede justificar en que las redes neuronales se forman aleatoriamente y se van seleccionando genéticamente. Dada la abundante cantidad de combinaciones de configuraciones de red neuronal que se pueden formar, un entrenamiento puede dar resultados muy distintos a otro, incluso con la misma configuración de parámetros del algoritmo genético.

Recordemos que la red neuronal se construye para colocar neuronas de "detección" pixel a pixel (250x250 aproximadamente), y que una mutación genética hacía operaciones neurona a neurona.

Quizás convendría que una neurona de "detección" agrupase varios píxeles. Recordemos el ejemplo que vimos anteriormente. Ahora viendo que cada una de estas neuronas colocadas se activan o desactivan cuando pasa un avión, bala, etc (Coincidencia pixel con pixel).



Representación gráfica de los resultados

Diseccionamos la estructura de los ficheros .backup para poder graficar la información:

En la cabecera de cada fichero tenemos información de la generación entera (todas las especies):

```
gen: 3
maxFitness: 443
n de especies: 170
```

Después, tenemos por cada especie lo siguiente:

```
topFitness de la especie: 200
parametro de history (NEAT): 2
número de genomas en la especie: 1
```

Y por cada especie los individuos en bucle:

Fitness: 200

Max Neurons: 61697

Tipos de mutación con su ratio:

connections

0.2375

step

0.105263

link

1.9

enable

0.210526

bias

0.38

node

0.526315

disable

0.421052

done

genes del genoma: 3

input: output: weight: innovation: enabled:

61697 1000003 -1.9648426770837 13 1

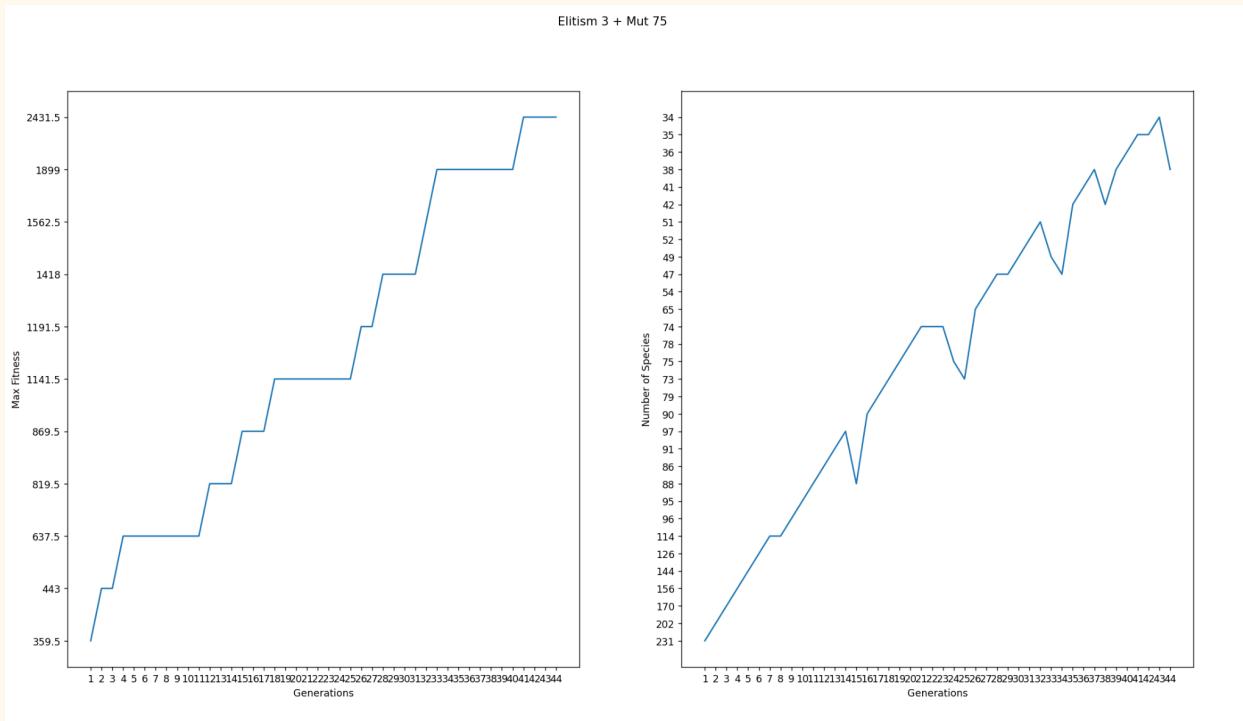
51839 1000003 -0.49653614917447 12 0

37731 1000004 -0.93514816736351 11 1

De esa manera podríamos leer ya el fichero siguiente (copia de seguridad de la generación 3 de elit3 + mut75):

[backup.3.1942.state.Pool](#)

Con un script en Python, sacamos una conclusión rápida de la configuración con elitismo 3 y mutación 75%:



A medida que avanza el tiempo, el fitness máximo va aumentando, así como el número de especies va disminuyendo.

Observando las gráficas vemos que existen mesetas, que representan un estancamiento del avance del algoritmo. El fitness máximo no varía un ápice, lo que implica que las mutaciones crean un individuo especialmente bueno y el elitismo lo mantiene hasta que aparece otro mejor.

Las especies disminuyen. Asumiendo que la población se mantiene constante, eso significa que las diferencias entre individuos disminuyen con el tiempo y que al final acabamos con más individuos en menos especies.

Los tutores en este momento no están al tanto de que no existe condición de parada, y se preguntan por qué paramos en la generación 44, si el crecimiento es lineal y no hay ninguna corrección en la pendiente.

Se decidió en su momento parar en la generación 44, porque habían transcurrido ya dos días y se vieron resultados interesantes. Para mantener una comparación justa se empezó a parar en esa generación.

En este caso podemos ver el valor de una condición de parada. Y es que no por parar siempre en la 44 tenemos una comparación más justa. Hay que dejar que el algoritmo siga su trabajo hasta el final, definiendo una condición de parada. Por ejemplo, debemos parar cuando pasen 10 generaciones sin encontrar una mejora. Si un entrenamiento tarda 44 generaciones para llegar a ese punto y otro 60 no pasa nada. Lo importante es ver a dónde puede llegar la evolución. Pero si el algoritmo sigue teniendo la capacidad de generar individuos mejores, debe continuar.

Ahora nos encontramos en la tesitura de trabajar a contrarreloj, realizando entrenamientos de nuevo, cumpliendo con unos estándares de calidad más altos para poder realizar comparaciones correctas. En este momento recordamos que el emulador tiene la funcionalidad de aceleración. Podemos ejecutar el juego (y el algoritmo) a un máximo de 4 veces la velocidad normal.

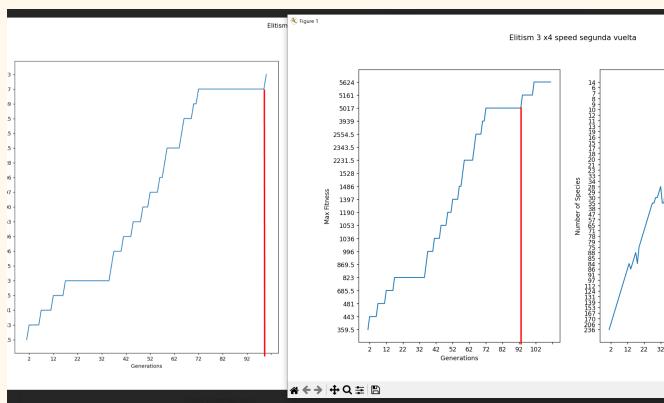
Al principio del trabajo, descartamos utilizar esta herramienta, porque parecía introducir variabilidad en los resultados. Pero viendo que esta variabilidad no viene de la ejecución rápida sino de otros factores, decidimos aplicar una aceleración al juego de ahora en adelante.

Esta herramienta solo aumenta la frecuencia del reloj de la CPU emulada, es razonable asumir que un entrenamiento acelerado es simplemente un entrenamiento normal que ocurre más rápido. Siempre y cuando el algoritmo no tenga complicaciones con esto.

Además planteamos la posibilidad de introducir una implementación propia del algoritmo, fruto de no haber conseguido conclusiones hasta ahora.

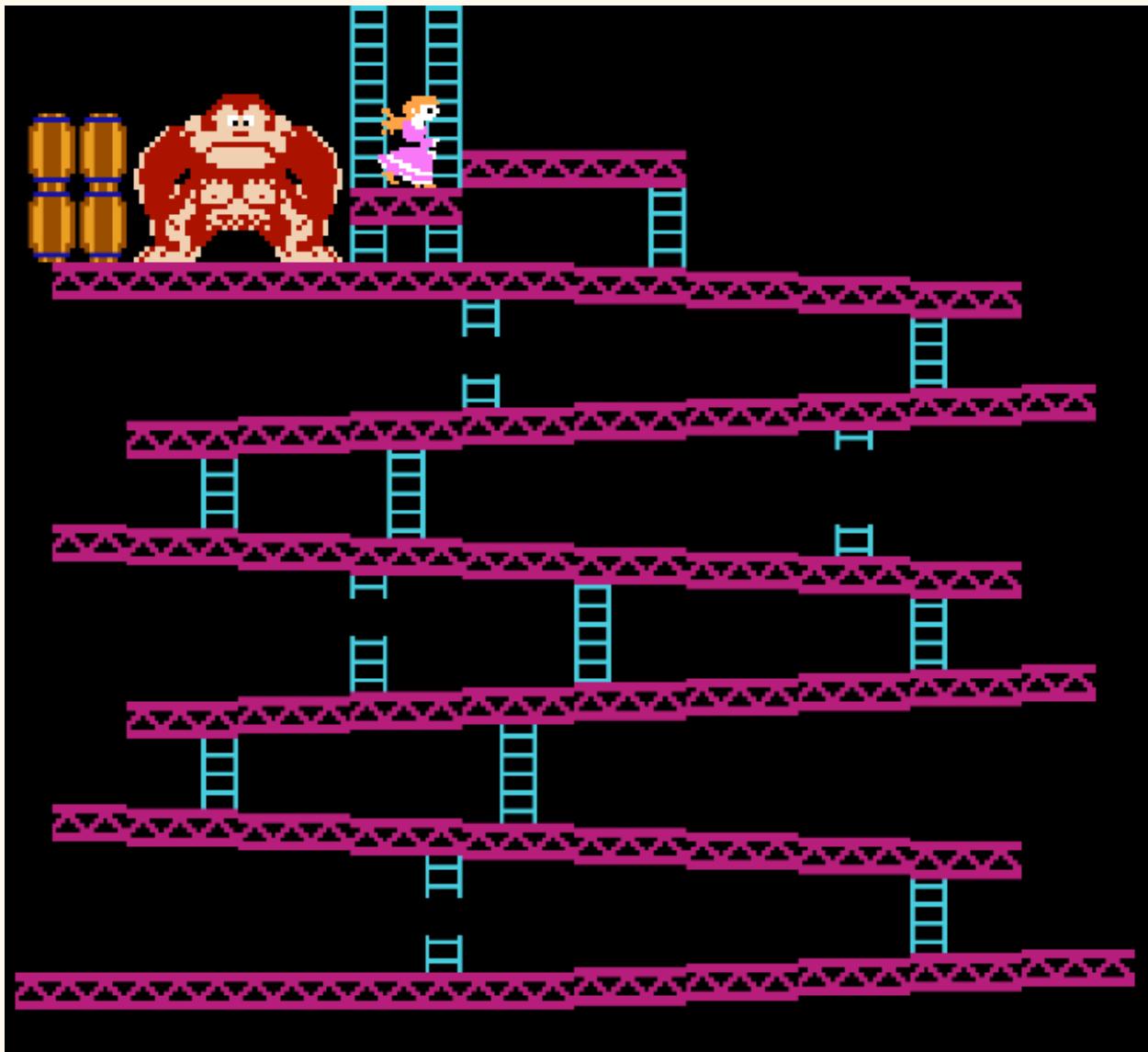
Ahora, reentrenamos con el juego acelerado la configuración con elitismo 3. Además lo hacemos dos veces. Aquí nos damos cuenta de que no hay una aleatoriedad evidente. Los dos algoritmos se ejecutan igual. Y es que en Lua, la función random de librería estándar de math parece repetir la misma secuencia a no ser que la inicialices con un seed en un principio ¹².

Aquí tenemos dos entrenamientos frente a frente, tenemos un crecimiento idéntico:



Ahora tenemos que probar a entrenar el algoritmo poniendo un seed que garantice la aleatoriedad. Utilizaremos el epoch del tiempo del sistema operativo en milisegundos. Así nos cercioramos de que en cada ejecución tenemos una semilla distinta.

Mientras se produzcan los entrenamientos, en paralelo se irá fabricando un juego sencillo en Unity al que aplicarle el algoritmo NEAT. Será un clon del primer nivel del juego de NES Donkey Kong.

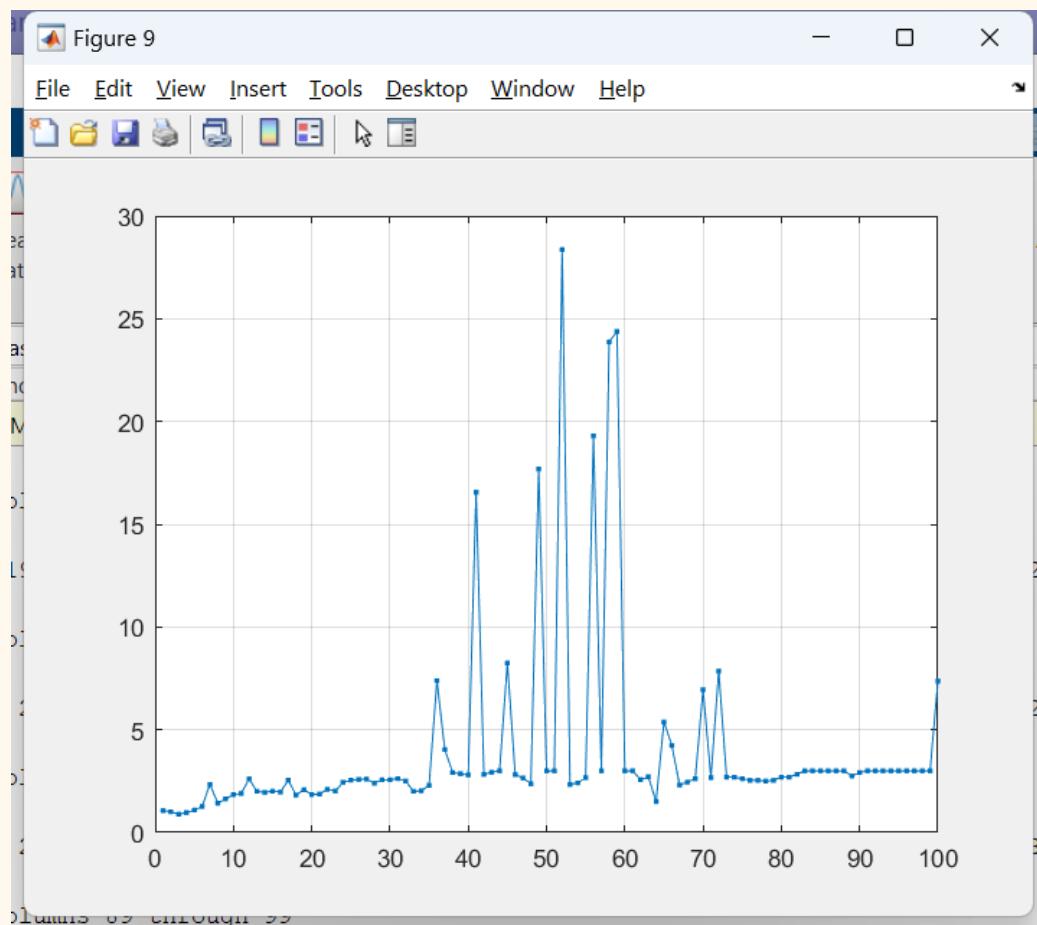


En los sucesivos entrenamientos, se observa que hay muchos individuos con fitness 0.

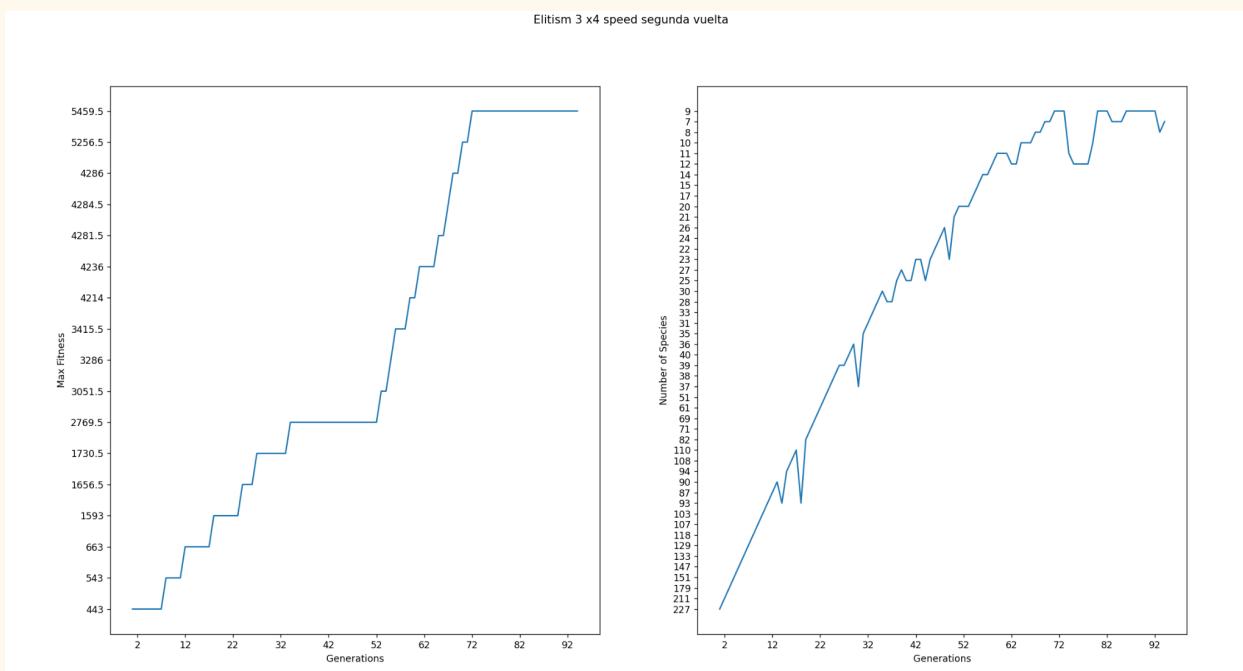
Esto se debe a que el algoritmo engendra los nuevos individuos y estos quedan con fitness a 0, hasta que les toca jugar en la siguiente iteración. La información que nos llega en los ficheros, entonces no es del todo representativa del rendimiento de la población durante la generación. En un fichero .backup estamos respaldando la información de la población al principio de la generación, entonces incluimos individuos engendrados y sin evaluar, y perdemos los datos tras la eliminación de individuos y especies ‘malos’. Entonces tenemos que descartar los ceros, puesto que son individuos que no han sido evaluados, y después tener en cuenta que se han eliminado de los datos los peores individuos de la generación.

Esta conclusión coincide con el dato de que alrededor del 80% de los individuos del fichero tienen fitness 0.

En el siguiente gráfico observamos el promedio de individuos por especie con fitness no nulo en cada generación:

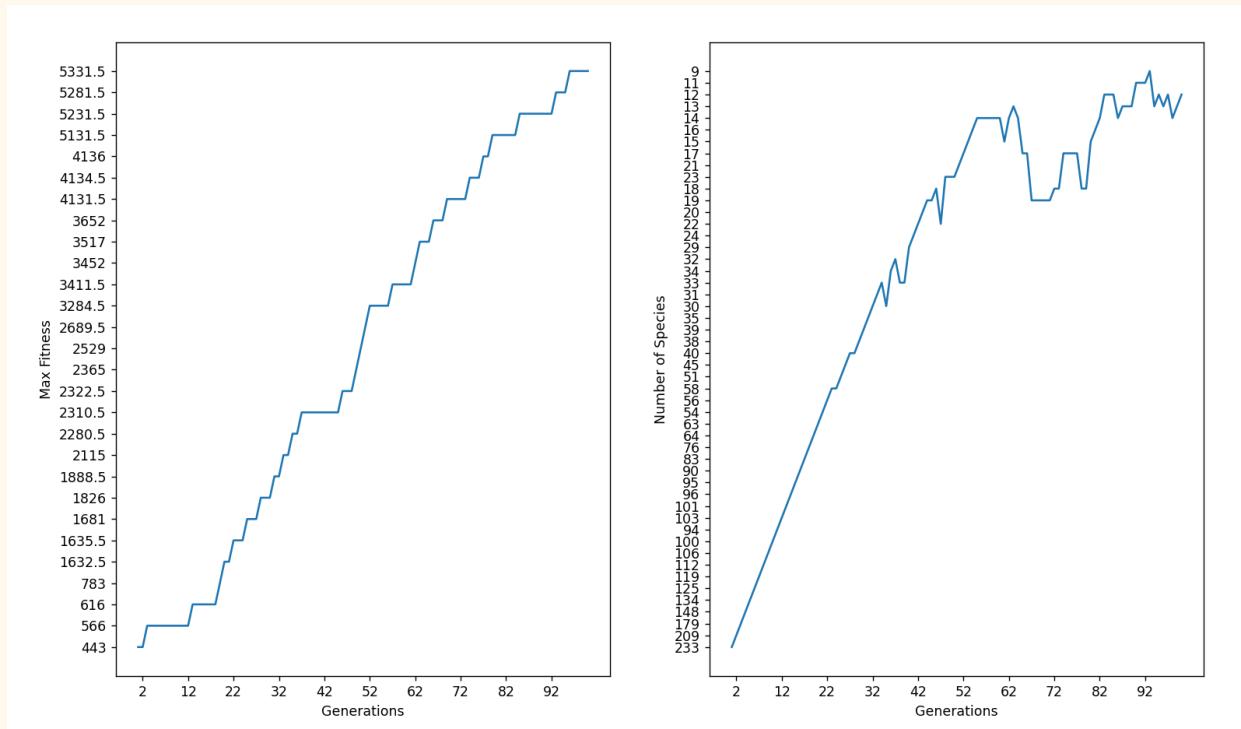


Lo que puede, en parte, coincidir con la meseta que observamos en la tendencia del fitness.

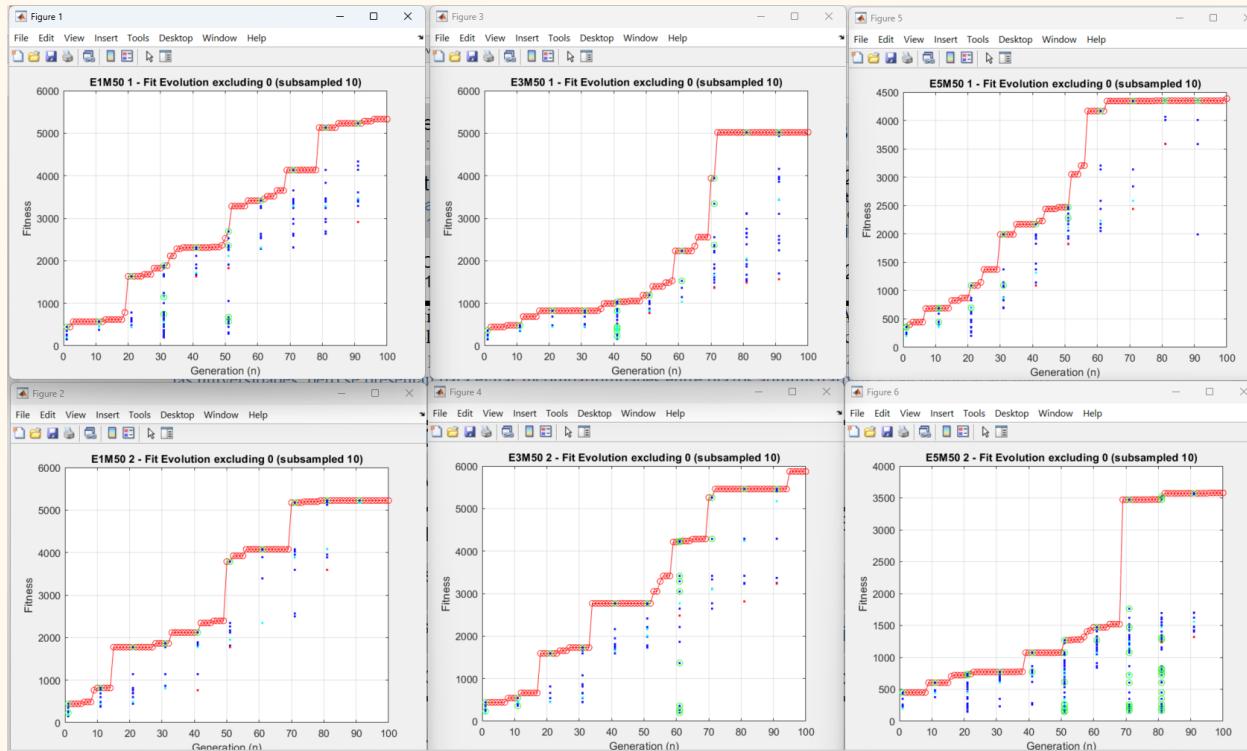


Aun así intentaremos trabajar con el 20% de individuos no nulos.

Vemos que la tendencia general converge. Tras 100 generaciones, los algoritmos suelen terminar con un max fitness en torno a 5000. Vemos entonces que el primer gran resultado de elitismo 3 simplemente fue capaz de llegar a esa cifra antes de tiempo. Pero lo normal es que lleve mucho más tiempo llegar hasta ahí.

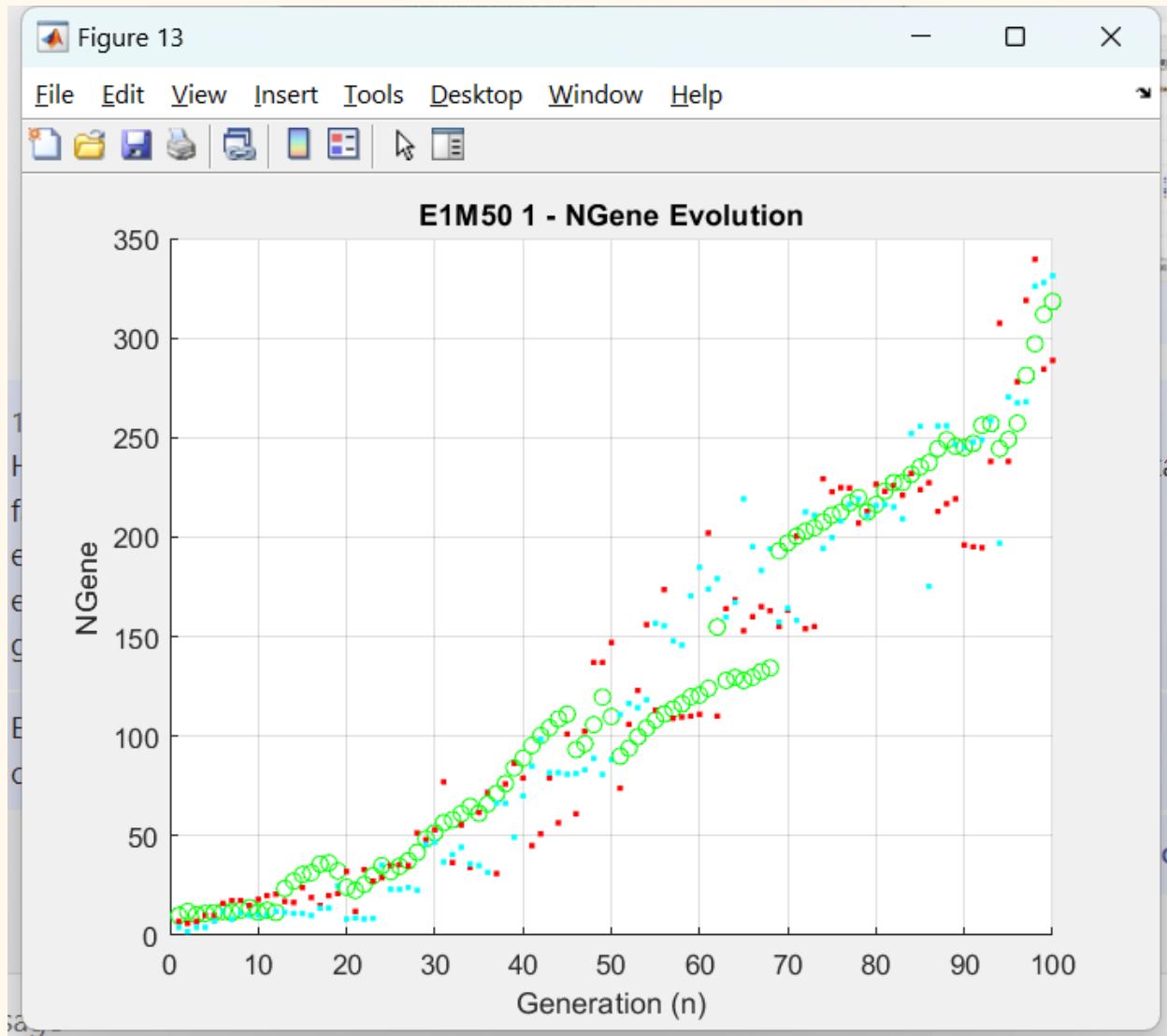


Durante las siguientes semanas, iteramos sobre el proceso de reentrenar todas las configuraciones vistas. Dando los siguientes resultados, que terminan convergiendo.



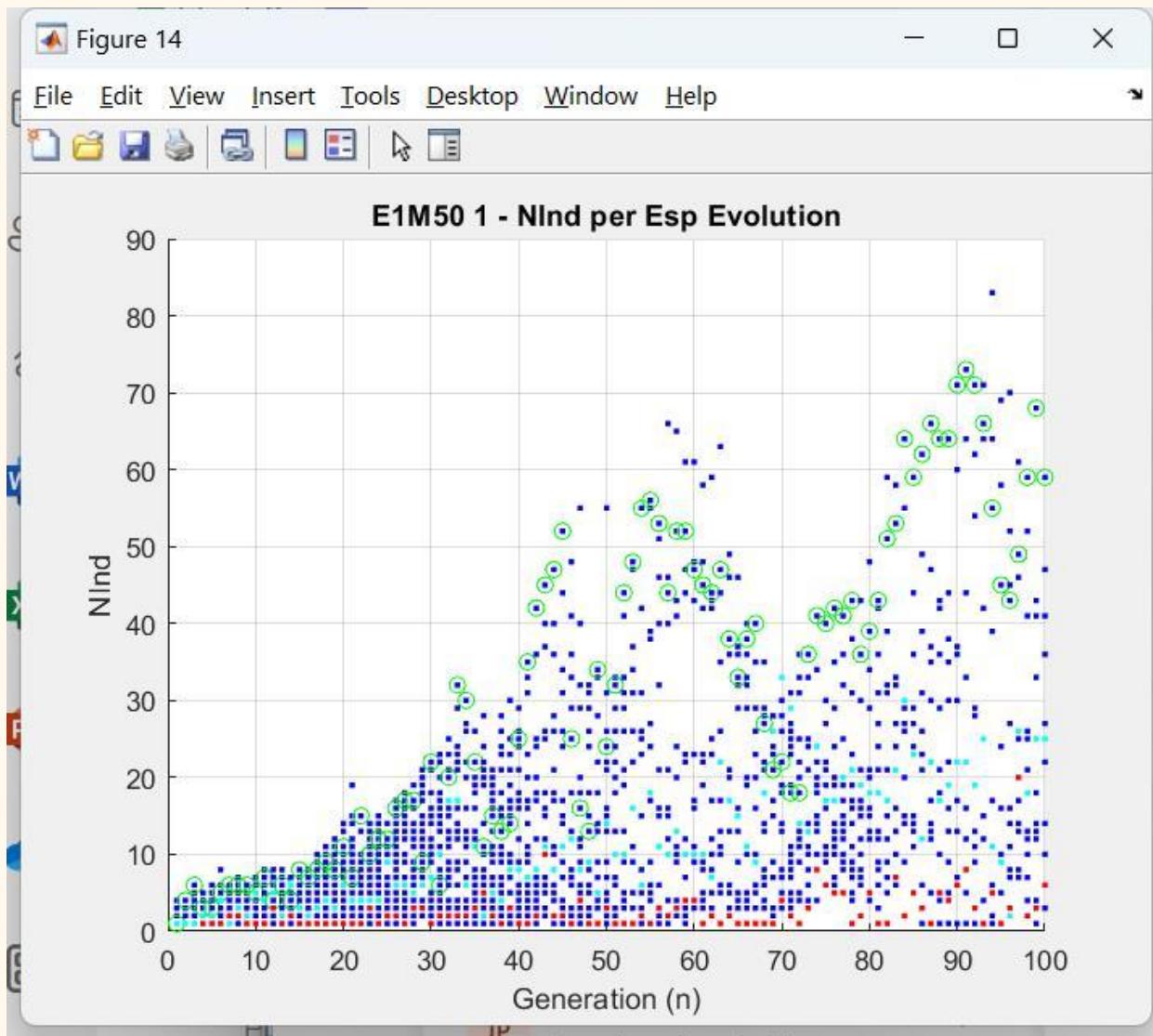
Las ejecuciones Elit1 y Elit3 parecen dar resultados similares, mientras que Elit5 se comporta peor.

Además podemos determinar, que no hay una correlación clara entre longitud del genoma y el fitness del jugador:



Se observa un periodo entre las generaciones 50 y 70 donde los individuos de mayor longitud (en verde) tienen peor comportamiento. Es decir, no porque la red sea más compleja da mejores resultados.

Por otro lado, el número de individuos por especie sí está en correlación con el fitness. Suponemos que es algo derivado del proceso de selección, que cruzará más a los mejores individuos.



Se ve como la mejor especie (verde) acaba teniendo entre 60 y 70 individuos, mientras que la peor (rojo) se queda por debajo de 10 individuos.

El algoritmo en su estado actual es insuficiente para ser eficaz en juegos complejos, pero es evidente que NEAT puede emplearse en videojuegos. Aun así es cierto que deben implementarse optimizaciones que permitan al algoritmo desarrollarse cómodamente. Por ejemplo, puede quedar trabajo agrupando píxeles para las neuronas de detección.

Unity

Mientras se terminaba la parte de comparar resultados y conclusiones sobre el algoritmo en Lua, se trabajó en la implementación de un juego relativamente sencillo en Unity.

Llevó alrededor de cuatro días ponerlo a punto. Hay un nivel con diferentes plantas y un sistema que lanza barriles con frecuencia. El jugador debe subir por las escaleras sorteando obstáculos para llegar a la meta (la princesa). El jugador puede moverse lateralmente, saltar y desplazarse verticalmente por las escaleras. Hay animaciones para los barriles, Donkey Kong, Mario y Pauline.

Además habrá que desarrollar una interfaz que permita al algoritmo comunicarse con cada jugador.

En un principio se intenta implementar el algoritmo directamente sobre el mismo proyecto del juego, pero la complejidad que supone eso para el proceso de debugging (buscar fallos) acaba arrastrándonos a crear proyectos aislados. Esto modulariza cada parte del proyecto total en partes más manejables, que cuando ya están listas pueden unirse con un par de ajustes.

Para hacer esa primera implementación fallida, seguimos un tutorial para implementar NEAT. Que aunque sea en Java puede trasladarse a otro lenguaje orientado a objetos sin problema.

[Tutorial NEAT](#)

Ahora implementamos desde cero y en otro proyecto una representación del genoma para ver visualmente el proceso de mutación y de cruce con otros genomas.

[2023-06-16 16-45-44.mkv \(sharepoint.com\)](#)

Queda una forma bastante interactiva de entender lo que pasa y qué es lo que puede fallar.

A continuación habría que conseguir implementar un algoritmo genético básico en Unity. El plan es hacer que funcione evaluando a todos los individuos a la vez, acelerando muchísimo el proceso con respecto a la versión de Lua. Esto se conseguirá con corrutinas, que son procesos asíncronos que proporciona Unity. Llevará unos días conseguirlo, pero finalmente:

[Genetico.mp4 \(sharepoint.com\)](#)

Ahora habría que adaptar el código para funcionar con NEAT. No hay que cambiar mucho más que la propia idiosincrasia del algoritmo. Sustituir el genoma, que en lugar de ser un vector de comandos será una red neuronal y añadir el concepto de especiación.

[NEAT.mp4 \(sharepoint.com\)](#)

Tras unos días conseguimos que funcione. Lo más complicado fue la especiación, porque hay que saber cuándo repartir los individuos en especies, cuando limpiar las especies y de qué manera, cuándo generar los hijos y sobre todo, implementar el requisito de la función de distancia. Esta función, como ya se explicó, permitirá la entrada de un individuo a una especie solo si tiene una distancia menor al límite. Para ello revisamos el tutorial en la parte de implementar las especies.

[Neat - Java Implementation 7 - Client and Species](#)

A pesar del funcionamiento limitado que demostró en el entrenamiento que se aporta en NEAT.mp4, vimos que algunos individuos superan la prueba. Así que traspasamos el algoritmo al juego definitivo. Los resultados son pobres, pero si da tiempo podemos tratar de mejorarlos un poco.

Para el juego de Donkey Kong hay 5 salidas para la red neuronal, una para cada control. Arriba, izquierda, abajo, derecha y salto.

Además hay 36 entradas. Esto, aunque parezca mucho es órdenes de magnitud inferior a la implementación de 1942, en la que teníamos en torno a 60000 entradas. Son las siguientes:

- ¿Está el jugador en el suelo?
- ¿Está el jugador escalando?
- ¿Está el jugador en posición para escalar? (Cerca de unas escaleras)
- La posición relativa al jugador de todas las escaleras en el nivel
- La posición relativa del barril más cercano
- La velocidad del barril más cercano
- La posición del jugador en X e Y

[NEAT Donkey 1.mp4 \(sharepoint.com\)](#)

• APORTACIONES:

En un mundo donde los videojuegos son cada vez más importantes en la economía, es pertinente pensar en los beneficios que puede aportar la aplicación de algoritmos genéticos para resolverlos.

Que una máquina juegue en lugar de una persona, no solo tiene interés por lo entretenido que puede resultar verlo. También puede utilizarse como herramienta de pruebas sobre el juego. Tal vez de manera única o quizás de forma complementaria al testing humano.

Que una máquina trate de completar el juego de forma automática y sin parar, buscando siempre una manera óptima de hacerlo puede ayudar a encontrar fallos, u otros comportamientos indeseables de parte del desarrollador.

Tenemos casos reales del empleo de agentes inteligentes para este ámbito⁷.

Y aunque en este caso se apliquen a un videojuego real, se pueden crear videojuegos que representen la realidad. Un simulador de conducción puede usarse para entrenar un algoritmo que aprenda cómo enfrentarse al tráfico real. O simplemente, como paso previo antes de aplicarlo al ordenador de un coche inteligente, donde seguirá aprendiendo por transfer learning. El transfer learning es una técnica que consiste en que un entrenamiento resulte familiar al algoritmo por aquellas cosas que ya pudo haber aprendido anteriormente en un caso parecido.

- **MANUAL DE USUARIO Y SOFTWARE:**

deberán incluirse obligatoriamente en la memoria los extractos más relevantes del código desarrollado. Siempre que sea posible, deberá proporcionarse acceso a un repositorio software.

- **NORMATIVA Y LEGISLACIÓN:**

Autorización del uso del plugin

En la cabecera del *plugin* proporcionado por SethBling^{1,2} se autoriza el uso del código, prohibiéndose la redistribución del mismo. Este proyecto no tiene intereses comerciales.

Obtención de la ROM de los videojuegos

No hay beneficio económico ni intereses comerciales, el uso del material de este proyecto es puramente académico.

• ASPECTOS ECONÓMICOS Y TEMPORALES

Para entender el coste económico de este trabajo primero tenemos que entender el coste temporal.

Entendiendo que un estudiante no puede cobrar como un ingeniero experimentado, se propone aplicar el salario mínimo a jornada completa. Esto es, 1.260 euros brutos al mes¹³. Una jornada completa son 40 horas semanales y completa 160 horas mensuales. El salario por hora sale a 7,875 euros/h brutos.

Sin calcular impuestos podemos aplicar el siguiente cálculo. Si el trabajo ha durado X horas, $X * 7,875$ euros = Y euros.

El tiempo que el algoritmo empleó entrenándose son tareas en segundo plano, y no se considerarán horas de trabajo. Fueron días (quizá superando un mes) de tiempo de cómputo que no se han valorado económicamente por no ser posible en las circunstancias en las que se realizaron (PC personal, domicilio familiar...).

Una estimación podría ser mirar el precio por mes de alquilar un servidor que haga el trabajo.

En [VULTR](#) tenemos un plan de 20\$ al mes por un servidor que parece suficiente para ejecutar Windows y Bizhawk. Menos que 4 GB podría hacer el sistema inutilizable. Se han estado entrenando algoritmos desde noviembre de 2022 hasta junio de 2023. 8 meses de alquiler, son 160\$. Que podemos añadir al cómputo total de Y euros para obtener un ejemplo más realista del coste.

• CONCLUSIONES Y TRABAJOS FUTUROS:

Durante el transcurso de este trabajo se han realizado trabajos de investigación, ingeniería inversa, datamining, lectura y aprendizaje de un paper científico, se ha jugado con los parámetros de un algoritmo, se ha desarrollado un videojuego en Unity y se ha hecho una implementación propia del algoritmo NEAT.

Podemos decir con certeza que se han cubierto las competencias de las asignaturas de la intensificación de computación del plan 2010 del Grado en Ingeniería Informática.

A pesar de que los resultados no siempre fueron llamativos, los objetivos se han alcanzado con éxito. Además el valor intangible que ha proporcionado el trabajo es muy alto.

A partir de aquí el trabajo podría continuar realizando optimizaciones que mejoren el rendimiento y funcionamiento del algoritmo, tanto en su versión de Lua como en la de Unity. De haber dispuesto de más tiempo, ese hubiera sido el foco.

Bibliografía y fuentes de información

1. Sethbling (2016). [MarI/O - Machine Learning for Video Games - YouTube](#)
2. SethBling (2016). [MarI/O Followup: Super Mario Bros, Donut Plains 4, and Yoshi's Island 1 - YouTube](#)
3. Data Crystal (2022). [Super Mario Bros.:RAM map](#)
4. Kenneth O. Stanley, Risto Miikkulainen (2002). [Evolving Neural Networks through Augmenting Topologies](#). Massachusetts Institute of Technology. Evolutionary Computation 10(2): 99-127
5. The8bitbeast (2021). [TAS Tutorial Part 4 - RAM Search and RAM Watch - YouTube](#)
6. The8bitbeast (2017). [TAS Tutorial - Finding RAM Addresses in Bizhawk](#)
7. Camilo Gordillo, Joakim Bergdahl, Konrad Tollmar, Linus Gisslen (2021) [\[2103.13798\] Improving Playtesting Coverage via Curiosity Driven Reinforcement Learning Agents \(arxiv.org\)](#). SEED - Electronic Arts (EA)
8. R. Ierusalimschy, L. H. de Figueiredo, W. Celes, Lua.org (2006). [Lua 5.1 Reference Manual](#)
9. TASVideos (2023). [Bizhawk/LuaFunctions - TASVideos](#).
10. Data Crystal (2022). [Bomberman II:RAM map - Data Crystal \(romhacking.net\)](#)
11. Data Crystal (2022). [1942:RAM map - Data Crystal \(romhacking.net\)](#)
12. Stack Overflow (2016). [Difference between math.random\(\) and math.randomseed\(\) in Lua - Stack Overflow](#)
13. La Información (2023). [¿Cuánto es lo mínimo que se puede cobrar por hora? Nuevas cuantías en 2023 \(lainformacion.com\)](#)