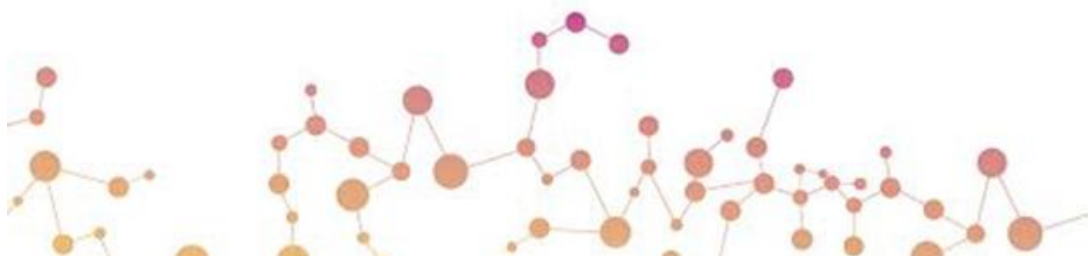


JUGANDO A SER DIOS

EXPERIMENTOS
EN VIDA ARTIFICIAL

MANUEL LÓPEZ MICHELONE



Jugando a ser Dios

Experimentos en vida artificial

Manuel López Michelone

Text Copyright © 2015 Manuel López Michelone

Todos los derechos reservados

Portada: GD Marga López

A la memoria de mi padre

Tabla de Contenidos

Introducción

Capítulo I

Capítulo II

Capítulo III

Capítulo IV

Capítulo V

Capítulo VI

Capítulo VII

Capítulo VIII

Capítulo IX

Capítulo X

Capítulo XI

Capítulo XII

Capítulo XIII

Capítulo XIV

Capítulo XV

Capítulo XVI

Conclusiones

Bibliografía

Apéndice I

Apéndice II

Apéndice III

Apéndice IV

Apéndice V

Apéndice VI

Apéndice VII

Apéndice VIII

Apéndice IX

Introducción

El seis de agosto del 2012 la sonda Curiosity tocó suelo marciano. Su misión era la de buscar vida en el planeta rojo, entre otras. Pero ¿qué significa exactamente esto? Sin duda la gran incógnita a resolver es si en algún momento hubo vida en Marte o si hoy en día la hay. Desde luego, no se buscan pequeños seres verdes, sino quizás organismos unicelulares, vida primitiva pero que nos demuestre de una vez por todas que no estamos solos en el vasto universo.

Hay sin embargo, una seria dificultad con esta búsqueda. Buscamos organismos que conocemos como vivos en la Tierra, basados en el carbono. Pero... ¿no podría ser que la vida se pudiese expresar en organismos basados por ejemplo, en el silicio o en otro elemento? Nuestro conocimiento de la vida es tan pequeño y no tenemos referencia de ninguna otra forma de vida más que la que en el planeta Tierra se ve, por lo que no nos queda más remedio que iniciar la búsqueda de la vida interplanetaria desde ese enfoque.

Curioso es que ni siquiera sabemos qué es exactamente la vida y esto ha sido motivo incluso de debates filosóficos. Tenemos ejemplos claros de lo que es viviente y lo que no. Pasa algo parecido con el tema de la inteligencia. Sabemos o intuimos que algo es inteligente, pero definirlo no parece fácil. Ocurre igual con el tema de la vida. Por ejemplo, una piedra es sin duda algo sin vida. Una hormiga en cambio sí es un ser viviente. Tal vez podríamos decir que lo que se mueve tiene vida, pero no es así necesariamente. Una piedra puede girar por alguna razón y aún sigue sin tener vida. Tendríamos entonces que decidir con base a más parámetros. La vida entonces podría definirse como algo que se mueve

autónomamente. Pero fallamos aquí. Hay unos pequeños juguetes que incluso simulan una mascota, un perro, que se mueve autónomamente, que incluso interactúa con el entorno, pero no está vivo.

Entonces ¿qué define a la vida? ¿Acaso debemos considerar a lo que respira? ¿O quizás a lo que se reproduce? En cualquiera de los casos, podremos hallar dificultades para definir el concepto de lo viviente. No obstante esto, el estudio de los seres vivientes, de la vida, es algo fascinante y fundamental.

Es importante porque las cosas más interesantes en el mundo son las que involucran a los seres vivientes. Lo vivo crece, se desarrolla, se alimenta, se junta con otros de su propia especie, se reproduce, compete con otros, evoluciona.

La vida la podemos estudiar desde varios enfoques, por ejemplo, a partir de sus reacciones químicas y físicas. Otra opción es a través de crear robots que actúen como si estuviesen vivos y así analizar su comportamiento. Una opción más es la de crear programas, *software*, que incluya los elementos lógicos que hallamos en la vida y así estudiar lo que pasa en estos mundos de vida virtual.

Esto último es lo que haremos aquí. Estudiaremos la vida de manera artificial – esto es lo que se llama **vida artificial**– utilizando sistemas creados por el propio hombre y que se comportan, al menos en principio, como si estuviesen vivos. El propósito es entender mejor cómo es que trabaja esto que parece eludirnos y que denominamos vida.

Una ventaja de este enfoque es que es económico comparado con los otros dos mencionados. Lo único que necesitamos es una computadora relativamente moderna, en la cual haremos todo género de experimentos en vida artificial. Crearemos autómatas celulares en una y dos dimensiones. Simularemos hormigas y observaremos su comportamiento en el monitor de la computadora. Haremos algo parecido a lo que hacen los biólogos cuando estudian organismos

vivos. La ventaja que tenemos es la manipulación conceptual de estos seres vivos virtuales, a veces imposible de hacer en el mundo real. Igualmente, veremos temas como el de la evolución, tratado como si fuese un videojuego y de ahí sacaremos valiosas conclusiones.

La vida artificial es pues más que un juego de computadora. Es poner a prueba los elementos que forman la vida misma de forma binaria, a través de la tecnología de la computadora digital. El camino es fascinante y las conclusiones asombrosas. Entremos pues en materia.

Manuel López Michelone

Julio 2015

Capítulo I

¿Qué es la vida?

*¡Oh insensato hombre, que no puede crear un gusano
y sin embargo crea dioses por docenas!*

Miguel de Montaigne

No cabe duda que la vida en general es un fenómeno extraordinario. Es fascinante ver como diferentes organismos se desarrollan y emergen en el planeta. Son autónomos, caminan, corren, se arrastran, nadan, reptan incluso. Contra los objetos inanimados, los organismos vivientes son francamente asombrosos.

La vida en el planeta es además enorme, frondosa. Pareciera que se abre paso a la menor provocación. Quizás por ello quedamos desconcertados al ir a otros planetas, como Marte, y no hallar ni vida unicelular. Y en cambio, en la Tierra, la vida tiene un sinfín de diversas manifestaciones. Su riqueza es extraordinaria.

Pero ¿qué caracteriza a la vida? ¿Qué es la vida? Una primera idea es esta autonomía que tienen los seres vivos, los cuales lo que no está vivo no lo tiene. Un insecto vivo se mueve mientras que una piedra no. Y si acaso se moviese la piedra, no hay voluntad propia de moverse. No tiene “intención”, como diría algún filósofo. Lo hace por efectos naturales, viento, gravedad, etcétera. Otro concepto fundamental es que los organismos vivos pueden reproducirse, pueden crear copias de sí mismos. De alguna manera buscan la supervivencia de su propia especie.

Hablamos pues que un organismo está vivo si puede organizarse, que hace algo por sí mismo. Un ser vivo busca alimentarse, crecer, mantenerse vivo. Conserva esta interesante propiedad de *auto-producirse* constantemente, que es lo que hace cuando desarrolla alguna actividad como comer, por ejemplo. En el largo desarrollo de la biología como ciencia, se fue llegando a una conclusión: lo que distingue a los seres vivos de los no vivos es simplemente la *organización*, que sólo poseen los primeros. Todas las funciones vitales de los individuos demandan un alto grado de organización.

Podemos quizás tomar otro camino, y éste tiene que ver con los elementos químicos que forman la vida. Los organismos basados en el carbono, junto con el hidrógeno, el nitrógeno y el oxígeno, son los elementos básicos que se hallan en todo lo que tiene vida, en todas las células. Quizás hay formas de vida basadas en otros elementos pero desde luego, no tenemos un solo ejemplo de ello y a lo más podríamos especular cómo podría ser la vida considerando otra química, sin embargo, el poder considerar otra posible química para los organismos nos podría enseñar mucho sobre cómo se manifiesta la vida y la riqueza de dicho proceso. De hecho, Chris Langton justifica así el estudio de la vida artificial, considerándolo como ***la biología de lo posible***.¹

Puede haber polémica sobre qué significa estar vivo, porque no parece ser una sola propiedad lo que caracteriza a los seres vivientes. El problema es añejo y ya Claude Bernard, en 1878, enumeró cinco características comunes de los seres vivos:

- Organización
- Reproducción
- Nutrición
- Desarrollo
- Susceptibilidad a la enfermedad y muerte

¹ *Artificial Life: The Proceedings of an Interdisciplinary Workshop of the Synthesis and Simulation of Living Systems*; Addison-Wesley, 1987

Por supuesto que Bernard no fue el único que se atrevió a definir la vida. Los fisiólogos dicen, por ejemplo, que *todo sistema que puede ejecutar funciones tales como la ingestión, metabolismo, excreción, respiración, movimiento, reproducción y reacción a los estímulos exteriores* es sin duda un sistema vivo. La dificultad con esta definición es que tenemos máquinas que bien podría considerarse que hacen algunas de estas funciones. Por otra parte, sabemos de bacterias que no respiran oxígeno. Aparentemente no existe una definición completa y amplia de la vida. Todos los intentos encuentran contraejemplos que invalidan uno o varios de los conceptos definitorios.

Pero si vamos a la vida artificial, ¿cómo podríamos caracterizarla, basándonos en lo que sabemos de los organismos que consideramos vivos? Probablemente, al menos en el estado actual de las cosas, la vida artificial parece ser una metáfora o analogía de la vida en el mundo real. Los autómatas de von Neumann son de alguna manera un ejemplo de “seres vivos” que se reproducen, que se auto-repican. ¿No es suficiente para pensar que están vivos de alguna manera? O pensemos en los virus informáticos, que pegan su “código genético” en otros programas y cambian el comportamiento de los mismos de maneras poco amables con los usuarios. ¿No hablamos entonces de alguna manera de algo vivo?

La comparación entre el formalismo de lo que conocemos en un ser vivo y un programa de computadora bien puede enseñarnos los elementos que hacen que la vida sea como la conocemos. Las instrucciones de un programa de computadora, por ejemplo un virus informático, bien podría ser considerada la química artificial de este “ser vivo”. ¿Por qué no?

Podríamos llegar a una serie de conclusiones como punto de partida para comprender la vida artificial:

- Los virus de computadora son estructuras informáticas y no objetos materiales.
- Los organismos digitales (virus, por ejemplo), son capaces de reproducirse.
- La reproducción de estos “organismos” se basan en la recursión y en el control de la misma.
- Pueden estos organismos interactuar en un entorno virtual.

Tal vez la vida artificial sea el siguiente paso para entender más la vida biológica, la de los seres vivos. Podría finalmente sentar las bases de la vida a partir de argumentos de la lógica. Hay camino recorrido, pero es claro que apenas vamos avanzando.

John von Neumann y los autómatas celulares

Siempre me he preguntado si un cerebro como el de von Neumann no es un indicativo de una especie superior al hombre.

Hans Bethe

Uno de los personajes más influyentes en la ciencia, en el siglo pasado, fue sin duda John von Neumann. Nacido en Budapest, el 3 de diciembre de 1903, von Neumann fue reconocido a muy corta edad como un prodigio. Por ejemplo, podía hacer divisiones con dos números de ocho cifras en su cabeza. Entretenía a amigos y visitas de sus padres memorizando columnas de nombre, direcciones y teléfonos. Una anécdota cuenta que en una ocasión su madre estaba tejiendo cuando de pronto se detuvo mirando hacia la nada. El pequeño John entonces le preguntó: “¿Qué es lo que estás calculando?”

A los veinte años von Neumann dio una definición formal de los números ordinales y ya a los veinticinco era un matemático de primer orden, el cual atacaba problemas de la mecánica cuántica a través de una teoría vectorial. De hecho fue von Neumann quien descubrió que los estados de los sistemas cuánticos podían ser representados por vectores en un espacio n -dimensional. Para 1931, año en el

que emigró a los Estados Unidos, era uno de los matemáticos más importantes en el planeta.

Von Neumann trabajaba en muchos temas. Su capacidad era notable y en algún momento, después de la Segunda Guerra Mundial, se preguntó acerca del fenómeno de la vida. Una característica que parecía definirla era esa capacidad de auto-reproducción que comparten los seres vivos. ¿Sería esto lo que diferencia a la vida de la no vida? Von Neumann pensó entonces sobre la posibilidad de que una máquina creara un duplicado de sí misma. ¿Habría una limitación en este sentido? Alguna contradicción de estos sistemas que él estaba pasando por alto? ¿Qué tal pensar en una máquina que creara una más compleja que la original? ¿Sería eso factible? Y con ese mismo criterio habría que considerar que si esto fuese posible, la segunda máquina podría crear una más compleja y así sucesivamente. La idea era absolutamente fundamental y von Neumann, que trabajaba incansablemente, decidió dar una respuesta a estas interrogantes.

Su primer intento en el campo de *la vida artificial* fue la concepción de una máquina, una especie de robot, que podría hacer una copia de sí mismo. Von Neumann describió el asunto como un robot que flota en un lago. También flotan piezas y componentes que permiten a éste construir otro robot. Él pensaba entonces que se podía diseñar un robot –al menos en teoría– que se copiara a sí mismo. Desde luego las dificultades tecnológicas de los años cincuenta del siglo pasado (e incluso hoy día), hacen imposible aún que exista físicamente semejante robot auto replicable, pero en esencia el problema estaba planteado. A esto se le llamó el *modelo cinemático*, propuesta de quien terminara la formulación de von Neumann, Arthur W. Burks.

Aunque von Neumann podía demostrar la capacidad de su modelo para crear un robot que fuese copia de sí mismo, no estaba conforme en el sentido que su imaginario robot tenía que buscar cada pieza necesaria para replicarse en una sopa de componentes. Por ello, la sugerencia de Stanislaw Ulam, otro fantástico

matemático, de crear un modelo en dos dimensiones, de células, con reglas que permitiesen crecer, desarrollarse y en todo caso, auto reproducirse, parecía una mejor opción. De esta manera, el problema se simplificaba a un universo bidimensional en donde células luchan por crecer y multiplicarse bajo ciertas reglas perfectamente conocidas.

Ulam estaba fascinado con la idea de los patrones y compartía estos intereses de Von Neumann. Dice Ulam en su autobiografía:² *Cuando tenía cuatro años estaba viendo en el piso un tapete oriental que tenía una serie de patrones visuales dibujados. Recuerdo a la figura de mi padre, alto como una torre, parado a mi lado, que sonreía. Él sonríe –comenta Ulam– porque piensa que soy un niño pequeño pero yo sé que estos son patrones curiosos. Ulam indica que probablemente éstas no fueron sus palabras exactas, pero dice: yo sentí que –definitivamente– sabía algo que no sabía mi padre. Tal vez incluso sabía más al respecto que él.*

Ulam, usando las novedosas computadoras de Los Álamos, jugaba a inventar juegos de patrones que dadas ciertas reglas fijas, cambiaban con el tiempo. A estos patrones Ulam los llamó “objetos geométricos creados recursivamente”. Cabe decir que la recursión es la forma en la cual se especifica un proceso basado en su propia definición, la cual aparentemente no parece tener sentido en algunos casos pero que es muy útil y usada en matemáticas y cómputo³, por ejemplo.

² **Adventures of a Mathematician**; S.M. Ulam; *University of California Press*, 1976.

³ De niños nos enseñan que lo definido no puede estar en la definición, pero esto no es cierto. Por ejemplo, podemos definir la operación factorial, que se anota con ‘!’ de forma recursiva. Decimos que $n! = n (n - 1)!$ Y además que $0! = 1$. Así entonces, $3! = 3 (2!)$; pero $2! = 2 (1!)$; pero $1! = 1 (0!)$; pero $0! = 1$ entonces podemos regresar en los cálculos que tenemos pendientes: $1! = 1 (0!) = 1$; $2! = 2 (1!) = 2$; $3! = 3 (2!) = 6$. Hay una historia en donde se decía que *Iteratum humanum est, recursivum divinum est* cuyo autor era el creador del lenguaje Pascal, Niklaus Wirth (15/2/1934 –). Sin embargo, el propio Wirth ha desmentido constantemente esta afirmación y no se sabe a ciencia cierta porqué se le adjudica a él.

Ulam halló que los patrones eran muy sofisticados y entonces le propuso a su amigo John Von Neumann que “construyera” un universo abstracto para su análisis de las máquinas auto-replicantes. Desde luego que para ello habría que considerar tener reglas muy claras, inmutables, que gobernaran este “universo” en miniatura. Las reglas deberían ser lo suficientemente sencillas y entonces éstas podrían considerarse algo así como *las leyes de la física en este universo alterno*. Con ello podría demostrarse la idea de la creación de una máquina que se creara a sí misma, sin tener que pasar por la problemática de la construcción real.

La idea le llamó poderosamente la atención a von Neumann. Éste desarrolló un mundo en una gran cuadrícula como si se tratase en un mundo bidimensional. A esto Ulam le llamaba “juegos de autómatas”, en donde cada patrón eran celdas acomodadas de cierta manera en un tablero cuadriculado que en principio no tenía fronteras. Se aplicaban las reglas definidas y se veía la siguiente generación de las “células”. Dichas reglas en general se basaban en observar las celdas contiguas a la celda de una célula y aplicarles una operación que bien podía duplicar el resultado o quizás aniquilarlo. Así pues, la suerte de las células se definía por los estados de las células vecinas, las que estaban adyacentes.

La ventaja de este enfoque es que su estructura de reglas era muy simple. Von Neumann adoptó este tablero de celdas ilimitado y en ellas podría haber cualquier número de estados y su distribución inicial era un patrón, una máquina de dichas celdas.

En poco tiempo el científico llegó a tener una concepción completa. Aquí, las células podían tener hasta 29 estados. Un estado podría ser tener un espacio vacío (donde podría ir una célula). Los otros 28 estados eran diversas maneras que podían tener las células en este espacio de dos dimensiones.

Lo que trataba de hallarse era entonces un mecanismo que permitiera la auto replicación. Las células podrían tener una descripción codificada de su propia auto

organización, lo cual quizás podríamos verlo como un mapa genético actualmente. En este mapa, habría un constructor universal y basándose en las ideas de Turing sobre la máquina universal que había concebido, se podía coordinar todo este proceso.

La reproducción, de acuerdo a von Neumann sería de la siguiente manera: un patrón leería su propio plano, su propio mapa genético –valga la expresión. Éste mandaría la información a su constructor universal. Este constructor entonces tendría una especie de brazo, una península de células activas, que se movería creando nuevas células. Una nueva copia de este patrón se auto-replicaría casi de la misma forma en cómo se crean las imágenes en un televisor analógico, línea por línea.

Von Neumann no tuvo posibilidades reales de ver si su autómatas podía hacer efectivamente la tarea de auto reproducción, pero en su mente estaba claro que eso era literalmente una obviedad. Considerando lo primitivo de las computadoras de la época, simplemente no había manera de programar las ideas de von Neumann. Sin embargo, ahora esto ya es desde luego posible. A pesar de la complejidad del sistema de autómatas celulares de von Neumann, ya existen programas que permiten probar las ideas en cuestión⁴.

Pero hay más dificultades: los organismos vivos son finitos y se reproducen en un tiempo finito. El problema con los autómatas de von Neumann es que el constructor universal sigue reglas ciegas muy precisas y por ende, genera una réplica de la máquina original sin siquiera saber lo que está haciendo y no sabe

⁴ Umberto Pesavento, entre 1992/94, con 16 años de edad, programó las ideas de von Neumann, asesorado por Renato Nobili, que había emprendido esta tarea unos años atrás. El programa funciona en la plataforma Windows y fue compilado usando Watcom C++. El resultado fue presentado en la conferencia *Artificial Worlds and Urban Studies*, en Venecia, Italia, a fines de 1994. Dos años después se publicó un artículo denominado *An Implementation of von Neumann's Self-Reproducing Machine* (*Artificial Life Journal* **2**, 337-354) cuando ya Pesavento estudiaba física en Princeton, en los Estados Unidos. El programa, con una profusa ayuda, puede descargarse de http://www.pd.infn.it/~rnobili/au_cell/.

dónde detenerse causando una regresión infinita. Por ello, von Neumann decidió incorporar un mecanismo extra, el cual llamó “unidad supervisora” la cual genera una dificultad inesperada: hace a la máquina por replicar más compleja aún. De esta manera se evita la replicación infinita pues el mapa describe al constructor universal y a la unidad supervisora. Cuando hay que hacer un nuevo mapa, éste es su propio mapa. Ahora entonces se interpreta de dos formas, primero como un conjunto de instrucciones que deben seguirse para hacer un cierto tipo de máquina. Entonces la unidad supervisora cambia al segundo modo, las instrucciones en el mapa son ignoradas. El mapa solamente es entonces el material para realizar la copia.

Lo notable de esta descripción de los autómatas definidos por von Neumann es que siguen el modelo reconocido de lo que hace el ADN para replicarse. Del cómo el código genético de los seres vivos se replicaba era un misterio en la época del desarrollo de los trabajos de von Neumann. Watson y Crick demostraron que el mecanismo descrito por el científico húngaro/norteamericano era prácticamente el mismo en el ADN y entonces se llega a la conclusión inevitable que no se necesita una fuerza mística o ininteligible (como la fuerza vital de la que hablaba el extraordinario físico Niels Bohr), para entender cómo se replican las cadenas de ADN. Esto caló hondo en la mente de los biólogos de la época, sepultando de forma inevitable la idea de esta fuerza de vida, de esta *fuerza vital*, era la que definía la diferencia entre seres vivos y objetos sin vida. De hecho, los biólogos han adoptado el punto de vista de von Neumann que en esencia dice que **la auto-replicación finalmente es organización**, es decir, la habilidad de un sistema para contener una descripción de sí mismo y usar esa información para crear nuevas copias.

La idea de una unidad supervisora choca en un principio, pero quizás en la concepción de von Neumann sea una necesidad. Sin embargo, en cómputo tenemos ejemplos en donde la regresión infinita se impide cuando se utiliza una función recursiva. En cierto sentido, la recursión es un proceso basado en sí

mismo, en su propia definición. Esto conlleva regresión infinita a menos que se planteé una condición que elimine dicha situación. Por ejemplo, en Prolog podemos definir un predicado, *member*, el cual nos puede decir si un elemento es parte de una lista. Para ello, lo definimos de esta manera:

```
member(X, [X|_]) .      % member(X, [Cab|Cola]) es cierto si X = Cab
                        % esto es, si X es cabeza de la lista
member(X, [_|Tail]) :- % o si X es miembro de la cola,
    member(X, Tail) . % si member(X, Cola) es verdadero.
```

Aquí el mecanismo de la recursividad tiene una “unidad supervisora” que es simplemente una sola instrucción, la primera (*member(X, [X|_]) .*), que se llama en muchos casos la “condición terminal o de salida”. Esto termina de tajo con la regresión infinita.⁵

Dicho de otra manera, la unidad supervisora de von Neumann quizás no necesita ser tan compleja como podría esperarse en un inicio. Simplemente decide el modo de actuar de acuerdo a una simple condición binaria, de verdadero o falso.

Von Neumann finamente pudo probar con su juego de patrones celulares lo siguiente: ***Hay patrones que pueden reproducirse a sí mismos***. Si se empieza con un patrón que se puede auto replicar, eventualmente tendremos dos de ellos, después cuatro, ocho, etcétera. Y el científico pudo además demostrar lo que sospechaba desde un principio: que incluso con estas reglas sencillas la creación de máquinas que se auto reprodujeran sería un asunto complejo, el cual le llevaba unos 200,000 cuadrados en el tablero en donde ocurría la simulación.

Aparte de esto, von Neumann llegó a demostrar que la auto-replicación era posible en un universo simplificado sin contradicciones inherentes. Si esto es así, entonces es posible crear máquinas que se puedan auto-replicar en el mundo real.

⁵ Los programas en el lenguaje funcional Prolog se ejecutan de arriba hacia abajo. De hecho, ésta es la razón por la cual la condición terminal debe ir antes de la función recursiva. Si se ponen al revés, el resultado es que se cicla el programa o se detiene por falta de memoria en el *stack*.

Y aunque nadie ha logrado semejante situación, ya nadie pone en duda de que esto es posible⁶.

Von Neumann no solamente mostró que podía crear una máquina que se podía auto-replicar, sino que además demostró que una máquina puede crear otra incluso más compleja que la original. Y aunque este hecho no ha tenido aplicación práctica parece ser evidente que esto ocurre en la biología y la prueba más fehaciente de esto es la evolución del ser humano, asunto que no se discute más.

Y es claro que las máquinas de Von Neumann no son seres vivos pero lo que se demostró es fundamental: que una serie de reglas basada en la lógica eran suficientes para hacer máquinas que pueden reproducirse a sí mismas. Y si regresamos a la biología, es razonable pensar en las células de los seres vivos como máquinas complejas que se auto replican y en esos términos, la analogía con las máquinas de Von Neumann es bastante adecuado.

En resumen, von Neumann logró prácticamente crear una definición de vida basándose en la teoría de la información, esto bajo el supuesto de que la reproducción biológica es en última instancia un evento meramente mecánico. He aquí las valiosas conclusiones de su trabajo:

1. Un sistema vivo encapsula una descripción de sí mismo.
2. Impide la regresión infinita (la recursión) incluyendo una descripción **de** la descripción **en** la descripción.
3. Así, la descripción tiene un doble papel. Es una descripción codificada del resto del sistema y al mismo tiempo, es una especie de modelo de trabajo (que no requiere decodificarse) del mismo.
4. Hay una parte del sistema que se llama “unidad supervisora” que sabe de este rol dual de la descripción y se asegura que la descripción se interprete de ambas maneras durante la reproducción.

⁶ Es claro que los seres humanos son un ejemplo de *máquinas* que se auto-reproducen.

5. Hay otra parte, llamada “constructor universal”, que puede construir cualquier clase de objetos, incluyendo el sistema vivo, dada las instrucciones correctas.
6. La reproducción ocurre cuando la unidad supervisora instruye al constructor universal a construir una copia nueva del sistema, incluyendo la propia descripción.

Es importante aclarar que von Neumann, al hablar de auto replicación (o de reproducción) de los autómatas celulares, está hablando de recursión, en donde el mapa genético está en el mismo autómata. Vamos, que lo definido está en la definición y esto es precisamente una condición indispensable para que haya la auto-replicación.

Capítulo III

El juego de la vida de John Conway

*Un matemático es un hechicero que revela sus secretos.*⁷

John Horton Conway

Hay un juego de computadora fascinante, llamado *Juego de la Vida*, el cual fue diseñado en 1970 por el matemático británico John Horton Conway⁸, de la Universidad de Cambridge, Inglaterra (en ese tiempo). Se hizo muy popular desde que Martin Gardner, en su columna de octubre de ese año en la revista *Scientific American* hablara de las ideas de dicho matemático. Pero más allá de ser un interesante pasatiempo, podríamos decir que el juego de la vida contiene las ideas que originalmente von Neumann intentó plasmar en su autómata celular. Lo importante aquí es que Conway halló una serie de reglas simples, para su autómata celular en dos dimensiones, que permitió superar las dificultades que von Neumann tuvo en su momento para crear máquinas que se auto-replicaran.

El juego de la vida ocurre en un tablero cuadrulado, en donde cada casilla o escaque puede haber una célula o estar vacío. La idea es acomodar una serie de

⁷ **Open problems in communication and computation**, T.M. Cover and B. Gopinath, editores, Springer, 1987; p. 9.

⁸ John Horton Conway (26/12/1937). Matemático británico cuyos campos de acción son la teoría de conjuntos finitos, teoría de juegos y teoría de números entre otras. Por muchos años estuvo en Cambridge. Hoy trabaja en la Universidad de Princeton. Es probablemente más conocido precisamente por la invención del “juego de la vida” (*Life*).

células en la malla cuadriculada y observando las vecindades de cada célula, cada cuadro pues, utilizando las reglas de Conway (ver más abajo), ir calculando las nuevas configuraciones de células que aparecerán en la siguiente generación.

Puede verse que el juego de la vida de Conway no es estrictamente un juego de video, pues el “jugador” no hace nada más que ver la evolución que en cada tiempo, en cada generación, aparece en la pantalla. Las reglas de evolución en el juego de la vida son:

- No ha de haber ninguna configuración inicial para la que pueda demostrarse fácilmente que la población crecerá ilimitadamente.
- No deben existir configuraciones iniciales que aparentemente crezcan indefinidamente.
- Han de existir configuraciones iniciales sencillas que crezcan y cambien durante períodos de tiempo considerables, antes de acabar en una de estas tres posibilidades:
 - (a) extinguirse completamente (ya sea por superpoblación o por excesivo enrarecimiento)
 - (b) adoptar una configuración estable, invariable en tiempos sucesivos o
 - (c) entrar en fase oscilatoria, donde se repitan sin fin, cíclicamente dos o más estados.

Cabe decir que las reglas que se definan para el juego de la vida deben ser tales que la conducta de la población resulte a un tiempo interesante e impredecible. Las reglas de evolución, dadas por el propio matemático, llamadas también *reglas genéticas*, son de una sencillez deliciosa y pensamos que Conway las fue descubriendo (¿inventando?) poco a poco.

Tomemos un plano cuadrículado de dimensiones infinitas. Cada sitio, cuadro o casilla, tiene 8 casillas vecinas: cuatro ortogonalmente adyacentes, en diagonal, 2

en vertical y 2 en horizontal. En cada sitio es posible poner un valor binario (hay célula o no hay en esa casilla). Las reglas son:

- **Supervivencia:** cada célula o ficha, que tenga dos o tres fichas vecinas sobrevive y pasa a la generación siguiente.
- **Fallecimiento:** cada ficha que tenga cuatro o más vecinas muere y es retirada del tablero, por sobrepoblación. Las fichas con una vecina o solas fallecen por aislamiento.
- **Nacimientos:** cada casilla vacía, adyacente a exactamente tres cifras vecinas -tres, ni más ni menos- es casilla *generatriz*. Es decir, en la siguiente generación habrá de colocarse una ficha en esa casilla.

Es importante hacer notar que todos los natalicios y fallecimientos ocurren simultáneamente, y constituyen en su conjunto una generación en particular, al paso del tiempo t , también llamado *tíc* del reloj.

Se puede descubrir que para ciertas configuraciones iniciales de sitios distintos de cero (casillas vacías), las generaciones subsiguientes van experimentando cambios constantemente, algunos parecen insólitos y otros inesperados. En algunos casos la sociedad termina por extinguirse (al quedar eliminados todos los sitios en donde hay células), y esto puede acontecer después de muchas generaciones (pasos del reloj). Sin embargo, casi todas las generaciones terminan por alcanzar figuras estables, que Conway bautizó como *naturalezas muertas*, incapaces de cambio, o formaciones que oscilan por siempre.

En un principio el inventor de este singular “juego” conjeturó que ningún patrón inicial podría crecer ilimitadamente. Dicho en otras palabras, ninguna configuración compuesta por un número finito de fichas puede crecer hasta rebasar un límite superior finito, que limita el número de fichas que puede contener el campo del juego. Seguramente esta sea la cuestión más difícil y profunda que planteé este pasatiempo.

Conway ofreció un premio de 50 dólares a la primera persona capaz de probar o de refutar la conjetura antes de finalizar el año 1970. Una forma de refutarla sería dar con configuraciones que, generación tras generación, añadiesen más piezas al terreno de juego: *un cañón* (es decir, una configuración que repetidamente dispara objetos en movimiento), o bien el tren *puf-puf* (configuración que al paso del tiempo t avanza dejando tras de sí una estela de “humo”).

La conjetura de Conway se refutó en noviembre de 1970. Un grupo integrado en el proyecto de inteligencia artificial del MIT, comandado por William Gosper, halló un cañón lanza-deslizadores, el cual genera un deslizador cada 30 pulsos de reloj (o generaciones). La existencia de tal cañón suscita una interesante posibilidad de que el juego de Conway pueda simular una máquina de Turing⁹, la cual es capaz de hacer, en principio, cualquier cálculo. Si el juego permite esta alternativa, entonces la siguiente pregunta a resolver es si sería capaz de poderse crear un constructor universal. De lo cual se encontraría una máquina con capacidad de auto reproducción no trivial. Desafortunadamente, a la fecha, no ha podido construirse.

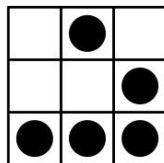
La máquina de “Turing” fue descrita por su propio creador, Alan Turing, en 1936, quien la llamó una “máquina automática”. Esta “máquina” no está diseñada como una tecnología de computación físicamente, sino como un dispositivo hipotético que representa una máquina de computación.

Turing dio una definición sucinta del experimento en su ensayo de 1948, “Máquinas inteligentes”. Refiriéndose a su publicación de 1936, Turing escribió que la máquina de Turing, aquí llamada *una máquina de computación lógica*, consistía en:

⁹ Una máquina de Turing es una máquina universal que puede ejecutar cualquier cálculo. De hecho, el juego de la vida de Conway —está demostrado por el propio matemático— es una máquina de Turing.

...una ilimitada capacidad de memoria obtenida en la forma de una cinta infinita marcada con cuadrados, en cada uno de los cuales podría imprimirse un símbolo. En cualquier momento hay un símbolo en la máquina; llamado el símbolo leído. La máquina puede alterar el símbolo leído y su comportamiento está en parte determinado por ese símbolo, pero los símbolos en otros lugares de la cinta no afectan el comportamiento de la máquina¹⁰.

La cinta puede ser movida hacia adelante o hacia atrás, lo cual se definen como operaciones elementales de la máquina de Turing. Lo interesante es que el juego de la vida de Conway puede caracterizarse de acuerdo a esta definición. En principio es posible usar configuraciones muy específicas, como las de los *deslizadores*, para llevar a cabo cualquier cómputo que pueda efectuarse con la más potente de las computadoras digitales. Mediante la disposición de cañones lanza-deslizadores y otras formas de “vida”, es posible calcular π , e , la raíz cuadrada de 2, o de cualquier otro número, con cualquier número de cifras decimales.

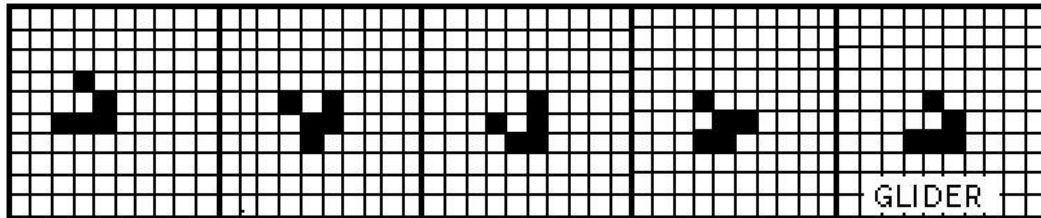


Un deslizador (glider) del juego de la vida

No obstante, se ha encontrado que el juego de la vida es finalmente un autómata celular bidimensional, el cual aparentemente no podemos saber su estado en la generación n sin hacer la simulación explícita. Esto es, por la teoría de la indecibilidad sabemos que no hay manera de saber por adelantado si un problema cualquier es resoluble o no, y por consiguiente, no hay formas de saber anticipada si en el juego de la vida dada una configuración cualquiera, continuará cambiando

¹⁰ **Intelligent Machinery**, Allan Turing, *National Physical Laboratory (Report)*, 1948; pág. 61.

o si alcanzará un final estable. Conway mismo, en una carta a Martin Gardner comenta: "... si los deslizadores pueden formar por colisión un pentadecatlón, entonces se pueden producir máquinas auto replicantes; la cuestión de si una máquina dada es auto replicante no es decidible".



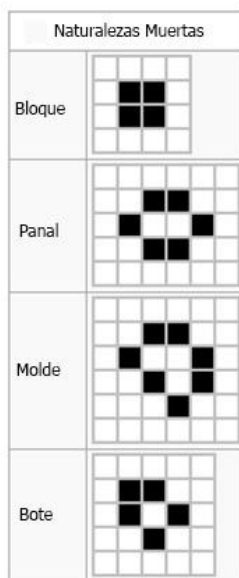
Desarrollo en el tiempo del deslizador (glider)

Sin embargo, es un hecho comprobado que los deslizadores pueden crear pentadecatrones al colisionar, por lo que en el espacio de configuración del juego es posible construir máquinas auto replicantes, es decir, máquinas que construyan copias exactas de sí mismas. La máquina primitiva puede permanecer en el espacio, o bien ser programada para que se autodestruya para cuando haya sacado una copia de sí misma. Hasta ahora no se sabe de nadie que haya construido una máquina de este tipo, pero si Conway está en lo cierto, entonces es posible construirlas.

Probar diferentes configuraciones de "células" en el juego de la vida es la manera más simple de entender lo que puede pasar a través de las generaciones. Estos son finalmente los patrones básicos de este juego. Hay dos maneras de realizar este proceso, una es "a mano", es decir, usando una hoja cuadriculada, dibujando las celdas y viendo su desarrollo de generación en generación de acuerdo a las reglas que Conway diseñó. El problema de este enfoque es que es fácil cometer errores al crear la siguiente generación. La segunda posibilidad es mucho más cómoda: utilizar algún programa de computadora que permita dibujar las células en una cuadrícula (en la pantalla), y hacer que el software mismo nos muestre cómo se desarrollan y evolucionan las siguientes generaciones. Esta evita errores humanos y además, hay muchísimos programas de código abierto que permiten

interactuar con el juego de la vida¹¹.

Por ejemplo, al empezar a jugar, poniendo diferentes configuraciones de células, hallamos patrones simples y estáticos, es decir, que no cambian a través de las generaciones. A ellos el propio Conway les llamó “naturalezas muertas (*still lifes*)”. Los más comunes son:

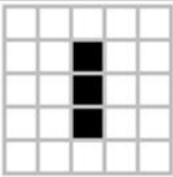
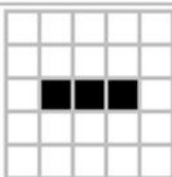
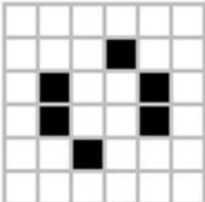
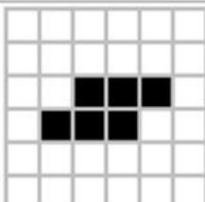
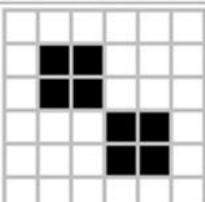
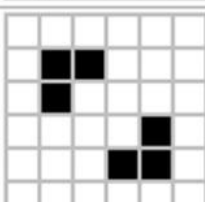


Algunas de las ‘Naturalezas Muertas’

Cuatro células unidas forman un **bloque**, el cual se mantiene así por el resto de las generaciones. La **colmena o panal** (*beehive* en inglés), se forma con seis celdas que ya no cambian en el tiempo. Otra configuración común es el **zángano** o **molde** (*loaf*) y el **bote**, las cuales son todas estáticas.

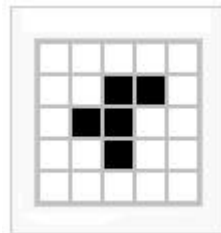
Sin embargo, tenemos otras configuraciones que cambian de generación en generación, que son estables y que además son oscilatorias. Algunos consideran éstas como un súper-conjunto de las naturalezas muertas. Las siguientes tienen período 2, es decir que cada dos generaciones se repite la configuración inicial:

¹¹ Por ejemplo, <http://www.delphidabbler.com/software/life> es uno de tantos sitios en donde hay software para jugar con la idea de Conway (ver apéndices).

OSCILADORES		OSCILADORES	
Blinker (período 1)		Blinker (período 2)	
Sapo (período 1)		Sapo (período 2)	
Faro (período 1)		Faro (período 2)	
Generación 1		Generación 2	

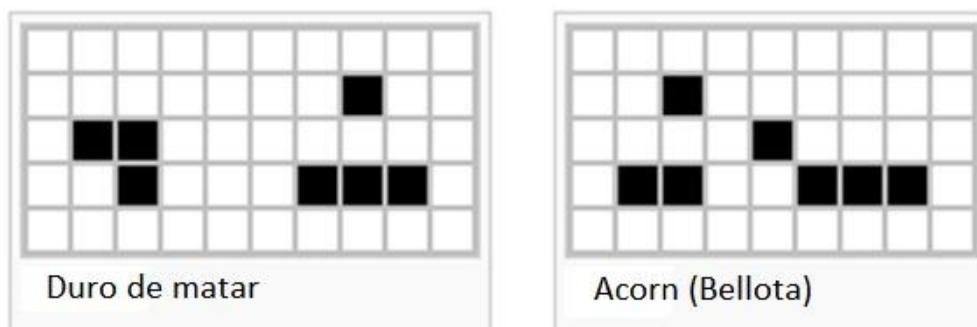
Aquí tenemos el **parpadeante** (*blinker*), el **sapo** (*toad*) y el **aviso** o **faro** (*beacon*). Estas tres configuraciones son también bastante comunes. Hay, evidentemente, otras que tienen períodos más largos. Por ejemplo, el **pulsar** es el oscilador más común con un período 3 de oscilación. La mayoría de los osciladores son de período 2, pero hay otros que tiene períodos mucho más largos. Se han hallado osciladores con período 4, 8, 14, 15, 30, incluso a partir de configuraciones iniciales puestas al azar.

Hay patrones llamados “Matusalén” debido a que pueden evolucionar por largos períodos de tiempo (generaciones), antes de que se estabilicen. El primero de ellos fue el F-pentómino.



F-pentómino

Otro muy interesante en esta categoría es el llamado “Diehard”(duro de matar), el cual es un patrón que eventualmente desaparece (después de mucho rato antes de estabilizarse) después de 130 generaciones. Se ha conjeturado que éste es el número máximo de generaciones para los patrones con 7 o menos celdas. El patrón “Acorn” (bellota) toma 5206 generaciones para crear 6333 células, incluyendo 13 naves (*gliders*), que se ven que escapan.



Los patrones DieHard (duro de matar) y Acorn (bellota)

El juego de Conway es uno de los más simples en términos de los sistemas de “complejidad emergente” o “sistemas auto-organizados”. Esto tiene implicaciones muy importantes en el tema de la vida artificial, sin duda. Es, por ejemplo, un estudio de cómo elaborar patrones y comportamientos que puedan emerger de reglas tan simples. Ayuda esto a entender finalmente algunos fenómenos naturales, como por ejemplo, ¿por qué las franjas de los tigres o cebras tienen esas formas?¹² De ahí probablemente el gran interés que suscitó.

En el juego de la vida, como en la naturaleza, hay fenómenos fascinantes sin

¹² En términos generales, el juego de Conway es un autómata celular en dos dimensiones. Poder explicar las manchas que pigmentan la piel de algunos animales, incluso alguna familia de serpientes, se puede hacer utilizando un autómata celular en una sola dimensión, el cual bajo reglas tan o más simples que las diseñadas por Conway, pueden generar patrones extremadamente complejos. Steven Wolfram (el creador del software *Mathematica*, ha trabajado extensivamente sobre autómatas celulares en una sola dimensión, y ha hallado propiedades sorprendentes de estos (ver el capítulo correspondiente a los autómatas en una dimensión de Wolfram).

duda. La Naturaleza, sin embargo, es mucho más complicada que el juego matemático de poner células en una malla y ver qué pasa en las siguientes generaciones, porque no tenemos todas las reglas del juego en el universo real. Sin embargo, la idea de Conway, muy exitosa en términos de lo simple de sus reglas y de la naturaleza no determinística del resultado, nos permite estudiar y entender patrones y comportamientos más complejos.

¿Qué tan complejo puede ser el juego de la vida? Se sabe que el juego de Conway es “Turing completo”, lo que significa que pueden implementarse las compuertas lógicas AND y NOT, así como un sistema de almacenamiento de memoria. Con estos elementos es posible entonces construir una máquina de Turing universal. Sin embargo, una cosa es decir esto y otra hacer la construcción directamente en el juego de la vida.

Paul Chapman en el 2002 logró construir una máquina de Turing universal¹³ y por ende, también resulta posible crear un constructor universal. Parece ser que el mismo Conway demostró que esto es posible. En el 2004, Chapman junto con Dave Green presentaron –asómbrense– un constructor universal programable¹⁴. Así pues, si un sistema tan simple de reglas puede ser Turing completo, quizás las leyes de la física lo son, y podrían considerarse Turing computables, asunto que se conoce como al *tesis fuerte Church-Turing*¹⁵. Si esto es cierto, en principio al menos, el juego de la vida podría tener el suficiente poder para simular un universo como en el que vivimos¹⁶.

¹³ <http://www.igblan.free-online.co.uk/igblan/ca/>

¹⁴ Véase http://groups.google.com/group/comp.theory.cell-automata/browse_thread/thread/c62c88b336a917ca/d26a604b1081e460?pli=1

¹⁵ La tesis de Church-Turing formula la equivalencia entre una función computable y una máquina de Turing, que expresado en términos simples “todo algoritmo es equivalente a una máquina de Turing”. Cabe decir que esto no es un teorema de las matemáticas, es una afirmación formalmente indemostrable que curiosamente ha sido aceptada en lo general. (http://es.wikipedia.org/wiki/Tesis_de_Church-Turing).

¹⁶ Esta interesante conclusión es de una comunicación privada sostenida con el Dr. Juan Claudio Toledo, del Instituto de Astronomía de la UNAM. Un buen ex-estudiante mío pero mejor amigo.

Las consecuencias de este juego son fascinantes. Richard Dawkins¹⁷, biólogo evolucionista ha experimentado con modelos de selección artificial en un programa de computadora. Los resultados de esto aparecieron en su libro “El Relojero Ciego”¹⁸, en el que su tesis principal es que la existencia de Dios para entender la vida puede no ser necesaria, contradiciendo quizás la idea de que “si hay un reloj, debería existir un relojero que armó dicho aparato”. Aparentemente no tiene que ser así. Bastan reglas incluso simplistas para poder generar comportamientos complejos.

Esto va en contra del argumento que esgrimía Niels Bohr sobre la vida, en donde hablaba de un fluido vital, el cual estaba en la esencia de los seres vivos y que desde luego, lo inanimado no tenía. Sin embargo, las ideas de comportamientos complejos, emergentes, que son auto-organizados y que se pueden auto replicar, sepultó la idea de Bohr.

Evidentemente los autómatas celulares hablan de auto organización y por tanto, son parte fundamental de los seres vivos. Por lo tanto, su estudio tiene relación directa con los procesos que denominamos inteligentes. Este tipo de entes abstractos nos dan a entender que en última instancia un conjunto de instrucciones que se siguen ciegamente pueden dar origen a la vida, particularmente cuando hablamos en biología. La pregunta sería si podemos partir de ahí para crear vida que sea consciente de sí misma.

¿Qué demostraría el hecho de que reglas simples generan comportamientos complejos? ¿Qué conclusiones podrían sacarse de esto? ¿Estaría totalmente

¹⁷ Richard Dawkins (26/03/1941) Científico británico. Es autor de muchísimas obras de divulgación científica como “El Gen Egoísta” (1976), “El Fenotipo Extendido” (1982), en donde afirma que los efectos fenotípicos no están limitados al cuerpo de un organismo, sino que pueden extenderse incluso al entorno y a otros organismos.

¹⁸ **The Blind Watchmaker**; Dawkins, Richard (1996) [1986]; New York: W. W. Norton & Company, Inc. Hay una versión gratuita del libro de Dawkins en formato PDF en <http://uath.org/download/literature/Richard.Dawkins.The.Blind.Watchmaker.pdf>. Cabe señalar que no es versión pirata. Se han otorgado permisos para poder ser descargada por cualquiera interesado en estos temas.

muerta la idea de Bohr sobre un fluido vital que es parte esencial de los seres vivos? Cabe decir que este fluido vital tiene una especie de esencia mística, como si estuviese más allá de las explicaciones de la ciencia. Si el gran Bohr pensaba así no es casualidad, simplemente habla de las dificultades para defender y entender el concepto de vida.

Por otra parte, es un hecho ya demostrado que los experimentos con autómatas celulares, ya sea con las ideas de von Neumann o de Conway, pueden generar comportamientos emergentes de auto replicación y auto organización, dos temas asociados completamente con el tema de la vida biológica. Si estas reglas ciegas pueden generar esta complejidad, la idea del fluido vital desaparece aparentemente, pero ¿no estaremos eliminando la idea de Bohr demasiado pronto? Porque sí, las reglas ciegas parecen contener finalmente en el autómata celular, las condiciones para la auto-reproducción y auto-organización pero... ¿por qué se produce esto? ¿Qué mecanismo es el que genera la auto-organización, por ejemplo?

Capítulo IV

La nueva ciencia de Stephen Wolfram

¿Piensas que soy un autómatas?

¿Una máquina sin sentimientos?

Charlotte Brontë (*en Jane Eyre*)

Los autómatas celulares de von Neumann y de Conway sin duda hicieron despertar la imaginación de otros investigadores. Una idea sensata es quitar una dimensión a los autómatas bidimensionales, como los del juego de la vida, y ver qué pasa si ponemos a trabajar a los autómatas en una sola dimensión. Esto, en principio, podrías sugerir una simplicidad mayor que al trabajar con autómatas como en el juego de la vida, que ocurren en dos dimensiones. Sin embargo, esta simplificación es aparente, pues en una dimensión, los autómatas hallan desarrollos emergentes complejos.

Consideremos una línea recta de células, de casillas, las cuales pueden estar ocupadas o vacías. Pueden pues tener células o no tenerlas. Aquí, las reglas del autómatas, también ciegas, se basan simplemente en la vecindad de cada célula con la que se está trabajando en cada generación. De acuerdo a las reglas que se definen, podremos ver lo que pasa en la línea de células, generación tras generación. Además, para darnos una mejor idea, podemos colocar la primera generación (la configuración inicial), y ver el comportamiento, pintando la segunda generación debajo de la primera. Podemos hacer esto para cada ciclo t en el tiempo y en un momento determinado tendremos decenas, centenas o miles de

generaciones desplegadas. Con ello podremos hacer un análisis de lo que está pasando.

Formalmente podríamos decir que los autómatas, las células, tienen valor 1 si están en alguna casilla de la línea de cuadros disponibles o bien contienen un valor 0 (cero), es decir, no hay nada en una casilla. Las reglas de evolución se presentan solamente analizando la vecindad de cada casilla de interés, una por una.

Uno de los primeros científicos en estudiar cuidadosamente este tipo de autómatas fue Stephen Wolfram. Hijo de refugiados judíos, se le consideró como niño prodigio. Estudió en el *Eton College* después de que se le otorgara una beca por publicar su primer artículo sobre física de partículas, a los 16 años. Un año después se pudo matricular en la Universidad de Oxford. Consiguió su doctorado a la edad de 20 años, en física de partículas, por parte de *CalTech*.

Wolfram es más conocido por *Mathematica*, un programa de computadora que puede hacer matemáticas simbólicas. Sin embargo, mucho de su trabajo lo ha enfocado al comportamiento de los autómatas celulares. En 1983 se trasladó a Princeton e hizo muchísimas simulaciones de computadora para entender el comportamiento de los autómatas celulares en una dimensión.

De acuerdo principalmente al trabajo de Wolfram, existen diferentes clases de autómatas celulares, los cuales se han clasificado de acuerdo a su comportamiento. Hay cuatro grandes clases de acuerdo a su nivel de predicción en su evolución:

- **Clase 1.** Son aquellos que su evolución está literalmente predeterminada (probabilidad 1), independientemente de su estado inicial o configuración.

- **Clase 2.** Son aquellos en los que el valor de un sitio en particular, a muchos pasos del tiempo, está determinado por los valores iniciales en algunos sitios en una región limitada.
- **Clase 3.** El valor de un sitio en particular depende de los valores de un número siempre creciente de sitios iniciales. Esto quiere decir, en términos coloquiales, que conducen a comportamiento caótico.
- **Clase 4.** Son aquellos en los que en un sitio en particular puede depender de muchos valores en la configuración inicial y en los que la evolución del autómata simplemente no se puede predecir. Vamos, hay que hacer la simulación explícita del mismo.

Esta definición tan formal puede confundirnos pero como veremos, el entender cómo evolucionan estos autómatas es verdaderamente sencillo.

Wolfram habla de *reglas locales de evolución*, que no son otra cosa que las reglas ciegas que descubrió Conway en el juego de la vida o las reglas en la hormiga de Langton, como veremos más adelante. Estas reglas son las que definen el comportamiento en la evolución del autómata. Por ejemplo, consideremos una línea de células y las siguientes reglas locales:

$$\begin{array}{c} \downarrow \downarrow \\ \begin{array}{l} 000 \rightarrow 0 \\ 001 \rightarrow 1 \\ 010 \rightarrow 0 \\ 011 \rightarrow 1 \\ 100 \rightarrow 1 \\ 101 \rightarrow 0 \\ 110 \rightarrow 1 \\ 111 \rightarrow 0 \end{array} \end{array}$$

Reglas locales de un autómata en una dimensión

Por ejemplo, consideremos una línea que contenga dos células en el medio de la misma, es decir:

000...001100...000

La evolución de la siguiente generación pide revisar cada célula en la línea. Cuando tenemos 000...000... no hay cambio alguno, pero llega un momento en que tenemos 001 (recuérdese, sólo hay dos células en la línea del autómata). Hallamos que tenemos que calcular el resultado cuando encontramos 001. Eso, de acuerdo a la tabla da como resultado un 1. Se pone en la siguiente línea, debajo de la célula que estamos analizando. Ahora movemos nuestro apuntador a la siguiente célula y hallamos la configuración 011, la cual da como resultado un 1. Pasamos a la siguiente célula y encontramos la configuración 110, lo cual vuelve a darnos un 1. Seguimos con este procedimiento y hallamos los valores 100, lo cual de nuevo, da un 1. Inmediatamente después todos los valores dan 0. El autómata quedará entonces así en la segunda generación (la primera generación es la inicial):

000...001100...000 [Generación inicial]

000...011110...000 [Segunda generación]

Este sencillo procedimiento se puede programar en una computadora y así no tener que hacerlo “a mano”. En los apéndices puede hallarse un programa en Pascal para realizar esta simulación.

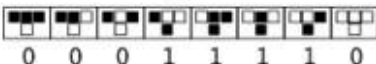
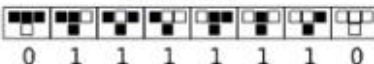
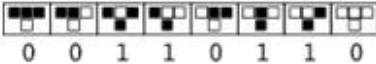
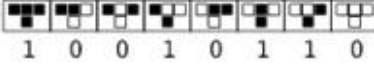
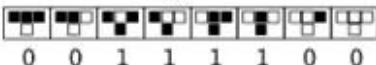
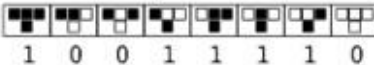
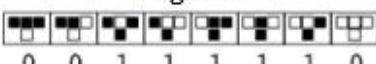
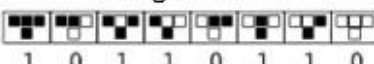
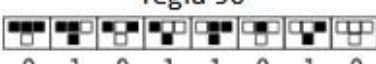
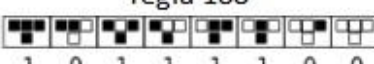
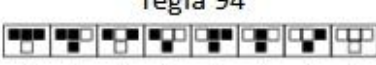
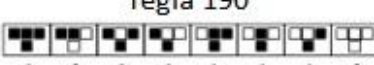
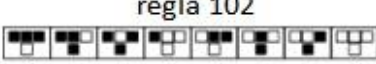
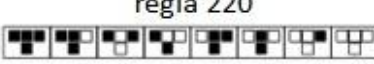



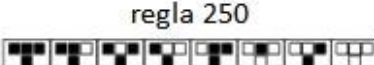
Wolfram define además las reglas “legales”¹⁹, las cuales tienen las siguientes características:

¹⁹ Evidentemente decirle a las reglas “legales” o “ilegales” no tienen esta interpretación en términos cotidianos. Es simplemente una manera de definir cuáles reglas son a las que les voy a dedicar las simulaciones.

- La configuración: 0000000...00000... se puede considerar como un estado nulo (o estado base), y esto no cambia con el tiempo. A esto los físicos y matemáticos suelen decir que es “invariante” en el tiempo.
- Las reglas locales deben ser simétricas, es decir, 001 debe dar el mismo resultado que 100, por ejemplo.

Wolfram analiza estos autómatas inicialmente considerando la vecindad de cada célula a su casilla inmediata, a la izquierda o a la derecha. No hay razón para no pensar en considerar una vecindad más amplia, pero evidentemente, el tomar la vecindad más pequeña hace el análisis más simple, al menos en un principio.

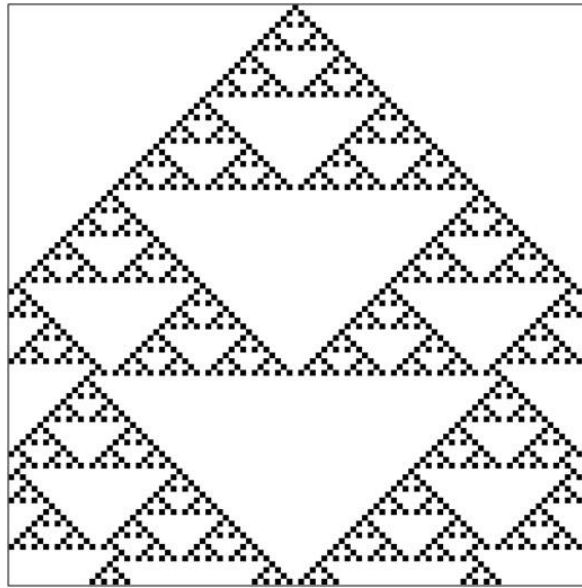
He aquí una serie de reglas locales, que Wolfram ha estudiado:

regla 30 	regla 126 
regla 54 	regla 150 
regla 60 	regla 158 
regla 62 	regla 182 
regla 90 	regla 188 
regla 94 	regla 190 
regla 102 	regla 220 
regla 110 	regla 222 
regla 122 	regla 250 

Diferentes reglas locales

Veamos un ejemplo completo. Tomemos las reglas locales ya descritas anteriormente. Esta es la regla 90 (binario **01011010**).²⁰ Si consideramos el conjunto de reglas locales, solamente tenemos 256 posibilidades, de 0 a 255. Considerando que en la primera generación, la inicial, ponemos una sola célula en medio de la línea completa, el desarrollo (que se ve cada generación una línea hacia abajo), encontraremos lo siguiente:

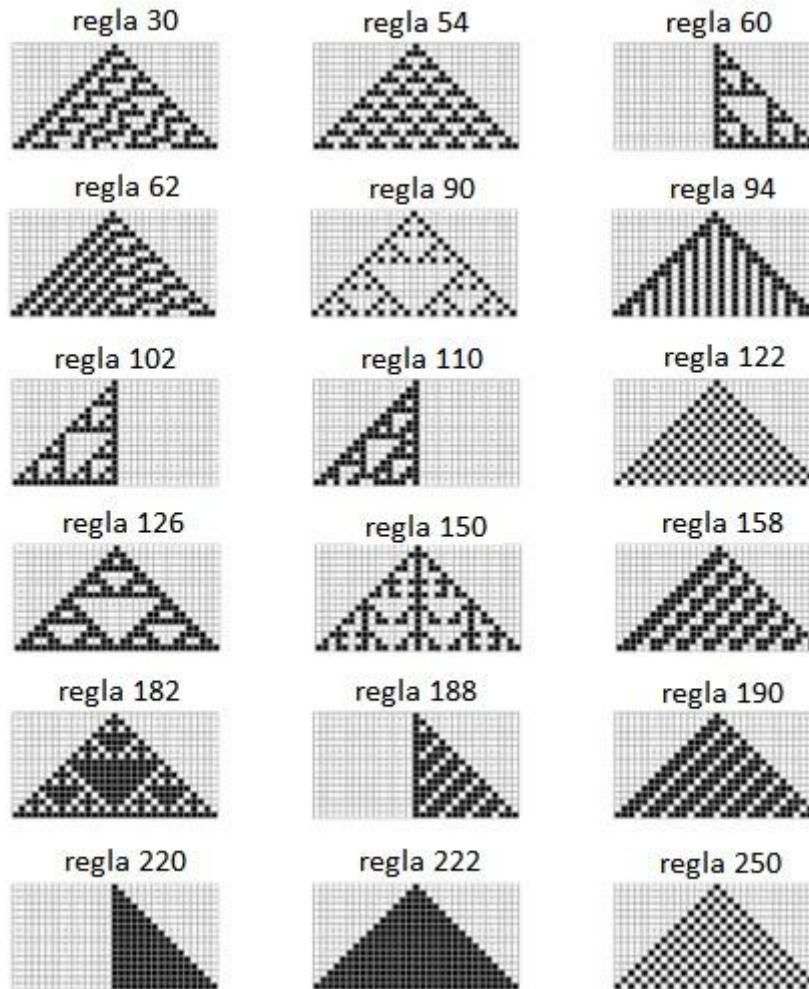
²⁰ El número de regla es simplemente los resultados de las reglas locales leídas de arriba abajo formando un número binario. En el caso del ejemplo, 01011010 binario es en decimal 90.



Regla 90 del autómata celular de una dimensión de Wolfram

Puede verse que la imagen generada es literalmente un *fractal*, es decir, un objeto cuya estructura básica, fragmentada (o irregular), se repite a diferentes escalas. Los árboles, por ejemplo, son fractales. Sus ramas se van adelgazando en más ramas pequeñas, etcétera.

Otras configuraciones interesantes pueden verse en la siguiente figura:



Otras reglas locales y sus resultados

La cuestión aquí es entonces saber qué resultados nos proporcionan los autómatas celulares en una sola dimensión. ¿Qué nos dicen?

Debemos comprender que estos autómatas son sistemas dinámicos y pudiesen ayudarnos para modelar la creación de las formas biológicas. De hecho, hay ejemplos notables de formas –en la biología– que pueden modelarse de esta manera. Algunos dibujos en la piel de ciertas serpientes, las manchas de algunos felinos o bien, la pigmentación de una concha de caracol, puede mostrar cómo estas reglas locales de un autómata celular en una dimensión, en donde

solamente dichas reglas tienen influencia en una vecindad de cada célula, modelan muy bien estos eventos biológicos.

Pero más allá de los fenómenos de la vida, los autómatas pueden ser útiles incluso para intentar dar más luz sobre teoremas y conjeturas no resueltas en matemáticas.²¹ Eso habla probablemente de su universalidad.

²¹ Véase el apéndice sobre la conjetura de Collatz y los autómatas celulares en una dimensión.

Capítulo V

Los autómatas primitivos de Heiserman

Ahora considérese el término ‘automatón’. Esta palabra tiene la misma raíz que ‘automático’, y lo que es aún más significativo es el hecho de que comparte una herencia común con la palabra ‘autónomo’ – y esta es la palabra clave que define un robot.

David L. Heiserman²²

Uno de los primeros intentos de estudiar vida artificial nace probablemente cuando la computadora personal se vuelve accesible a las mayorías. Las microcomputadoras de 8 bits, Apple II, Radio Shack, Commodore 64, se pusieron a precios relativamente accesibles y de pronto mucha gente tuvo la oportunidad de hacer cómputo en su propia casa.

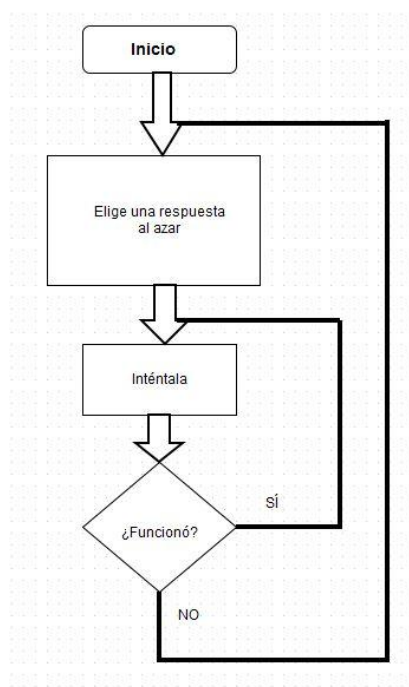
Heiserman, que desde niño tuvo inclinación por armar robots, ha escrito más de 30 libros sobre diversas tecnologías, en donde la programación de computadoras y la construcción de sistemas auto-programables han sido temas recurrentes. En su libro “Projects in Machine Intelligence for your Home Computer”²³, el autor decide incursionar en la vida artificial. A partir de programas escritos para la computadora Radio Shack y Apple II, define una serie de creaturas, que no son

²² http://www.beam-wiki.org/wiki/Heiserman,_David_L.

²³ TAB Books, 1982,

otra cosa que puntos en la pantalla gráfica de la computadora, los cuales se mueven de acuerdo a reglas específicas ciegas, es decir, no hay criterios diferentes a los que están pre-programados por Heiserman.

El autor define dos tipos de creaturas. Las primeras las llamó *clase alpha*, las cuales tienen un comportamiento como puede verse en la siguiente figura:



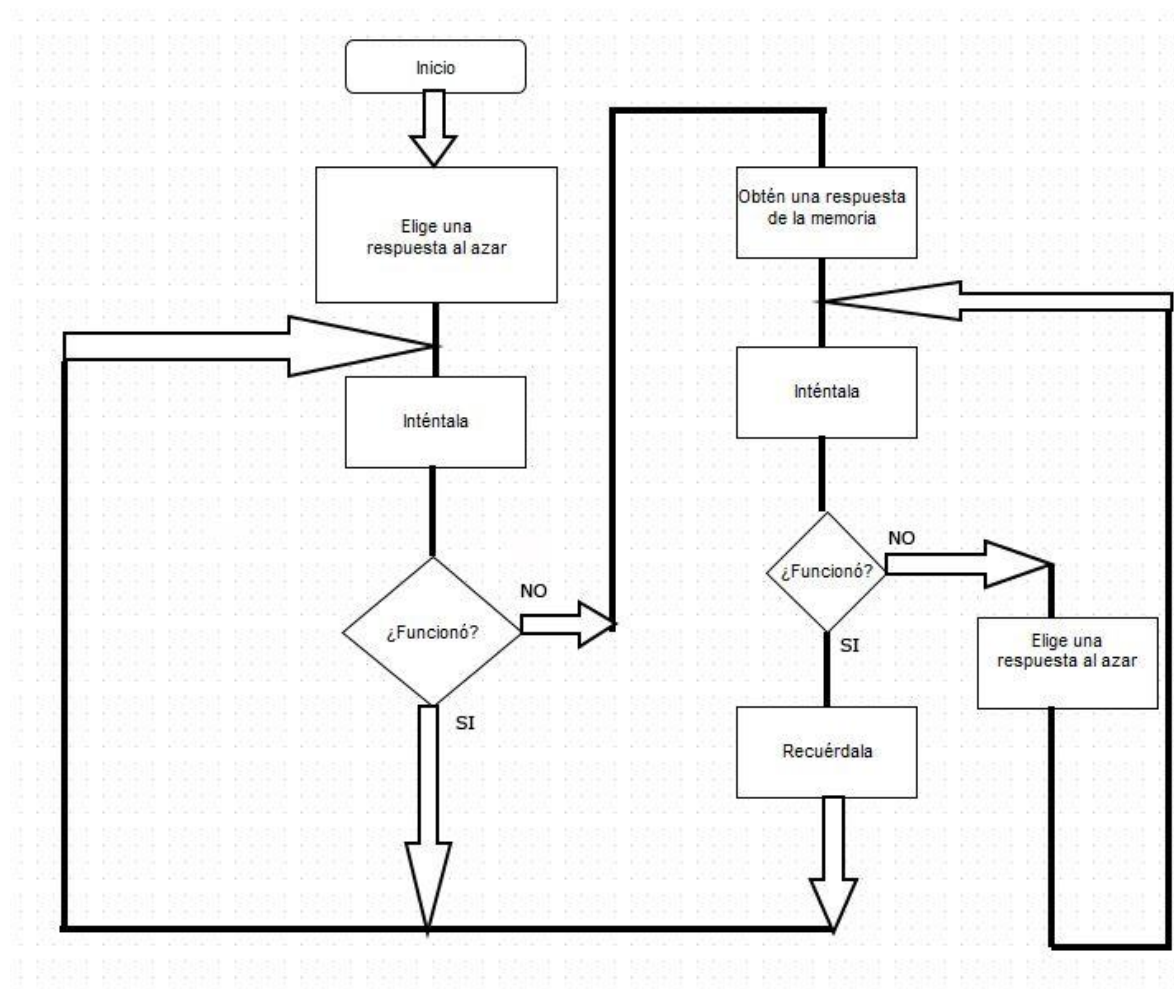
Comportamiento de las creaturas clase alpha, de Heiserman

Este tipo de creaturas se comporta de manera muy elemental. Inicia su vida seleccionando una respuesta al azar, es decir, hacia dónde puede moverse. En principio no hay manera de saber hacia dónde se moverá, pues el paso que dé es completamente aleatorio. Su respuesta se deja al azar²⁴.

²⁴ En términos estrictos, la computadora no puede generar una secuencia absolutamente al azar. Utiliza de hecho un algoritmo que genera una secuencia de números pseudo-aleatorios, los cuales podrían conocerse si fuese necesario. Sin embargo, para los fines prácticos del movimiento de la creatura alpha, el movimiento parece ser, a simple vista, aleatorio.

Una de las características que hacen de las creaturas alpha poco inteligentes es que no tienen memoria del pasado. Simplemente buscan una respuesta y si ésta puede ejecutarse, lo hacen. En caso contrario busca una nueva acción a ejecutar. Podría pasar que de pronto la creatura alpha no encontrara una respuesta a ejecutar y entonces el sistema se cicla infinitamente.

Sin embargo, Heiserman define un segundo tipo de creaturas, llamadas clase *beta*, la cual tiene memoria de encuentros pasados con su medio ambiente. Su diagrama de flujo puede describirse de esta manera:



Comportamiento de las creaturas clase beta

Cabe notar que las criaturas beta empiezan su vida con una selección igualmente aleatoria que en el caso de las alpha. Intenta ejecutar la acción y guarda en memoria si la selección funcionó o no. Si funciona, la criatura sigue haciendo lo que la selección azarosa inicial le dice que debe hacer. La diferencia con respecto a las criaturas alpha es que cuando la selección deja de funcionar, entonces la criatura beta busca en su memoria sobre situaciones pasadas similares y la acción tomada en esos momentos.

Una criatura beta nace con la memoria nueva, sin recuerdos de nada, la cual se llenará en la medida que el proceso se ejecuta. Si la criatura no encuentra ninguna referencia al pasado, entonces elige de nuevo una selección azarosa e intenta ejecutarla. En todos los casos nos referimos a criaturas que simplemente se mueven en la pantalla. Esas son las selecciones que el sistema hace al azar.

Cuando se ejecutan estos programas (los cuales Heiserman describe en Applesoft Basic y Radio Shack Basic), lo que se ve es un punto en la pantalla que se mueve de acuerdo a las selecciones que se generan al azar. ¿Qué podríamos decir de estos comportamientos? ¿Podríamos asignarle inteligencia acaso? ¿O voluntad? El punto es investigar hasta donde hay un comportamiento autónomo que además, se vuelva complejo, a partir de reglas tan sencillas.

Heiserman escribe una serie de programas en las dos modalidades de BASIC (cuyo código fuente está completo en su libro para ser transcrito) y pone a sus criaturas a moverse en el limitado espacio que le da la pantalla gráfica. El comportamiento de las criaturas alpha y beta es similar, pero pareciese que las criaturas beta son más inteligentes.

Sin embargo, Heiserman entonces decide investigar este punto y pone a criaturas alpha y beta juntas, en la misma arena. ¿Qué pasará? ¿Cómo competirán entre ellas? El autor propone un mecanismo muy simple para el comportamiento de sus criaturas. En el caso de las alpha, tienen 50% de decidir si eliminan a su

competidor (cuando choca con él) o bien no hace nada. Para las creaturas beta, el comportamiento es similar, cuando éste ocurre por primera vez, pero en este caso, las creaturas beta van formando una memoria de los hechos pasados y del cómo ha actuado en ese momento. Ese comportamiento se repite cuando ya existe el antecedente. Podemos decir que las creaturas beta tienen memoria del pasado mientras que las alpha no se acuerdan jamás de lo que pasó anteriormente.

El experimento entonces consiste en poner muchas creaturas alpha y beta en la misma arena y correr el programa hasta que quede un solo ganador. Heiserman halla que las creaturas alpha tienden a ganar más veces que las creaturas beta. ¿Por qué se da este hecho? ¿No esperaríamos que las creaturas con memoria sacaran provecho de su experiencia pasada? El autor concluye que eso era de esperarse, pero el mecanismo inicial para que una creatura beta decidiera qué acción tomar al chocar contra una alpha, se decide en un 50% de posibilidades a favor de eliminar al antagonista o bien, dejarlo vivir. En ese caso, la primera elección azarosa de alguna manera condena el futuro de esa creatura beta.

Por su parte, las creaturas alpha siempre tienen el 50% de posibilidades de eliminar a su adversario o dejarlo vivir. Bajo ese criterio, es una cuestión de simple suerte, de simple azar, que una creatura alpha sobreviva por mucho tiempo e incluso emerja como la ganadora en la arena. Lo que está claro es que las creaturas alpha tienden a ganar más veces en este juego y eso podría demostrar que la memoria de hechos pasados para repetir acciones conocidas no necesariamente es lo adecuado, porque las decisiones tomadas originalmente quizás están mal de raíz.

Heiserman fue pionero en el tema de la vida artificial. Sin embargo, no va más allá. No propone un factor de reproducción y replicación y por ende, sus autómatas no son más que un interesante juego al principio, que se torna bastante aburrido con

el tiempo. Sin embargo, sus ideas son, sin duda, un punto de partida para el estudio de la vida artificial²⁵.

²⁵ En <http://atariage.com/forums/topic/208936-musings-in-machine-intelligence/> el usuario “Stargunner” reescribió algunos de los programas del libro de Heiserman para correrlos en una –ya obsoleta- computadora Texas Instruments (TI-99). Y aunque es probable que el lector no tenga semejante máquina, puede usar un emulador de la misma (un programa que se comporta igual que si se tuviese la computadora real). Hay un sinfín de emuladores para todo tipo de máquinas. Consúltese este sitio: <http://www.99er.net/emul.html>.

Capítulo VI

Virus informáticos y vida artificial

Pienso que los virus de computadora deberían contar como si fuesen vida. Considero que dicen algo sobre la naturaleza humana, que la única forma de vida que hemos creado hasta ahora es puramente destructiva. Hemos creado vida a nuestra imagen y semejanza.

Stephen Hawkins (*The Daily News*, Agosto 4, 1994)

En 1981 apareció el primer virus de computadora y la historia de este fenómeno ha sido muy importante porque se ha convertido en toda una industria y además, los mecanismos para “infectar” sistemas de cómputo tienen interés por sí mismo. Y aunque hoy día el problema de los virus informáticos podríamos calificarlo de menor, pues ya se conocen todos los posibles mecanismos de infección, no deja de ser interesante verlo a través de la óptica de la vida artificial. De alguna manera, finalmente, un virus informático es un organismo que se auto-replica. Y aunque “vivan” dentro de la computadora, esto no elimina la posibilidad de clasificarlos como una forma de vida.

Las computadoras se han diseñado para ejecutar instrucciones de manera secuencial, una por una. Estas instrucciones (que pueden ser en gran número), suelen hacer cálculos, desplegar información en la pantalla del monitor, hacer gráficas, etcétera. Sin embargo, se pueden escribir programas que ejecuten instrucciones con aviesas intenciones. Por ejemplo, se puede hacer un programa

que dé formato al disco duro automáticamente, eliminando toda la información que contenga éste. Puede ocurrir que en algunos casos los programas por error o accidente, cometan acciones que dañan nuestros datos o que haga que el programa se comporte de manera errática. A eso llamamos en última instancia un error, un *bug*, el cual se reporta al desarrollador o la empresa de software para que ésta, de alguna manera, elimine el inconveniente y nos provea de una versión funcional.

Cuando un programador decide que su software cause algún problema en la computadora, hablamos de *malware*, de programas dañinos en nuestra máquina. Hay muchas maneras en que estos programas puedan ser ejecutados, normalmente sin el consentimiento del usuario. A esto se le ha dado en llamar virus informático por la analogía con los virus biológicos.

Cabe decir que la palabra “virus” hace referencia a lo *venenoso*. En términos biológicos, una infección viral se pone en marcha por un virus inyectando su información genética a las células del individuo que va a ser infectado. Al infectar las células, éstas producen réplicas del virus y el individuo entonces enferma cuando el número de estas réplicas sobrepasa la capacidad inmunológica del cuerpo. Así pues, un virus de computadora es un programa en código nativo, en código de máquina, que se copia a sí mismo (o bien hace una copia modificada de sí mismo), en los programas ejecutables de la computadora. Cuando estos programas infectados son ejecutados, el código viral se ejecuta y el virus se replica más veces, en ocasiones infectando a más programas o bien a través de mecanismos de transmisión de datos, por ejemplo, el correo electrónico.

La capacidad de un virus informático para infectar puede ser enorme y hacer su labor de reproducción en el sector de arranque del disco duro (que implicaría nuevas potenciales infecciones cada vez que se enciende la computadora), o bien, en un manejador de dispositivos (*device driver*) e incluso, la consola de intérpretes

de comandos. Las formas de replicación de un virus informático son variadas y muchas veces, muy ingeniosas.

Los virus de computadora solamente afectan a los programas ejecutables. Los datos no son infectados normalmente porque los datos no se ejecutan como un programa y por lo tanto, no es posible hacer replications de código cuando no existe el mecanismo para hacer la reproducción. En términos de Von Neumann, los datos no tienen nada parecido a la unidad constructora y a la unidad de supervisión. Por ello, los datos no son susceptibles de ser infectados aunque bien podrían ser dañados, borrados o cambiados, por un virus informático en particular.

Fred Cohen²⁶, un connotado especialista en virus computacionales, así como otros, han intentado formalizar la definición de virus informático. Sin embargo no parece estar claro qué es y qué no es exactamente un virus. Cohen indica que *un virus de computadora es un programa capaz de auto-reproducción*. Desafortunadamente esta definición alcanza a los compiladores, que son traductores de un código que pueden entender los humanos a un código nativo de la computadora. Esto llevó al propio Cohen a decir que estos últimos serían “virus buenos”.

John Inglis²⁷ define los virus informáticos de la siguiente manera, la cual ha sido ampliamente aceptada:

Se define un virus informático como un fragmento de código que tiene dos características:

- *Tiene una capacidad automática (total o parcial) para reproducirse*
- *Usa un método para transferir lo que depende de su habilidad para duplicarse a sí mismo y a otras entidades de un sistema (programas, sectores en un*

²⁶ Véase el apéndice sobre el experimento de Cohen sobre virus

²⁷ <http://sjsu.rudyrucker.com/~shruti.parihar/paper/>

disco, sector de arranque, archivos de datos, etcétera), que se mueven entre estos sistemas.

Los virus informáticos llamaron la atención cuando empezaron a convertirse en un problema entre los usuarios de computadoras. El entorno del cómputo personal era inseguro, pues su diseño no pretendía que el usuario compartiera con otros datos o información fuera de la propia máquina. Pero este enfoque de seguridad débil llevó a terceros a empezar a escribir programas que se auto replicaban en archivos importantes, por ejemplo, aquellos que hacían que una computadora personal arrancara el sistema operativo de disco flexible (DOS), logrando de esta manera infectar otros programas, los cuales en un momento dado, al ser ejecutados, infectaban otros discos de sistema, haciendo perder información, alentar procesos o dejar inútil a la máquina misma.

Los virus informáticos vieron su mejor época, si podemos llamarla así, en los sistemas basados en MsDOS, de Microsoft, pero en esencia cualquier equipo de cómputo podría ser infectado por mecanismos parecidos. Es interesante señalar que los sistemas basados en Unix (Mac OS X o la mayoría de las distribuciones de Linux), no tienen problemas de virus porque ningún código que venga de donde venga, puede modificar un programa sin tener los permisos del administrador o del súper usuario. Llama la atención que en la plataforma Windows por años no se haya podido resolver este problema de los virus de manera definitiva.

El antecedente de los virus son los gusanos (*worms*), pero aunque se consideran igualmente virus en muchos casos, no lo son, pues esos programas no modifican a otros programas existentes. Uno de los más famosos gusanos fue creado por Robert Tappan Morris, en 1988, el cual con el tiempo se conoció como el “gusano Morris”. Este programa afectó a aproximadamente el 10% de todas las computadoras conectadas a Internet de ese entonces (llamado Arpanet), en mayor o menor grado por este programa.

Hoy se sabe que el programa buscaba las contraseñas de otras computadoras utilizando las más comunes. En cierto sentido, estamos hablando de ingeniería social, lo que significa que no se trata de decodificar contraseñas encriptadas, sino utilizar un criterio más social: ¿qué clase de contraseña usan las personas comúnmente? Aparte de esto, el gusano de Morris sacaba ventaja de un error de seguridad de los sistemas Unix (versión Universidad de Berkeley) y a través de una debilidad en el sistema llamado *Sendmail*, el gusano se replicaba de máquina en máquina. Se estima que unas 6000 máquinas fueron afectadas ese año. Internet/Arpanet tenía conectadas en el mundo en ese momento unas 60,000 computadoras a la red.

El autor del software, Robert Tappan Morris, de 23 años en ese entonces, simplemente creó un programa que se auto replicaba pero jamás pensó que este proceso se llevaría a cabo tan rápido de computadora en computadora. Se sabe ahora que el gusano de Morris fue en realidad producto de dos programadores. Morris hijo se inspiró en el código de su padre, que trabajaba en los Laboratorios Bell y que había diseñado un juego en los años sesenta en donde se trataba de crear un programa que al reproducirse ocupara toda la memoria, borrando con ello de la memoria al oponente. Esto era el programa llamado *Darwin*, del cual hablaremos en un capítulo posterior.

Morris hijo fue descubierto y sentenciado en 1990 por un jurado federal, el cual le dio como sentencia 400 horas de trabajo comunitario, tres años de libertad condicional y una multa de 10,050 dólares.

La estructura y operación de un virus es la siguiente. Por una parte, los virus informáticos tienen dos componentes, el que hace que la infección se propague y el que hace el trabajo de manipulación. Este componente podría no estar presente o no tener efecto alguno, o bien esperar a que ocurra algún evento o circunstancia para que se dispare.

Para que un virus de computadora funcione, se requiere que exista en él código ejecutable. El código viral se ejecuta usualmente antes de que el código infecte a su anfitrión. Una manera de clasificar a los virus informáticos es pues, basarse en los modos en como el virus puede añadirse al código del programa anfitrión: como un *shell*, como un añadido, como código intrusivo o como un virus de compañía, aunque en este último caso hablamos más bien de un troyano, un programa que contiene código malicioso que se ejecuta como un programa normal.

Los virus de *shell* significan que se escriben pensando en que el código malicioso esté envuelto en una especie de protección, una concha, por ejemplo, en donde el programa original infectado se convierte en una subrutina del virus mismo. En cambio, los virus añadidos son el esquema más tradicional de lo que es un virus de computadora y que altera la información en el código que se ejecuta desde un inicio. El programa anfitrión de hecho casi ni se toca y la única manera de sospechar que algo anda mal es que los programas de pronto han crecido en tamaño.

Los virus intrusivos operan sobre escribiendo una parte del código del programa anfitrión con el código virulento. El reemplazo puede ser selectivo en muchos casos. Hay pocos virus intrusivos, a todo esto.

Desde luego que se podría establecer toda una clasificación relacionada a los virus en la forma en cómo se activan o bien, en la manera en que buscan nuevas alternativas para infectar otros programas. Lo interesante es finalmente es que el modelo virus informático/computadora, es una analogía bastante exacta en donde los mecanismos de infección funcionan formalmente de manera idéntica con los virus biológicos (aunque evidentemente los mecanismos de infección son otros).

Es interesante ver cómo los virus informáticos, además, han evolucionado con el tiempo. Parecieran seguir -curiosamente- como las infecciones biológicas que de alguna manera se vuelven resistentes a los medicamentos. En ese sentido, se

reconocen cinco “generaciones” de virus, en donde cada nueva clase es más difícil de detectar y borrar. He aquí una breve clasificación:

- *Primera generación: virus simples.* Los cuales solamente buscaban replicarse. El daño a veces se producía por accidente y no porque el programador hubiese deseado dañar los datos de un tercero. Esta primera generación de virus no se escondía en el sistema.
- *Segunda generación: virus que se auto reconocen.* Un problema con los mecanismos de infección es que algunos virus no saben detenerse en su misión de infectar. Vamos, que no hay ningún mecanismo que pueda apelarse en el virus mismo que detenga la operación de infección una vez que se ha hallado ya que la infección se ha producido. Para prevenir un infección reiterada en un mismo archivo, los virus de la segundas generación usualmente implantaban una firma única que indicaba que el virus estaba ya infectado.
- *Tercera generación: virus disimulados (stealth).* Muchos virus se podían identificar por la búsqueda de un patrón de bytes que los identificara, su firma característica. Para evitar esto, algunos virus se disimulaban o se escondían mandando información falsa cuando se buscaba detectarlos. Por ejemplo, si una operación de lectura (del disco), se ejecutaba y ésta incidía sobre los sectores que el virus usaba, éste mostraba la información de un sistema sin virus, la cual era falsa. Esta idea buscaba pues disimularse en el entorno de la máquina para ser indetectable.
- *Cuarta generación: virus blindados.* Los cuales funcionan añadiendo código inútil o innecesario para hacer más difícil el análisis del mismo. Este tipo de código tiende a ser más largo que el de los virus simples y en alguna medida son más fáciles de detectar.

- *Quinta generación: virus polimórficos.* Como su nombre lo indica, este tipo de virus mutan continuamente. Por ejemplo, pueden infectar programas con código encriptado de sí mismos. Variando el código y las secuencias de bytes, un antivirus con una firma del virus definida, no tendría éxito en su búsqueda. De hecho, estos son los virus más difíciles de detectar.

Si hemos hablado de esta clasificación es para que, de alguna manera, entendamos que los mecanismos que los creadores de virus han programado para evitar ser detectados son parecidos a una serie de acciones que los propios virus biológicos toman para no poder ser eliminados. En este sentido, el modelo de los virus informáticos como vida artificial es notable. La pregunta es si efectivamente estos virus de computadora califican como vida artificial.

El problema termina siempre siendo el mismo: ¿Cómo caracterizar la vida? Pareciese que no hay una definición concisa y que en realidad al hablar de vida estamos hablando de una serie de parámetros que la definen. En el caso que nos ocupa, podríamos decir que:

- La vida (de un virus informático) es un patrón específico en un entorno de la computadora
- La reproducción es condición necesaria en los organismos
- El almacenamiento de información tiene una auto-representación
- Hay una interacción funcional con el entorno
- Hay estabilidad a pesar de posibles perturbaciones del entorno
- Tienen capacidad de evolucionar
- Pueden crecer y expandirse

En pocas palabras son sistemas que parecen tener todas las alternativas para ser considerados vivos. La computadora aquí es el entorno donde los virus pueden florecer cuando infectan a los programas anfitriones. Los mecanismos de infección buscan, por lo menos en esencia, el que sobrevivan estos virus una vez que se

han ejecutado, pues eventualmente, al correr un programa infectado, éste buscará infectar a otros. Esto de alguna manera busca garantizar “la supervivencia de la especie”.

Sin embargo, toda esta discusión parece ser en principio formal. Los virus informáticos fuera del anfitrión –los programas de la computadora– simplemente no tienen sentido mencionarlos, pero quizás la misma argumentación se aplica a los virus biológicos y a los organismos que infectan.

Capítulo VII

Auto referencia y recursión

Ciclo recursivo: Véase ciclo recursivo.

Del Índice del Manual de Borland Pascal con objetos 7.0 (1992)

Un elemento que parece condición *sine qua non* para la existencia de vida es la auto replicación de los organismos, lo que viene a ser en términos formales lo que llamamos auto referencia. Los organismos vivos, por ejemplo, poseen en su genética la información completa para crear otro ser vivo como él. Tienen pues un mapa completo de ellos mismos y en alguna medida es auto referente. Curiosamente, en lo que se refiere a las computadoras, que es el entorno en donde hemos analizado los problemas de la vida artificial, hallamos que éstas se programan a partir de lenguajes específicamente diseñados para generar las secuencias de instrucciones que hagan que la máquina haga la labor que queremos.

Los lenguajes de programación vienen en muchos colores y sabores. Los hay de bajo nivel, como los ensambladores, que permiten escribir con códigos mnemotécnicos, las instrucciones para manejar los registros, leer la memoria, guardar información en la misma, hacer operaciones lógicas y aritméticas con

ceros y unos, que es finalmente el lenguaje de más bajo nivel, el más primitivo, que podemos hallar en una computadora. Otros lenguajes de programación abstraen muchos de los detalles del código nativo y permiten programar sin tener que pasar por los detalles de la implementación física de la computadora. Así, podemos poner una instrucción como la siguiente:

$$A := B + C;$$

típica de Pascal, que suma los números $B + C$ y el resultado lo pone en la variable A . Este mismo programa en ensamblador del procesador 6800 (de Motorola, de 8 bits), se escribe de esta manera:

```
LDAA #$05
ADDA #$08
STAA $50
SWI
```

Lo cual dice que la secuencia para sumar dos números sería:

- Cárquese el acumulador A (un registro que tiene el microprocesador 6800) con el valor 5 (en hexadecimal) [LDAA #\$05]
- Añádase al acumulador A el valor 08 [ADDA #\$08]
- Guárdese el resultado de lo que hay en el acumulador A en la localidad 50 (hexadecimal) [STAA \$50]
- Termínesse el programa con una interrupción por software [SWI] (*software interrupt*)

Como puede verse, hay que hacer más operaciones en lenguaje ensamblador que en un lenguaje como Pascal. Por supuesto, Pascal requiere de un programa especial, llamado “compilador”, que es un traductor de código de alto nivel a código de máquina, para que la computadora pueda ejecutar la suma.

Lo importante aquí es que los lenguajes de alto nivel permiten sustraerse de los detalles y así abstraer la información, colocándola en otro nivel y así hacer que la programación sea más fácil de escribir y de seguir por otros. Los compiladores, cabe aclarar, convierten el código fuente (en Pascal, por ejemplo), en código objeto (código nativo de la máquina que lo ejecutará).

La pregunta que naturalmente surge, en el contexto de la vida artificial es: ¿Podemos crear un programa que sea auto referente? Es decir, ¿podemos escribir un programa que entregue como resultado el propio programa fuente? Este tipo de programas se llaman genéricamente “código Quine”, por ser el apellido del filósofo (y lógico), Willard van Orman Quine, que trabajó en la auto referencia en los lenguajes humanos. Por ejemplo, al decir “**esta frase es falsa**”.

La tarea de hacer un programa que se auto replique, se auto reproduzca, tiene mucho que ver con el fenómeno de la vida misma y desde luego, con el formalismo de la vida artificial. Actualmente el crear un programa que se auto reproduzca así mismo es un reto que se ha resuelto prácticamente en todos los lenguajes de programación, desde los más sofisticados hasta los más simples. Veamos algunos ejemplos.

En BASIC:

```
10 READ A$:PRINT 10 A$:PRINT 20 "DATA" A$
20 DATA READ A$:PRINT 10 A$:PRINT 20 "DATA" A$
```

Autor: **Christmas Hartman** (*hartman@symcom.math.uiuc.edu*), escrito para la Commodore 64²⁸

Una descripción somera de la idea de Hartman es la siguiente: En la línea 10 el programa lee una cadena de caracteres (la cual está en la sección DATA del

²⁸ http://www.nyx.net/~gthomps/self_bas.txt

código (línea 20). Nótese que la línea 20 es prácticamente la repetición del código de la línea 10, que se maneja no como código, sino como datos del programa, que son leídos por el código de la propia línea 10.

En Pascal, un programa que da como resultado el propio código fuente puede ser este:

```
program self(input, output);

type
  s = string[255];
  n=integer;
var
  a : array [1..100] of s;
  i,j : integer;

function t(a:integer):integer;
begin
  if a<7 then t:=a else t:=a+11
end;

function q(a:s):s;
var j:n;

begin
  for j:=strlen(a)downto 1 do
    if a[j]=#39 then strinsert(#39,a,j);
  q:=a;
end;

begin (*main*)
  a[1] := 'program self(input, output);';
  a[2] := 'type s = string[255]; n=integer;';
  a[3] := 'var a : array [1..100] of s; i,j :
integer;';
  a[4] := 'function t(a:integer):integer; begin if a<7
then t:=a else t:=a+11 end; function q(a:s):s;';
  a[5] := ' var j:n;begin for j:=strlen(a)downto 1 do if
a[j]=#39 then strinsert(#39,a,j);q:=a;end;';
  a[6] := 'begin';
  a[18] := '    for i := 1 to 11 do begin
setstrlen(a[i+6], 0);';
  a[19] := '        strwrite(a[i+6],1,j,'
a['',t(i):1,''] := ''', q(a[t(i)]), ''';';
  a[20] := '    end;';
```

```

a[21] := '    for i := 1 to 22 do writeln(a[i]);';
a[22] := 'end.';
for i := 1 to 11 do begin setstrlen(a[i+6], 0);
    strwrite(a[i+6],1,j,'    a['t(i):1,'] := '',
q(a[t(i)]), ''');
end;
for i := 1 to 22 do writeln(a[i]);
end.29

```

Si se analiza la estructura de este programa, puede verse de nuevo que dentro del código se encuentra como datos el código mismo. Y éste es el truco de los programas auto referentes. De hecho, parece imposible no hacer esto para resolver la tarea propuesta.

La dificultad con estos dos ejemplos es que los lenguajes utilizados, BASIC y Pascal son imperativos, en donde el compilador (o intérprete) requiere de separar los datos de las instrucciones. Dicho de otra manera, caemos en la idea que von Neumann analizara en los años cincuenta del siglo pasado y que además, diese con un modelo que ha sido probablemente muy parecido al que se genera en la auto replicación de las cadenas de ADN.

Pero existen lenguajes que no hacen distinción entre datos y código. Uno de ellos es Prolog, otro es *Postscript*. Uno más es LISP. A estos se les llaman lenguajes funcionales y pueden manejar las simbologías de maneras más poderosas. La solución del problema del código auto referente puede resolverse en Prolog así:

```

quine :- listing(quine).30

```

La auto referencia en lenguajes como Prolog puede verse en el siguiente código auto referenciable, basado en un problema propuesto por Bertrand Russell que

²⁹ De autor desconocido. Véase http://www.nyx.net/~gthompso/self_pasc.txt

³⁰ <http://rosettacode.org/wiki/Quine#Prolog>

dice así: “En un pueblo hay un barbero que rasura a todos aquellos que no se rasuran a sí mismos”. La pregunta a resolver es: “¿Quién rasura al barbero?”.

El código en Prolog tiene una sola línea:

```
rasura(barbero, Y) :- not (rasura (Y,Y)).  
//el barbero rasura a Y si éste no se rasura a sí mismo.
```

El problema con este código es que no tiene fin. No puede ser resuelto porque hay una contradicción lógica inevitable. Si el barbero no se rasura a sí mismo, entonces sería parte de las personas que no se rasuran a sí mismas, por lo que debería ser rasurado por el barbero, pero ¡ése es él mismo!. Puede verse que aquí el ciclo auto referente es incómodo y da al traste con la lógica del problema. De hecho, ejecutar este programa en una máquina que tenga un compilador de Prolog, nos llevará a un error de falta de memoria (*out of memory*), pues la recursión se convierte en infinita y las computadoras de la vida real no tienen memoria infinita y de ahí el error.

El fenómeno de la recursión se observa continuamente en los lenguajes de programación. Casi todos los lenguajes de alto nivel tienen alguna forma de hacer procedimientos o funciones recursivas, es decir, hacer que se definan ciertos algoritmos de manera auto referente. La dificultad de la recursión es que requiere de muchos más recursos de máquina para ejecutarse, particularmente se utiliza en muchísimas implementaciones de los lenguajes de programación, una estructura llamada “pila” (o *stack*). Una pila puede pensarse como una caja en donde ponemos –digamos– libros. El primer libro que meto en la caja queda en el fondo. Puedo ir metiendo más libros y precisamente hacer una pila con ellos. La cuestión es que el primer libro que entra a la pila es el último que puedo sacar. Eso en esencia es una estructura LIFO (*Last In, First Out*), es decir, el último que entra es el primero que sale.

En computadoras con pocos recursos, particularmente memoria, la recursión no se usa normalmente. En ese caso se utilizan algoritmos iterativos, que hacen la misma tarea que un algoritmo recursivo, pero utilizando muchos menos recursos. De hecho, se puede demostrar que **todo algoritmo recursivo tiene una versión iterativa**.

Es curioso notar que los fenómenos en vida artificial contengan un grado enorme de recursión. Pareciera que éste es el mejor mecanismo para poder replicar información. Pero ya hemos visto que esto requiere de más recursos que la rutina equivalente de manera iterativa. ¿Por qué la recursión en el fenómeno de la vida (real y artificial) tiene tanta relevancia?

La hormiga caótica de Langton y sus bucles

*Una hormiga, vista como un sistema de comportamiento, es muy simple.
La aparente complejidad de su comportamiento en el tiempo es largamente
una reflexión del medio ambiente en donde se halla a sí misma.*

Herbert Simon³¹

La vida artificial busca estudiar el fenómeno de la vida a partir de modelos lógicos simulados con la computadora. El término fue acuñado por Christopher Langton en 1986, en lo que fue la primera conferencia internacional de la síntesis y simulación de sistemas vivientes. Un ejemplo es el juego de la vida de Conway, el cual en la década de los setenta tuvo un *boom* notable, probablemente gracias a Martin Gardner, que publicó un interesante artículo en *Scientific American*, en octubre de 1970. Por ejemplo, se pudo demostrar que el juego de la vida es una máquina universal de Turing. Hay un campo vastísimo para estudiar este autómata celular en 2 dimensiones.

Pero hay otras ideas en este mismo sentido que son aún incluso un enigma en el estudio de la vida artificial. Uno muy interesante es la hormiga de Langton, la cual no es otra cosa que una malla en dos dimensiones blanca. Una “hormiga” de

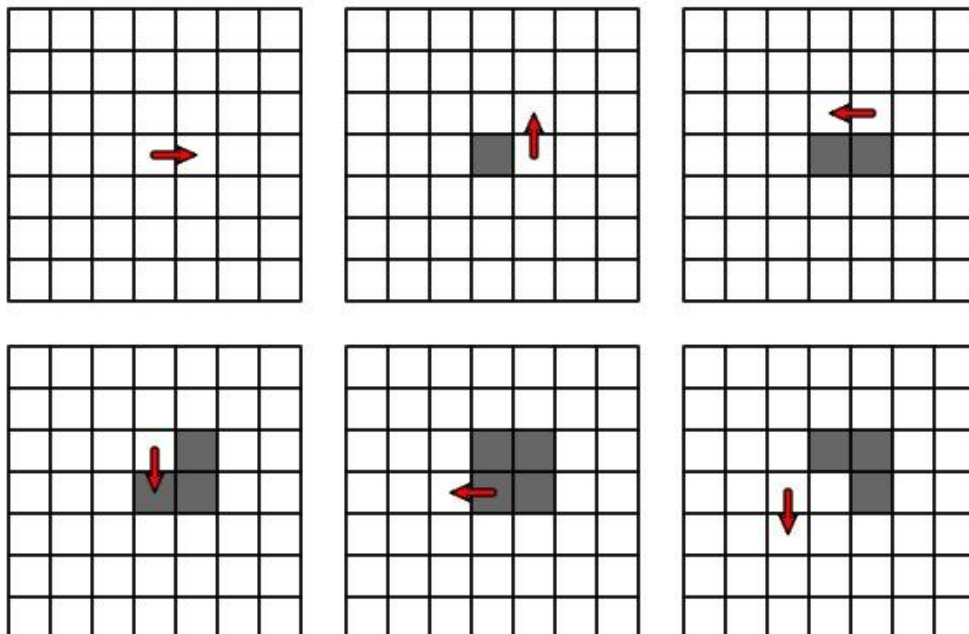
³¹ **The Sciences of the Artificial**; Herbert Simon; Cambridge, MIT Press, p. 64 (1981)

Langton se coloca en medio de la malla y la hormiga puede moverse de acuerdo a las siguientes reglas:

- Si la casilla es blanca, se convierte en negra. Se mueve la hormiga una casilla y se gira la derecha 90 grados.
- Si la casilla es negra, se convierte en blanca. Se mueve la hormiga una casilla y se gira a la izquierda 90 grados.

La hormiga de Langton se define como una máquina de Turing que tiene un resultado emergente muy complejo. Desde el año 2000 hay una prueba sobre la universalidad del autómatas de Langton, es decir, está demostrado que es una máquina de Turing, lo que significa que es capaz de hacer cualquier cálculo.

Hay de hecho maneras de generalizar con más colores este autómatas, por ejemplo usando las llamadas *turmitas*, o mejor dicho, termitas de Turing.



Los movimientos de la hormiga de Langton

Se puede hallar, al ejecutar la hormiga de Langton, tres comportamientos, a saber:

- **Simplicidad.** Durante los primeros doscientos pasos se crean patrones muy simples y frecuentemente simétricos.
- **Caos.** Después de estos primeros doscientos pasos, aparecen patrones irregulares y la hormiga traza caminos considerados pseudo aleatorios.
- **Orden emergente.** Finalmente, alrededor del paso 10,000 se crea una especie de camino, una trayectoria que se repite indefinidamente. Se dice que es un **atractor**, pero no ha sido demostrado. Lo que sí se sabe es que, sin importar la configuración inicial, la trayectoria de la hormiga no está acotada. A esto se le conoce como el **teorema de Cohen-Kung**.

Para experimentar con el comportamiento de la hormiga en una malla totalmente blanca o bien, llena de puntos negros y blancos al azar, se ha escrito un programa en particular (ver Apéndices), que permite observar el desarrollo del autómata celular y cambiar las condiciones iniciales.

Hay que considerar que la hormiga de Langton no es un juego de computadora, pues no se interactúa con el mismo. En ese sentido es similar al juego de la vida de Conway. Es una especie de ventana de observación en el mundo de una hormiga que se mueve a través de su universo bidimensional siguiendo reglas absolutamente ciegas.

La hormiga de Langton fue una idea novedosa, que planteó la siguiente pregunta: ¿Cómo reglas tan simples pueden generar de pronto caos y un atractor extraño? Sin embargo, la investigación puede enriquecerse si se extiende la idea original de Langton. En ese caso, Greg Turk y Jim Propp decidieron agregar colores a la malla de acuerdo a un comportamiento cíclico. Se usó un esquema muy sencillo de “izquierda” (L – left en inglés) y “derecha” (R – right) para asignar qué debía

hacer la hormiga en caso de tener que girar y a qué color cambiar. En este caso, la hormiga de Langton se rebautizó como “RL”.

Lo que hallaron Propp y Turk con esta variación de la idea original de Langton fue que se producían patrones simétricos una y otra vez. Uno de los más simples es la hormiga con las instrucciones “RLLR”. Una condición suficiente en este tipo de hormigas, que provoca que se cicle, es cuando se encuentra con un par de letras idénticas “LL” o “RR”. Aparentemente más investigaciones en este tipo de extensiones a las hormigas de Langton no dieron mayor información.

No obstante, también se han intentado otro tipo de extensiones a las hormigas de Langton, considerando estados múltiples de la máquina de Turing, como si la propia hormiga tuviese un color que pudiese cambiar. A este tipo de hormigas se les llamó “turmitas”, lo cual significa “Turing machines termines”. Se halló en este caso que los comportamientos comunes incluyen producción de atractores extraños, crecimiento caótico e incluso, crecimiento en espiral.

Todo este trabajo se hizo considerando una sola hormiga en el plano bidimensional, pero no hay ninguna restricción por la cual no se puedan poner más hormigas en el mismo “campo de batalla”. De hecho, las hormigas de Langton pueden coexistir en el plano y su comportamiento nos lleva a la conclusión de que estamos ante la presencia de un autómata que colectivamente puede construir una serie de estructuras ampliamente organizadas. También se halló este mismo comportamiento para las turmitas siempre y cuando exista una regla que indique qué es lo que pasa cuando se encuentran en una misma casilla. Por ejemplo, Ed Pegg Jr. Consideró turmitas que pueden ir ambas a la izquierda y derecha, dividiéndose en dos y aniquilándose una con la otra cuando se enfrentan directamente.

Christopher Langton, con su caótica hormiga, halló algo sorprendente en su momento: una serie de reglas que se seguían ciegamente de pronto generaban un

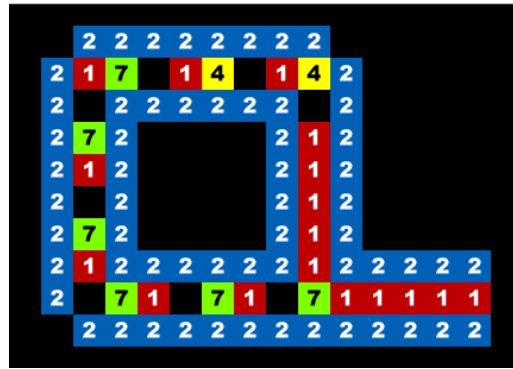
comportamiento emergente, que en el caso de su hormiga, era la de crear una *autopista*, un camino, el cual empezaba después de la generación 10,000 aproximadamente.

Este experimento de vida artificial llevó a Langton a pensar en otras posibilidades y así, eventualmente, llegó a lo que se le conoce como los “bucles de Langton”. En ese caso, hablamos de un autómata celular, el cual se define a partir de unas celdas que tienen un recubrimiento. Esto protege la información genética, la cual circula dentro del bucle. Los genes no son otra cosa que secuencias de instrucciones, las cuales definen claramente cómo crear un nuevo bucle. De hecho, la idea de Langton para su bucle se basó en el trabajo de Edgar Codd³² (1968).

Langton concibió entonces la idea de un autómata que soportara una estructura cuyos componentes tuviesen la información necesaria para poderse reproducir. La estructura es ella misma y su representación de sí misma. Es decir, hablamos de una estructura que se puede definir recursivamente.

El autómata de Langton usaba una serie de estados y 29 reglas. Su estructura se reproduce a sí misma como un bucle, el cual está constituido por una barrera protectora por donde circula la información necesaria para construir un nuevo bucle, necesario evidentemente para la reproducción.

³² Edgar “Ted” Codd, fue un científico inglés (1923-2003), más conocido por sus trabajos en lo que se refiere a bases de datos relacionales. Su tesis doctoral es sobre autómatas celulares, en donde analiza el trabajo de von Neumann y halla una manera de simplificar las 28 reglas de dichos autómatas, en donde usaba solamente ocho estados.



Bucle de Langton: las casillas de la cuadrícula pueden tener 8 estados.

El número dos corresponde a la protección, al recubrimiento del material genético.

La protuberancia que muestra el bucle se llama “brazo de construcción” y es por donde el bucle se desarrollará

Las reglas de Langton para su bucle son:

El material genético va rotando por el bucle en sentido opuesto a las manecillas del reloj, el cual lleva dicho material hasta el brazo de construcción. Este material genético codifica las instrucciones que seguirá el brazo. De hecho hay solo dos:

- 701 (verde-negro-rojo): se le indica al brazo de construcción que avance.
- 401 (amarillo-negro-rojo): se le indica al brazo de construcción que gire 90 grados.

Ejecutar estas instrucciones en la computadora mostrará el siguiente comportamiento: el bucle en primera instancia hará que el brazo constructor avance hasta un determinado punto, en el que girará dos veces. Entonces el brazo se encerrará sobre sí mismo, generando así otro bucle con el mismo código genético. Cuando el brazo se encierra, entonces se produce un nuevo estado que tendrá como consecuencia que se rompa la parte del brazo que une ambos bucles. Cuando la unión ha sido rota, cada bucle generará entonces un nuevo brazo constructor en alguna dirección en la que no haya otro bucle. Cuando el bucle está completamente rodeado, se queda sin material genético y se detiene.

Un ejemplo de este comportamiento se ha visto en el método de reproducción que sigue un coral, en donde solamente los elementos exteriores del coral son quienes hacen crecer a éste.

Capítulo IX

La teoría evolutiva moderna, según Kauffman

La vida artificial es una manera de explorar cómo los sistemas complejos pueden exhibir auto-organización, adaptación, evolución, metabolismo, y todo este tipo de cosas

Stuart Kauffman

(Origin of Order: Self-Organization and Selection in Evolution)

Stuart Kauffman es uno de los biólogos teóricos más importantes del mundo. Está muy relacionado con la vida artificial porque sus investigaciones lo han llevado a proponer una serie de conceptos que sin duda podrían generar el siguiente paradigma en la definición de vida. El problema central de Kauffman empieza con la interrogante de ¿cómo es que existe la vida? ¿Qué artes hubo que pasar la Naturaleza para que de pronto, elementos inanimados cobraran vida? Entender el mecanismo de creación de vida sin duda nos haría entender no solamente nuestra propia existencia, sino además, tendríamos al menos pistas importantes para sintetizar otras formas de vida.

Ya von Neumann había hecho notar que la vida no puede existir si no hay un grado de complejidad presente. Una vez alcanzado este estado, se empezaría un

mecanismo evolutivo para crear moléculas más complejas y en algún momento llegar a los organismos. Evidentemente este sería un largo y tortuoso camino de prueba y error constante. Kauffman hace notar que la probabilidad de que los sistemas complejos surgieran es mucho menor de lo que la gente cree. Quizás se nos olvida que la Naturaleza no tiene prisa y que eventualmente se pueden dar las condiciones para la creación de la vida.

La historia de Kauffman empieza en Dartmouth, donde estudió filosofía orientada a la lógica. Se convirtió en un experto en lógica booleana de hecho, en donde sólo hay dos valores con los que se trabaja, verdadero y falso, y una serie de conectores lógicos como son AND, OR, NOT y el OR exclusivo. El siguiente paso del recién graduado filósofo fue ir a Oxford, donde obtuvo la beca Marshall. Kauffman quería estudiar la filosofía de la mente, pero por alguna razón la biología teórica le llamó poderosamente la atención. De hecho, por razones que ni él sabe explicar, se hizo médico, en lugar de biólogo.

Entre los estudios que tuvo que hacer, fue el tratar de comprender la diferenciación de las células humanas, algo que aún es en muchos sentidos un misterio biológico. ¿Cómo es que un huevo fertilizado de pronto empieza a duplicar sus células hasta que llega un momento en que éstas empiezan a diferenciarse para crear nervios, músculos, tejidos diversos? Porque todas las células contienen el mismo ADN pero claramente un músculo no es un nervio, por ejemplo. Kauffman trabajaba en eso cuando dos investigadores, Jacob y Monod, hallaron un mecanismo que pudiese explicar este comportamiento de diferenciación, al cual llamaron “switching de genes” (del original en inglés, gene switching). Esta teoría indicaba que en algún momento las células apagan unos genes mientras que encienden otros.

La idea de los investigadores franceses parecía abrir nuevas alternativas a la investigación de la biología genética. Bastaría aislar los componentes para ver cómo se comportaban, pero el trabajo parecía fuera de toda posibilidad real, pues

los seres humanos poseen unos cien mil genes, y si consideramos la cantidad de combinaciones que se pueden dar, el análisis que habría que hacer parece sobrepasar todas las expectativas.

Kauffman sin embargo, pensaba que el problema era posible de atacar, pero siguiendo otro enfoque. En lugar de aislar elementos y ver qué genes eran los que participaban en este esquema que los habilitaba/deshabilitaba, se le ocurrió una idea que no sería vista con buenos ojos. Sugería utilizar un conjunto de instrucciones azarosas que podrían terminar por convertirse en algo complejo que de manera casi absurda, podría considerarse como la vida misma. El investigador creía en la capacidad de la auto-organización en donde en una red de conexiones trabajaría al azar y en algún momento, emergería lo que llamamos vida.

Así pues, no existe un plan divino para aclarar las confusiones de los genes, sino la acumulación de las fuerzas locales de todo el sistema que se comportarían que no podría predecirse a partir de las condiciones iniciales del sistema. Y Kauffman iba más allá: para él eran los poderes de la auto-organización y no un plan de diseño de una red de microcircuitos genéticos que dictaban, finalmente, la manera en cómo funcionaba la red de *switcheo* de genes.

La idea fue revolucionaria en muchos sentidos y Kauffman insistía que ése era el proceso de la vida. Esto crearía organismos complejos, nuevos, incluso podría ¿por qué no? Crear organismos más complejos que los seres humanos. De hecho, su idea pretendía llegar a la conclusión de que el mundo no era particularmente de una manera sino que en el sentido más matemático, era general, natural, “fundamental e inevitable”, de acuerdo a sus palabras.

Cabe señalar que estas eran las ideas de Kauffman en 1965. Y en ese entonces, para probar sus asertos, decidió conseguir alguna manera de fondearlos. Consiguió el poder usar una máquina IBM, un *mainframe* de ese entonces (estábamos lejos de la computadora personal), a cambio de unos 1000 dólares,

que le dio la universidad donde estudiaba aún medicina. Entonces escribió un programa en FORTRAN que generaba una red booleana, similar conceptualmente a las redes de switcheo de genes. En una red booleana los valores solamente pueden ser verdaderos o falsos, hay corriente o no, “0” ó “1”.

Kauffman comenzó con mil variables booleanas, un valor exageradamente menor a la cantidad de genes que tienen los seres humanos. Se pusieron reglas azarosas que encendían o apagaban esos bits. Imagínenlos como pequeñas lamparitas. Kauffman mismo no sabía qué iba a pasar pero mantendría su programa corriendo paso a paso para ver qué pasaba.

Pero ¿qué podría esperarse? Si los cambios en los bits eran al azar, de acuerdo a reglas ciegas, el sistema podría mantenerse en un estado azaroso por mucho tiempo. Llegaría un momento que se estableciera algún patrón interesante? ¿Qué emergiera alguna propiedad? Kauffman invirtió casi con fe ciega esos 1000 dólares y empezó la simulación. Primer paso, una serie de bits prendidos y otros apagados. Segundo paso igual. Tercer paso, similar... pero de pronto el sistema tuvo un cambio notable, en el paso catorce ¡se repitió el paso 10! Y de ahí en adelante el programa se cicló en una serie de estados reconocibles.

Increíblemente Kauffman parecía haber acertado. Cambio configuraciones y de nuevo, al poco tiempo la máquina le mostró una serie de patrones que se repetían. Había logrado la auto-organización. Esto, no lo sabía Kauffman en ese entonces, se llamaba en teoría de caos un “atractor periódico” pero a pesar de ello, su experimento terminó por definir lo que a la larga se llamaría el “modelo de Kauffman”.

Pero ¿qué tiene que ver esto con todo el asunto de la genética? Kauffman había demostrado que el switcheo de genes no necesariamente es especializado. Digamos que la máquina genética tiene una serie de órdenes en donde se emergen nuevas características pero no vía un diseño estructurado específico,

sino a partir de un mecanismo como el que Kauffman había usado en su experimento. Eso explicaría, por ejemplo, porqué hay relativamente pocos errores en la creación de un ser humano y esto, genéticamente hablando, no parece incidir en la vida de quienes quizás, tienen un gen que no debería ir donde va. Es decir, en este intercambio genético, en este switcheo de genes, algunas operaciones bien podrían no incidir en la vida del individuo que se está creando y eso parece pasar en la mayoría de los casos.

Capítulo X

Los sistemas L de Lindenmayer

El desarrollo de un organismo... puede ser considerado como la ejecución de un 'programa de desarrollo' presentado en el huevo fertilizado [...] Una tarea central de la biología es descubrir el algoritmo detrás del curso de desarrollo.

Aristid Lindenmayer (*Automata, Languages, Development*, 1976)

Arístides Lindenmayer (1925-1989) fue un biólogo húngaro especializado en botánica. Poco después de que terminara la Segunda Guerra Mundial, se fue a vivir a los Estados Unidos. Hizo un doctorado en fisiología de las plantas, por la Universidad de Michigan. Sin embargo, sus intereses se desviaron hacia la parte formal de las matemáticas, la lógica, por lo que se fue a hacer estudios postdoctorales al Reino Unido, con el especialista en lógica H. Woodger.

Lindenmayer tenía ya algunas interesantes ideas sobre cómo axiomatizar los fundamentos de la biología. Y en algún momento concibió un sistema para describir el desarrollo de las plantas. Esto significa simplemente que usó la teoría de los lenguajes formales, una gramática rígida, matemática, con reglas absolutamente claras, las cuales pasaban a ser parte de la teoría de algoritmos en ciencias como la computación.

La idea de Lindenmayer, sin embargo, no era del todo original. Ya muchos científicos habían atacado el código genético en términos de un lenguaje formal, el cual se expresa la vida usando un alfabeto de sólo cuatro letras el cual, de alguna manera, dicta cómo se desarrollan los organismos.

De alguna manera, el sistema desarrollado por Lindenmayer tiene una fuerte relación con los autómatas celulares. Su sistema evoluciona a pasos discretos en el tiempo, un paso a la vez, en donde los símbolos van cambiando de acuerdo a reglas de transformación. Lo más interesante en todo esto es que en la definición de sus reglas, hay involucrada la recursión, que parece ser está embebida en la naturaleza.

Un sistema L, o sistema de Lindenmayer es simplemente un conjunto de reglas que trabajan sobre símbolos. En términos teóricos a esto es lo que se le llama ***una gramática***. Lindenmayer halló que bajo este esquema se podían modelar procesos biológicos, por ejemplo, la manera en cómo se forma un número de organismos.

Como ya mencionamos, los sistemas L son recursivos y por ende, hay el fenómeno de la auto-semejanza, tal y como se da en los fractales, inclusive. Veamos cómo se definen y qué nos dicen:

Un primer ejemplo, que Lindenmayer usó para entender el crecimiento de las algas se basa en los siguientes elementos:

variables: A B

constantes: ninguna

inicio (generación cero): A

reglas: $(A \rightarrow AB)$, $(B \rightarrow A)$

Esto quiere decir que

- El símbolo A, al ser aplicado, genera AB.
- Y B, al ser aplicado, genera A.

Puede verse cómo la recursividad está ahí presente. Al usar estas reglas de transformación, hallamos:

$$n=0: A \rightarrow AB$$

para la generación llamada cero. Una vez que hemos obtenido el resultado de la misma, la ponemos en la entrada y así obtenemos la primera generación, a saber:

$$n=1: AB \rightarrow ABA$$

Podemos así seguir con las siguientes generaciones y hallaremos:

$$n=2: ABA \rightarrow ABAAB$$

$$n=3: ABAAB \rightarrow ABAABABA$$

...

Puede verse que esto es un simple esquema de reglas que se aplican ciegamente pero que tienen un comportamiento no decidible, es decir, hay que hacer la simulación completa para ver cada generación.

Veamos un nuevo ejemplo, en este caso el que representa a la serie de Fibonacci. Esta es la definición del sistema:

variables: A B

constantes: ninguna

inicio (generación cero) : A

reglas: (A \rightarrow B), (B \rightarrow AB)

el cual produce la siguiente secuencia de cadenas:

n=0 : A
n=1 : B
n=2 : AB
n=3 : BAB
n=4 : ABBAB
n=5 : BABABBAB
n=6 : ABBABBABABBAB
n=7 : BABABBABABBABABBABABBAB

Cuando se mide la longitud de cada cadena (sin importar si son As o Bs), se obtiene la famosa secuencia de los números de Fibonacci, 1 1 2 3 5 8 13 21 34 55 ...

Uno podría pensar: Todo esto se ve interesante, pero ¿dónde se puede visualizar el crecimiento de algunas plantas con este sistema? Para 1970, dos de los alumnos graduados de Lindenmayer, Ben Hesper y Pauline Hogeweg, se preguntaban si los sistemas L podrían tener una representación visual de lo que se supone estaban modelando en las plantas. Aparentemente a nivel microscópico las ideas de Lindenmayer eran ciertas.

Entonces Hesper y Hogeweg trabajaron en un programa de computadora por dos semanas para interpretar las reglas de los sistemas L, buscando así generar ramas de las plantas, por ejemplo. Hogeweg aclara que los sistemas L no generan imágenes. En realidad generan largas cadenas de caracteres, de símbolos. Así pues, se necesita un paso extra para generar ramas, para visualizar lo que Lindemayer representa con sus reglas.

Lo increíble del asunto es que lo que hay que hacer para visualizar los sistemas L no es particularmente complicado. En este caso se puede usar utilizar *Logo*³³ (o una variante de las gráficas de la tortuga – *turtle graphics*), un lenguaje de programación de alto nivel, inventado con fines de enseñanza por Seymour Papert, Danny Bobrow y Wally Feurzeig. El lenguaje tiene sus reminiscencias en LISP (*LISt Processing*), que por muchos años se ha usado en el tema de la inteligencia artificial, junto con Prolog.

Una idea explotada en Logo es el uso de un pequeño robot que puede pintar en un papel, moviéndose de acuerdo a las instrucciones de un programa. Seymour y colegas hicieron una tortuga robótica, la cual eventualmente se incorporó a la programación como una tortuga virtual, que resulta más barato e igualmente funcional. Ya en la pantalla de la computadora, la tortuga es un triángulo que podríamos decir, se ve “desde arriba”.

Un ejemplo de las instrucciones puede ser este:

forward 10 la tortuga se mueve 10 pasos
turnright 90 la tortuga gira a su derecha 90 grados
turnleft 30 la tortuga gira a la izquierda 30 grados

El programa original de Hogeweg y Hesper mostró una representación gráfica de la teoría de Lindenmayer. De hecho, con las imágenes resultantes, que se parecían notablemente a ciertas plantas y que son finalmente una representación gráfica de las reglas de transformación en los lenguajes formales y que les llamaron “morfemas”, se las presentaron a Lindenmayer que, curiosamente, no mostró mucha emoción al verlas. Hogeweg indica “no le gustó nuestro trabajo [a Lindenmayer]”. La razón es que para el creador de los sistemas-L , estos eran parte de una teoría matemática y que ésta no debería diluirse en producir

³³ *Logo* es una creación del Massachusetts Institute of Technology (<http://el.media.mit.edu/logo-foundation/logo/programming.html>).

imágenes. Aun así, el propio Lindenmayer tuvo que aceptar que al menos, las imágenes producidas eran una curiosidad atractiva sin consecuencias y tan lo aceptó que usó algunas de estas imágenes para sus tarjetas de fin de año.

Un ejemplo de este tipo de imágenes, con sólo 5 iteraciones, 22 grados de giro, con un axioma F y con una sola regla: $F = C0FF-[C1-F+F+F]+[C2+F-F-F]$, se genera lo siguiente:



En el apéndice III puede verse un programa en HTML5 (que corre en línea, dentro del navegador de Internet), el cual permite visualizar las ideas de Lindenmayer de manera asombrosa.

Hay muchas más imágenes creadas a partir de las ideas de los sistemas L. Por ejemplo:



Un helecho creado con los sistemas L de Lindemayer

O bien una expresión mucho más elaborada, usando gráficas tridimensionales:



(ver http://en.wikipedia.org/wiki/File:Dragon_trees.jpg)

Lo biomorfismos de Richard Dawkins

Lo que no puedo entender es por qué no pueden ver la extraordinaria belleza de la idea de que la vida surgió de la nada – eso es una cosa tan asombrosa, elegante, y maravillosa, ¿por qué querer saturarla con algo tan complicado como un Dios?”

Richard Dawkins³⁴

Uno de los científicos más importantes de nuestro siglo es Richard Dawkins. Nacido en el Reino Unido el 26 de marzo de 1941, etólogo, biólogo, escritor y creyente en la evolución, se convirtió en una celebridad con su trabajo de 1976 “The Selfish gene” (El Gen Egoísta), que popularizó la visión de la evolución a partir de los genes. Dawkins es quien acuñó el término “meme”, el cual es una especie de unidad de lo que Dawkins llama información cultural que se transmite de una persona a otra o incluso de una generación a la siguiente.

Dawkins piensa que tenemos dos maneras de procesar la información:

- El genoma, que es el sistema genético que está en los cromosomas de cada persona y que determina el genotipo. Esto es el ADN finalmente. Esto constituye la naturaleza biológica de los seres humanos. A través del

³⁴ Conversación con el Arzobispo de Canterbury, Dr. Rowan Williams (*The Telegraph*, 24/02/2012).

procedimiento de replicación, de hacer copias de sí mismos, los genes se transmiten en lo que llamamos el mecanismo de la herencia.

- El cerebro, que nos permite procesar información cultural recibida por diferentes medios: imitación, enseñanza, asimilación, etcétera, y Dawkins es precisamente a lo que llama, como analogía de los genes y queriendo mantener la expresión, les bautiza como “memes”.

El renombrado científico piensa que los rasgos culturales, los memes, también tienen este mecanismo de replicación. Se ha buscado compararlos con los conceptos conocidos de los cromosomas, intentando agruparlos en dimensiones culturales, es decir, categorizarlos.

Richard Dawkins es presidente de la Asociación Humanista Británica. Es un crítico duro del creacionismo y de la teoría del diseño inteligente. Por ejemplo, en “The Blind Watchmaker” (*El Relojero Ciego*), de 1986, considera que la analogía de que para que exista un reloj es necesaria la existencia de un relojero, es falsa, y que todos puede explicarse a través de reglas ciegas que además, van evolucionando hasta poder crear un reloj, valga la comparación, teniendo un relojero que es invidente, que simplemente no puede ver lo que está haciendo pero que poco a poco, evolutivamente, termina por construir un mecanismo tan complejo como podría ser un reloj.

El autor, en “El Relojero Ciego”, analiza los argumentos creacionistas. El más importante sea que la existencia de la compleja vida es simplemente una razón lógica para creer en la existencia de Dios. Dawkins piensa que incluso lo más complejo de la vida puede verse como una acumulación de pasos evolucionistas que nos han llevado a este nivel de complejidad. Hace un análisis muy conocido sobre el ojo humano, un mecanismo que muchos no pueden siquiera pensar que fuese producto de la evolución natural, pero Dawkins da argumentos para apoyar sus tesis evolucionista.

Lo que nos interesa a nosotros es la creación de los biomorfismos, que es un programa de computadora, desarrollado por el propio Dawkins, que demuestra el poder de las micro-mutaciones y de la selección acumulativa.

Dawkins empieza a partir de un algoritmo recursivo³⁵ para dada una iteración, se genera una nueva conexión. El plan original era generar formas arbóreas. Empieza con un tronco y a cada nueva iteración, se crea una nueva sub-rama. El uso de los biomorfismos entonces mostró que el algoritmo no estaba limitado solamente a la creación virtual de diferentes árboles, sino que podía generar otros tipos de formas, incluso las biológicas. Dawkins estaba poco menos que perplejo al descubrir un biomorfismo se parecía notablemente a un insecto, seguido de otras formas como aviones, murciélagos, etcétera.

Para poder jugar con los biomorfismos, se utiliza el ojo para precisamente jugar el rol de la selección natural. Dada una forma específica, el usuario selecciona el biomorfismo parecido visualmente a algo particular. Este parecido puede ser incluso muy sutil y no tiene porqué ser obvio. Después de un número de generaciones, el resultado puede sorprendernos por el parecido obtenido a la forma que esperábamos.

El visor de biomorfismos (el programa de Dawkins), permite ir a través de las opciones que nos permiten hallar el genoma que se parece más a la forma buscada. El espacio genético de Dawkins para sus biomorfismos es de unos 500 mil millones de combinaciones. A través de esta selección acumulativa, se puede hallar el genoma deseado en este espacio de 9 dimensiones en pocas iteraciones.

Pero con todo este universo de iteraciones, ¿cómo es posible esto? El mecanismo es la selección acumulativa. Para entenderla, podemos pensar en que queremos

³⁵ La recursión ya la revisamos en el capítulo "Auto referencia y recursión". Nótese la importancia de este tema cuando se habla de conceptos como la vida.

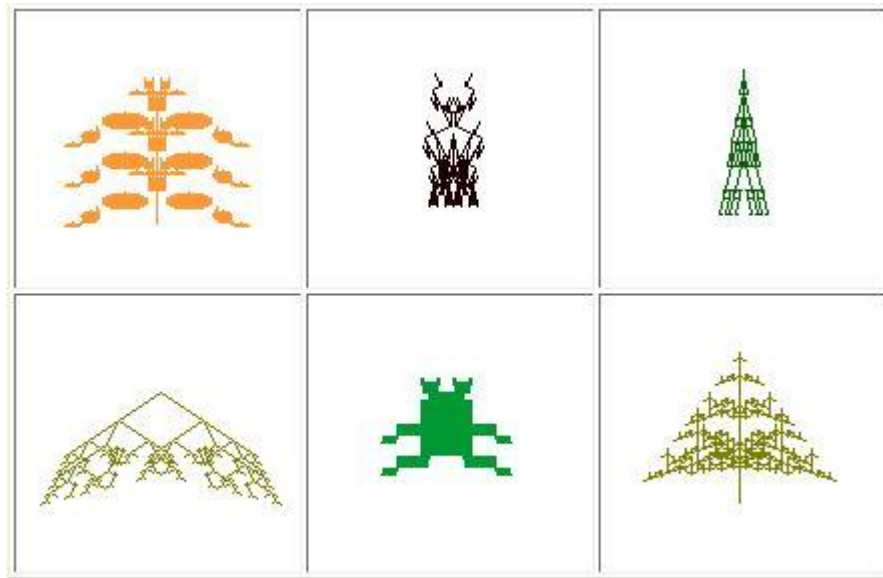
saber la clave de una tarjeta de crédito, que es de solamente cuatro números. Necesitamos máximo 10,000 combinaciones para dar con el número correcto. Sin embargo, si por ejemplo, después de dar con la primera cifra, se diese un aviso a quien busca, no tendría que barrer el espacio completo de las diez mil combinaciones, pues bastarían máximo 40 combinaciones para probar, pues no se tendrían que hacer más que diez intentos para cada número.

Y esto mismo es lo que pasa con la acumulación selectiva: para cada nueva iteración, el biomorfismo se acerca a la forma que será seleccionada. Dicho de otra manera, Dawkins propone que no se busca en el espacio completo del genoma, sino que en cada iteración se hace una selección progresiva de los componentes que vamos a necesitar.

Es importante decir que cada biomorfismo tiene un genoma de 11 enteros, con posibles valores que van de 0 a 20. Cada biomorfismo se dibuja como un árbol binario. El algoritmo para dibujar está influenciado por los genes de los biomorfismos individuales. Por ejemplo, el primer gene afecta la profundidad del árbol, en donde además, tres de esos genes afectan el color de cada nivel y otros genes afectan la longitud de los segmentos usados para dibujar cada nivel.

Los biomorfismos implementan en todo caso un modelo de evolución que es muy simple pero muy ilustrativo. En este modelo, el usuario selecciona el individuo que sobrevivirá a la siguiente generación solamente por razones estéticas (de ahí que la visión del usuario sea el rol del mecanismo de la evolución). Algunos biomorfismos se parecen a insectos y a otros objetos conocidos.

He aquí algunos biomorfismos:



Biomorfismos de Dawkins

El algoritmo de Dawkins es recursivo en donde se involucran algunos genes y mutaciones, así como un mecanismo de selección relativamente azaroso. Los fenotipos, es decir, como se ven los biomorfismos en el sistema de Dawkins son estructuras que empiezan como ya dijimos, como los árboles, con un tronco principal y ramas que van saliendo de acuerdo a las reglas programadas. El sistema básico incluye:

- Ángulo para el ramaje
- Profundidad del ramaje
- Número de líneas

Existen muchísimos programas de código abierto en donde se pueden dibujar los biomorfismos de Dawkins.³⁶

³⁶ Por ejemplo, <http://www.codeproject.com/Articles/17387/Al-Dawkins-Biomorphs-And-Other-Evolving-Creatures>

Capítulo XII

La genial idea de John Holland

Los programas de computadora que evolucionan en formas que imitan la selección natural pueden resolver problemas complejos, incluso algunos que sus propios creadores no entienden completamente.

John Holland

Los esfuerzos hechos en esta novísima ciencia llamada “vida artificial” empezaron con autómatas celulares unidimensionales, donde una serie de reglas ciegas, simples, generaban un comportamiento emergente complejo, es decir, de pronto el comportamiento esperado no ocurría, sino que se transformaba en algo que no es fácil de explicar, como en el caso de la hormiga de Langton, por ejemplo. Y todo esto llevó a los investigadores a prestar más atención a estos fenómenos. Uno de ellos fue John Holland, un verdadero pionero en este tema.

Nacido en 1929 en Indiana, Estados Unidos, John, de pequeño, ya mostraba ciertas capacidades para las ciencias. Era bueno en matemáticas y física y durante su último año en la preparatoria, hizo un examen en el que quedó como el tercero mejor, lo que le dio la oportunidad de una beca en el MIT. Ahí fue donde Holland empezó a trabajar sobre simulaciones de la evolución natural. Recibió su doctorado en ciencias de la computación y literalmente estaba asombrado y fascinando por la idea de construir redes de neuronas artificiales, que finalmente

eran sólo el principio de las neuronas reales, con las cuales creaba memoria y comportamiento emergente complejo.

John Holland se convirtió en un experto en cómputo e IBM lo invitó a colaborar con un grupo notable de ingenieros para planear y construir la primera calculadora, la “701”. Para probarla, se implementó un sistema de red de nervios y la computadora servía entonces como si se hablase de una rata de laboratorio. Esto le dio la pauta a Holland de que había una notable liga entre la biología y la computación. Las computadoras podían ser entrenadas para simular comportamientos que solían verse en los animales. Pero el punto de quiebre para Holland fue el libro “The Genetical Theory of Natural Selection”³⁷, que literalmente cambió la vida de científico.

El enfoque del libro planteaba que la evolución era como aprender una forma de adaptación al medio ambiente. Y esto ocurre de generación en generación, por muchas de ellas y no en el transcurso de una vida. Holland pensó entonces que si esta teoría era cierta para la Naturaleza, por qué no podría funcionar para programas de computadora. Y es aquí donde surge la gran idea, la del “algoritmo genético”, el cual es una serie de instrucciones, “una receta de cocina”, para analizar los problemas basándose en la teoría de Darwin sobre la selección natural, teoría que hoy en día nadie duda que sea cierta.

La idea -para expresarla brevemente- empieza con una población inicial de nodos individuales, cada uno con características particulares generadas al azar. Cada uno de ellos es evaluado por un método en particular y así ver cuál es el más exitoso. Los mejores, los de mayor éxito, se unen en una especie de “niño” que tiene la combinación de todas las características del padre. Y esto es realmente notable. La idea de Holland es un parteaguas porque utiliza la evolución para proveer de un mecanismo poderoso para desarrollar optimizaciones en una

³⁷ El autor es R.A. Fisher y el libro se publicó antes de 1923. Actualmente se puede conseguir la versión (prácticamente) escaneada en amazon.com por unos 20 dólares.

computadora y provee una ventana para quienes trabajan en la evolución. Es una manera muy particular de estudiar muchos fenómenos naturales.

¿Pero cómo llegó Holland a estas conclusiones? Él sabía que los organismos vivos pueden resolver problemas complejos y que exhiben una versatilidad que hace palidecer a las mejores computadoras que tenemos actualmente. Muchos científicos notaron que muchas de las soluciones que presentan los seres vivos en su entorno eran parte de la evolución y de la selección natural. Por ello, se buscó entonces emular estas ideas. La selección natural, por ejemplo, elimina un problema típico de la programación: la necesidad de especificar de antemano todas las características de un problema y de las acciones que el programa debe tomar para poderlo resolver. Mediante el mecanismo de la evolución, se pudieron “criar” programas que resolvieran problemas que de hecho nadie entendía perfectamente la estructura del mismo. Estos son precisamente los llamados algoritmos genéticos, los cuales se han usado en temas por demás variados y no sólo en la vida artificial.

Los algoritmos genéticos hacen posible el explorar un rango muchísimo más amplio de soluciones potenciales a un problema que lo que se puede hacer con programas convencionales. En algún sentido es un nuevo paradigma, el cual se puede presentar parecido al de la programación lógica en Prolog, el cual -a través del algoritmo de Robinson³⁸ se recorren todas las posibles soluciones a un problema planteado, en donde siempre cabe la posibilidad que no haya solución o que ésta sea no-decidible en un tiempo finito.

Así pues, los programas que usan este mecanismo de la selección natural pueden, bajo ciertas condiciones controladas, mostrarnos algunos detalles y dar más luz sobre cómo la vida y la inteligencia se desarrolló en el mundo real. La razón de esto parece estar basada en dos procesos elementales: la selección

³⁸ Véase <http://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/logic.htm> para una discusión formal del tema.

natural y la reproducción sexual. La primera determina qué miembros de la población sobreviven y se reproducen y la segunda hace algo quizás más importante aún: mezcla y recombina los genes para los siguiente seres en la nueva generación. Esto hace que las creaturas creadas evolucionen más rápidamente, pues no solamente contienen la copia original de los genes del padre, sino que incluso hasta mutaciones pueden darse.

Por otra parte, si un organismo, por ejemplo, no pasa la prueba de supervivencia planteada en el programa, entonces muere. ¿Qué tipo de prueba es ésta? Se trata de reconocer un depredador (y entonces huir), para poner un ejemplo. No darse cuenta de esto conlleva la muerte, la eliminación del programa mismo. Y este mecanismo de supervivencia lo usamos los seres humanos desde hace siglos desde que aprendimos a criar diferentes razas de animales.

Todo esto suena bien pero pasarlo a un programa de computadora es una labor que dista de ser trivial. El problema principal es la de crear esta especie de “código genético”, que puede representar la estructura de diferentes programas. De hecho, piénsese cómo sería una mutación de un programa de computadora. Si cambia una instrucción simplemente, podría ser la diferencia entre un programa funcional y uno inútil.

Los primeros intentos de hacer cómputo evolutivo, por llamarle de una manera, fue a fines de los años cincuenta y principios de los sesenta del siglo pasado. Fueron poco exitosos por varias razones: la falta de computadoras poderosas y el prohibitivo costo del uso de las mismas. Hans J. Bremermann, de la Universidad de California (Berkeley), sin embargo, añadió una característica que le dio más poder a los programas, pues permitía el cruce de dos creaturas para así compartir código genético. Este procedimiento era muy limitado en su momento, pero fue el inicio de mejores posibilidades. Fue claro que la recombinación de grupos de genes era parte crítica de la evolución y ya para mediados de los años sesentas

se había desarrollado una técnica de programación, que a la postre se llamaría el algoritmo genético, que podría lidiar con las mutaciones y con la evolución misma.

El resultado de esto fue un sistema de clasificación, que consistía en un conjunto de reglas, cada una desarrollando acciones específicas cada vez que las condiciones eran satisfechas por algunas partes de la información. Las condiciones y las acciones se representaban como cadenas de bits, ceros y unos, correspondiendo a ausencia o presencia de características específicas en las entradas y salidas de datos de las reglas. Por ejemplo, para cada característica presente, la cadena contendría un “1” en la posición apropiada y para una característica faltante, entonces el valor sería “0”. Cabe decir que estas características se definen en base a lo que estamos estudiando. Si se trata de animales, bien podríamos decir que hablamos de características en “1”, por ejemplo, “tienen pelo”, “corren rápido”, etcétera. Sin embargo, el programador debería tender a elegir las características más simples y primitivas, de manera que puedan ser combinadas como en el juego de las 20 preguntas.³⁹

Un sistema de clasificación, para resolver un problema, empieza con una población de cadenas de bits con “0”s y “1”s al azar y se valora cada una de ellas en términos de su supervivencia. Por ejemplo, las cadenas de más calidad se reproducen, las de menor calidad perecen. En la medida que pasan las generaciones, las cadenas asociadas con soluciones mejoradas serán predominantes. Más aún, el proceso de reproducción continuamente combina estas cadenas de nuevas formas, generando soluciones incluso más complejas.

Para ponerlo en términos del algoritmo genético: la búsqueda de una buena solución a un problema es la búsqueda de una cadena particular de bits. El universo de todas las posibles cadenas debe ser considerado como si fuese la

³⁹ Este es un juego clásico que se juega desde el siglo XIX. Se trata de que un jugador piensa en un objeto y el otro jugador, haciendo preguntas simples, de respuestas “sí” o “no”, llega a dar con el objeto pensado. Es de hecho un interesante ejercicio en cómputo para aprender a usar árboles binarios.

imagen de un escenario en donde los valles son las soluciones menos factibles y el pico de las montañas las más factibles.

Una de las ideas del algoritmo, que finalmente es una de las técnicas convencionales en las búsquedas, es la llamada “búsqueda de escalar montañas”(*hill climbing search* en inglés): se inicia en un punto al azar y si una ligera modificación mejora la calidad de la solución, se continúa en esa dirección; en caso contrario, se va a la dirección contraria. Sin embargo esto es un enfoque muy simplista porque estos escenarios tienen muchos puntos altos, lo que equivale a estar hablando de problemas complejos.

Lo importante es que Holland descubrió un procedimiento que permite de alguna manera evolucionar de acuerdo a los preceptos darwinianos: la capacidad de supervivencia y la reproducción.

Capítulo XIII

El dios darwiniano de Danny Hillis

Tu genoma sabe mucho más sobre tu historial médico que tú.

Danny Hillis

William Daniel Hillis, a quien le apodan “Danny” (nacido en 1956), ha pensado en que hay dos enfoques en el tema de la vida artificial. El primero es la creación de sistemas vivos pero claramente esto suena demasiado ambicioso. El otro enfoque es más simple y posible de obtener: usar la simulación por computadora para tratar de entender los procesos que implican la vida misma.

Hillis desde chico tuvo la motivación adecuada para dedicarse a la ciencia. Su padre era un epidemiólogo y su madre tenía un doctorado en bioestadística. Para Danny, hablar de ciencia en su casa era hablar de biología.

Por ello no es sorprendente que quisiese estudiar neurofisiología en el MIT. Ahí conoció a Marvin Minsky, uno de los pioneros de la Inteligencia Artificial. Hillis, antes de entrar al tema de la vida artificial, trabajó sobre el desarrollo de cómo enseñar a los niños a programar computadoras. Ahí es donde hizo contacto con Seymour Papert, uno de los creadores del lenguaje LOGO, del cual ya hemos hablado antes.

Sin embargo, la conexión intelectual con Minsky fue notable y este último se convertiría a la postre en el mentor de Hillis. Las discusiones entre ambos personajes sobre procesos emergentes e inteligencia artificial eran interminables. Es claro que Hillis aprendió mucho de Minsky, pudiendo apreciar cómo al combinar cosas simples se creaban cosas complejas. Hillis declararía alguna vez: “Las computadoras son un ejemplo de esto así como las neuronas. Y encuentro tan interesante al transistor como a las neuronas”.

A principios de los años setentas del siglo pasado, el laboratorio de Inteligencia Artificial de Minsky trabajaba sobre cognición, robótica y procesos asociados con la inteligencia y Hillis se convirtió en el estudiante estrella en donde fue urgido por su mentor a aprender matemáticas, tema del que se graduó en 1978. Después hizo una maestría en ingeniería eléctrica e informática, obteniendo el grado en 1981.

La inteligencia artificial en los años ochenta del siglo pasado, notó una especie de decepción generalizada. Por alguna razón las grandes esperanzas depositadas en esta ciencia no estaban cumpliendo las expectativas. Probablemente parte de la dificultad es que se creían entender los problemas, pero estos parecían que eran mucho más complicados de lo que se había supuesto.

Hillis estaba convencido que la dificultad empezaba en las máquinas que se usaban, las cuales no eran más que computadoras que ejecutaban las instrucciones secuencialmente, las cuales estaban modeladas por la llamada *arquitectura von Neumann*.⁴⁰ Y para Hillis el no modelarlas como los mecanismos que despliegan los seres vivos, era el gran problema de la inteligencia artificial. Un ejemplo de esto se observaba claramente en la *teoría de los sistemas expertos*, los cuales buscan emular el conocimiento de un experto en un tema en particular, por ejemplo, la capacidad de diagnóstico de un médico. Hillis notó que mientras

⁴⁰ Para más información sobre este tipo de arquitectura de computadoras, consúltese http://es.wikipedia.org/wiki/Arquitectura_de_von_Neumann.

más información se le da al sistema experto, más tarda en hallar una solución, cosa contraria a lo que un experto humano hace. Mientras más información tiene el ser humano de un tema, más rápido puede llegar a una conclusión. Es claro que la naturaleza de -al menos- ciertos procesos humanos no estaba en las computadoras y esto parecía ser decisivo.

La solución entonces parecía simple para Danny Hillis: Constrúyanse computadoras capaces de procesar información como hacen los seres humanos, en paralelo. Así entonces, bajo el modelo del cerebro humano, Hillis buscaba que unidades de proceso muy pequeñas trabajaran juntas unas con otras, en paralelo y así hacer emerger resultados que se antojarían sorprendentes.

Hillis había trabajado para algunas empresas que hacían juguetes electrónicos y entendía de chips, de circuitos integrados y comprendía lo que se podía hacer en términos de electrónica. Así, concibió lo que más adelante se llamaría la máquina de conexiones (la “connection machine”, del original en inglés). Esta máquina, una supercomputadora en paralelo, se diseñó en 1983. Dos años después Hillis obtuvo su doctorado por el MIT bajo el siguiente jurado de extraordinarios especialistas: Gerald J. Sussman, Marvin Minsky y Claude Shannon. Su tesis obtuvo el reconocimiento *Distinguished Dissertation ACM* (1985). Cabe decir que Holland había concebido unos 20 años antes esta idea de la computadora en paralelo, pero claramente la tecnología no tenía el avance suficiente. Cuando Hillis terminó el diseño de su máquina, le presentó a Holland una copia de la misma con la nota: “Pienso en esto como si se tratara de la máquina de Holland”.

El procesamiento en paralelo era desde luego, un tema recurrente en el laboratorio de Inteligencia Artificial del MIT y algunos fabricantes habían ya experimentado con dos, cuatro o más procesadores, que trabajaran en paralelo, pero nadie había siquiera considerado las ideas de Hillis que contemplaban en su diseño miles de procesadores, unos 16,000 para empezar, aunque el diseño buscaba cuadruplicar esta cantidad. Una idea extravagante y aparentemente fuera

de la realidad. Sin embargo, Marvin Minsky apoyó a Hillis y logró un apoyo de un millón de dólares, aunque a la postre el proyecto costó 5 millones de dólares.

Con el tiempo Hillis se dio cuenta que su proyecto necesitaba ser escalado y consiguió el apoyo de inversionistas. Entonces nació la empresa *Thinking Machines*, la cual tuvo un gran éxito commercial y en donde Hillis pudo concretar su diseño con una máquina con 65536 procesadores. Es importante señalar que estos procesadores trabajaban solamente con un bit cada uno, pero evidentemente el enfoque era hacer trabajos simples que eventualmente emergieran en algún resultado complejo.

Pero ¿cómo era la supercomputadora de Hillis, la Connection Machine? De acuerdo a la misma descripción del autor del diseño, se trataba de un arreglo hipercúbico de paralelismo masivo, con miles de procesadores (65,536 en total) con 4Kbits de memoria RAM cada uno de ellos. Los procesadores solamente podían procesar un bit a la vez, porque lo que se infiere que las operaciones que se podían hacer en este sistema eran muy rápidas pero muy elementales. El tamaño de estas máquina era de 1.5 metros cúbicos, dividido en ocho cubos más pequeños.

La Connection Machine usaba LISP como lenguaje de programación, el cual usa el paradigma funcional y es diferente al que se usa originalmente en los sistemas con Pascal, C, Java, etcétera. No obstante, muchas de las aplicaciones se escribieron en C* (*star C*), un súperconjunto de instrucciones que podían manejar paralelismo en el lenguaje C.

La apuesta de Danny Hillis era simular muchos procesos en paralelo, usando reglas simples en los algoritmos de evolución. Él esperaba hallar comportamientos emergentes, los cuales serían sin duda un hito en el trabajo de vida artificial. Hillis contaba con una ventaja: podía simular cientos de miles de generaciones y

además, podía ver los restos, los “fósiles” -valga la expresión- de ejecutar sus programas muchas, muchísimas veces.

El científico empezó con “organismos artificiales”, cadenas de números que representaban genes, los cuales expresaban un fenotipo específico cuando se ejecutaban los algoritmos en los programas. De alguna manera todo esto era lo que Holland habría querido hacer. Al principio se pusieron problemas simples, por ejemplo, ordenar los genes numéricamente de menor a mayor. Había criterios de supervivencia y se analizaban los cambios con cada generación.

Los experimentos de Hillis probaron que con el paso de las generaciones, los organismos “genéticos” más aptos, lograban cumplir con su meta y llegar al final del camino. Otros eran eliminados. Un proceso simple de evolución darwiniana que funcionaba perfectamente bien.

Un biólogo, Charles Taylor, vio el sistema y propuso usar reproducción, sexual y asexual.⁴¹ Esto puso a pensar a Hillis ¿por qué hay sexo? ¿por qué hay apareamiento? En las nuevas generaciones creadas por reproducción sexual se diluye el número de genes y se combinan, por lo que tal vez el mejor mecanismo debiese ser el de la reproducción asexual. Sin embargo, después de ejecutar el algoritmo genético por decenas de miles de generaciones, Hillis encontró que la reproducción sexual tenía mejor supervivencia y por ende, el ciego mecanismo de la evolución, que no tiene consciencia en su actuar, actuaba localmente pero tenía consecuencias a nivel global.

Esta es una consecuencia extraordinaria, pues explicaría el éxito de la reproducción sexual en el mundo real. Sin embargo, Hillis tenía otras problemáticas en mente. La supervivencia de sus organismos en su multimillonaria máquina, significaba que resolvían, a manera de analogía, el problema de subir

⁴¹ La reproducción sexual crea un nuevo organismo a partir de la combinación de los genes de dos organismos de una misma especie. En cambio, en la reproducción asexual, un organismo adulto es capaz de desprender una sola célula del mismo y así formar otro organismo idéntico a él.

una montaña. De manera simple esto es equivalente a subir una montaña hasta llegar a una meseta, que bien podríamos considerar el punto más alto. Pero resulta que este punto puede ser un máximo local, es decir, hay mesetas más arriba aún, a mayor altura. Hillis se preguntaba ¿por qué en el mundo real los animales buscaban convertirse cada vez en más aptos? ¿Por qué no se conformaban con llegar a un máximo local? El problema no era simple, pero Hillis siguió con diferentes tipos de organismos que se combinaban sexualmente y producían seres más aptos. Sin embargo, después de correr por miles de generaciones su simulación, halló algo que puede considerarse asombroso. Había organismos que no se apareaban sexualmente, pero que se combinaban con otros organismos tomando parte del código genético del anfitrión para sobrevivir. Vamos, ¡que Hillis había hallado el equivalente a los parásitos! Efectivamente había organismos que cómodamente usaban los genes de su anfitrión. Pero entonces el investigador decidió correr unas miles de veces más su simulación, ya con estos parásitos en el sistema y de pronto pasó algo aún más notable: los parásitos habían desaparecido y los organismos supervivientes eran más aptos que sus antecesores.

La conclusión de todo esto es aún más sorprendente y explica en cierta manera por qué los organismos reales buscan siempre ser más aptos (y no se conforman con máximos locales): los parásitos son necesarios en la vida para que los organismos decidan eventualmente deshacerse de ellos y así propiciar ser más aptos cada vez.

Los experimentos de Hillis son extraordinarios y hablan de que la evolución puede ser considerada sí, la supervivencia del más apto, pero más aún, habla de los mecanismos que provocan que esto ocurra y lleva a la conclusión de que Dios, si existe, bien podría considerarse darwinista.

Las colonias de hormigas de la UCLA

Un concepto importante en vida artificial e inteligencia artificial es el del algoritmo genético (AG). AG emplea el método análogo al proceso de la evolución natural para producir generaciones sucesivas de entidades de software que incrementan su supervivencia como uno de sus propósitos.

JACK COPELAND (*The Essential Turing*)

David Jefferson, a principios de los años ochenta del siglo pasado, era un científico de cómputo en la Universidad del sur de California. Su interés por la vida artificial parece haber despertado después de haber leído “El Gen Egoísta”, de Dawkins. Esto lo hizo interesarse notablemente en la evolución y los mecanismos de la información. Así entonces, decidió trabajar en un programa que simulara la vida y creó su propio mundo virtual al que llamó “programinales”.

Sus primeros experimentos fueron trabajar sobre el clásico problema depredador-presa. Los resultados iniciales fueron muy malos. Su simulación de lobos *versus* conejos no lo llevaba a la solución clásica de las ecuaciones diferenciales. En lugar de esto, veía cómo se extinguían los conejos y poco tiempo después, los lobos, que morían de inanición. Jefferson no entendía muy bien qué pasaba, hasta que leyó un artículo de Robert May, que hablaba de la inestabilidad de los ecosistemas. Ahí Jefferson se dio cuenta que la solución a sus problemas era tratar con poblaciones mucho más grandes y entonces encontraba la solución clásica, la oscilación en las poblaciones de una y otra especie.

Habiendo logrado de alguna manera simular el comportamiento del problema mencionado, Jefferson decidió crear dos organismos que eventualmente tuviesen las mismas habilidades. ¿Qué pasaría entonces? Para su sorpresa, no obtuvo la esperada oscilación, sino que una especie acabó con la otra. La exterminó. Analizando los resultados, el programador se dio cuenta de algo: la especie ganadora había tenido una mutación azarosa (programada por el propio Jefferson, sin saber él mismo cuando las mutaciones podrían ocurrir), y aunque la mutación debilitaba a la especie, impidiendo por ejemplo, el movimiento de sus extremidades, esto resultó beneficioso porque esta especie gastaba menos energía, pues estaba quieta. Eso obligaba a la otra especie a acercarse y entonces en el ataque fracasaban pues estaban minados energéticamente.

Puede pues decirse que sus simulaciones no necesariamente son exactas al mundo real y segundo, que Jefferson tenía que dar más información biológica a sus organismos, lo cual era un problema pues no estaba entrenado como un profesional en esa área. Las dificultades se resolvieron cuando Jefferson conoció a Charles Taylor, un biólogo especialista en poblaciones, quien en realidad estaba más interesado en la génesis de la consciencia. ¿Cómo era posible que meras transformaciones químico-eléctricas en el cerebro de pronto dieran con esta posibilidad que nos hace darnos cuenta de nosotros mismos? Taylor aprendió de máquinas de Turing e Inteligencia Artificial. Obtuvo una plaza en California y aprendió fascinado el algoritmo genético de Holland. Jefferson le dio una plataforma de cómputo y le hizo ver que la vida artificial podía ser el escaparate correcto para modelar miles de miles de características e interacciones entre ellas, que podrían finalmente emerger en lo que llamamos consciencia.

Jefferson y Taylor fundaron en la universidad un nuevo departamento: el de Ciencias Cognitivas. Con la ayuda de Taylor, los estudiantes graduados hicieron que los sistemas de Jefferson se transformaran en una nueva arma para el desarrollo de simulaciones biológicas, capaces de modelar el comportamiento de

las poblaciones y de la evolución, con detalle nunca antes logrado por ningún otro programa. A este gran sistema le llamaron RAM.

El software era un medio ambiente para la construcción de cadenas de bits en LISP⁴² que representaban a los animales u organismos. En cada *tick del reloj*, los organismos en RAM podían ejecutar alguna de las siguientes tareas:

- Examinar la vecindad del entorno.
- De acuerdo a la edad del organismo y su estado (incluyendo la historia genética pasada), decidir quizás por probabilidad o azar, qué acción tomar.
- Tomar acción, incluyendo alguna de las siguientes: actualizar estadísticas, actualizar su propia memoria, reproducirse, modificar el entorno, moverse o incluso, morir.

Pronto la Universidad de California, en Los Angeles (UCLA) se convirtió en la capital de los estudios sobre vida artificial. Otros investigadores se unieron a los trabajos de Jefferson y Taylor. Uno de ellos fue Michael Dyer, el jefe del Departamento de Inteligencia Artificial. La efervescencia de estar frente quizás una nueva ciencia logró apoyos monetarios, como el de un millón doscientos mil dólares, para hacerse de una *Connection Machine* (Hillis), que tenía 16,384 procesadores. Esto llevó a la siguiente generación en el trabajo del departamento de Taylor y Jefferson. A esto le llamaron el proyecto “Genesys”.

El sistema Genesys se planteó como un programa mucho más ambicioso para ver la evolución bajo la vida artificial. Los creadores, David Jefferson, Robert Collins, Claus Cooper, Michael Dyer, Margot Flowers, Richard Korf, Charles Taylor y Alan Wang buscaban la posibilidad de probar comportamientos complejos a partir del uso de la evolución. Así, Genesys toma una población de programas que están

⁴² LISP significa en inglés “LISt Processing” (Procesamiento de LIStas). La lista es la estructura más importante del lenguaje y los programas de LISP pueden verse como la manipulación de código fuente que bien puede entenderse como una lista también. Así pues su definición es recursiva.

simplemente codificados como una cadena de bits, los cuales se desenvuelven usando el algoritmo genético de Holland (con la intención de optimizar el comportamiento en algunas tareas específicas).

Lo que buscaban resolver, estrictamente, este grupo de investigadores era si la evolución podía ser usada para crear programas, es decir, el texto de un programa (en lugar de sus parámetros), podía ser el material en un algoritmo genético. Y si esto es así, ¿de qué tamaño tiene que ser la población? ¿Cuál debería ser la velocidad de las mutaciones, de la recombinación, de la selección de parámetros, de la estrategia para reproducirse?

Pero además, se buscaba entender ¿qué representación para los programas era la más apropiada para el uso del algoritmo de Holland?, en particular, ¿son las redes neuronales capaces de ser diseñadas para trabajar con el algoritmo genético? Y si estas preguntas parecen complejas, hay más: ¿Pueden los problemas de la biología -en la teoría de la evolución- ser atacados por algoritmos genéticos sobre poblaciones de animales artificiales?

Teniendo a la mano una computadora como la Connection Machine, era claro que el experimentar con creaturas originales y probar las teorías evolutivas era muy tentador. Uno de sus experimentos más notables fue lo que llamaron Tracker,⁴³ una “hormiga” que tenía que recorrer un camino irregular, llamado “el camino de John Muir” y que pasadas algunas generaciones de creaturas, lograba seguirlo perfectamente. Cuando esto ocurrió, ya Chris Langton (el creador de la hormiga caótica), tenía en mente un nuevo experimento, el cual llamó “el recorrido de Santa Fe”, en donde retaba al grupo de hormigas que se basaban en el algoritmo genético para llegar a emerger propiedades con el paso de las generaciones, a recorrerlo correctamente. Curiosamente las hormigas que ya habían hecho el recorrido de Muir no fueron muy buenas en este nuevo camino. Pero una nueva serie de “hormigas bebés”, que empezaban sin ningún conocimiento previo,

⁴³ Véase <http://www.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html>

mostraron que podían con el reto. ¿Por qué las otras hormigas, que estaban especializadas en Muir no podían hacerlo tan bien? Los investigadores llegaron a la conclusión que estas hormigas estaban hiper-especializadas y eso trabajaba en su contra. De acuerdo a las explicaciones de Jefferson, “Tomaban [las hormigas bebés] cualquier heurística, cualquier atajo, cualquier cosa que pudiese usarse en el camino” y entonces el investigador piensa que una situación similar pudo haber ocurrido cuando evolucionaron los reptiles para vivir bajo el agua. “Quizás hubiese sido mejor empezar con los seres anfibios”. Para Jefferson esto es un ejemplo de macro-evolución en la todos creían pero que nadie había podido demostrar “hasta ahora”, añade Jefferson.

Los diferentes experimentos con estas “colonias de creaturas/hormigas” mostraron sin embargo que la parte más interesante era el comportamiento social que observaban dichos organismos. Una colonia de hormigas puede verse muchas veces como una especie de súper-organismo, algo global, en donde las hormigas participan como partes del mismo más que como si fuesen entes individuales. Esto quiere decir que actúan de forma cooperativa. Se halló que las hormigas dividían sus trabajos, se volvían especialistas, buscaban metas comunes y se adaptaban a las circunstancias de su entorno. Vamos, se comportaban como una verdadera colonia de hormigas. Lo curioso de todo esto quizás ya lo ha adivinado el lector: no había un controlador central, nadie que distribuyese las tareas al inicio de la sesión. Así pues, las hormigas actuaban por sí solas en sus propias tareas, emergiendo su comportamiento individual a uno global coordinado. Esto sin dudas es algo fascinante.

La “granja de hormigas” fue un experimento muy complejo en términos reales. Se estaban simulando 16,384 colonias de hormigas, cada una consistiendo de 128 miembros para hacer un gran total de 2,097,152 hormigas. Las hormigas tenían cada una 25,590 genomas, que comparados con los 450 genomas de Tracker, las hacían mucho más complejas. Es importante decir que cada hormiga tenía el mismo genoma que todas las demás, pero al interactuar con el entorno, este

genoma cambiaba y entonces cada hormiga generaba su propio comportamiento individual y único. El programa podía manejar los genomas más largos y manejar información del entorno de forma que tuviesen datos sobre dónde había comida y del trazo de “feromonas” que dejaban sus congéneres. Con ello, las hormigas podían acumular comida, alimentarse, dejar rastros de feromonas e incluso usar una especie de brújula mental para poder regresar a casa.

Hubo también cambios en la manera en como el AG elegía a las hormigas que podían aparearse. Jefferson y Collins, particularmente no estaban contentos en como el AG hacía esta labor y quizás fue uno de los cambios más importantes. Así, en lugar de aparearse de manera azarosa, se modeló una idea más sofisticada: los organismos caminaban azarosamente en una vecindad de su nido e inevitablemente hallaban otras colonias en donde podía haber más comida. Después de cada caminata, las hormigas elegían los nidos con mayor éxito y quienes lo lograban entonces se apareaban. Esto, a decir de los investigadores, era más cercano a como la Naturaleza actuaba.

No obstante, la granja de hormigas de la UCLA no mostró ningún comportamiento emergente directamente de la cooperación entre los individuos de un nido. Después de un año de simulaciones, nunca pudieron observar un comportamiento en donde las hormigas dejaran un rastro (feromonas), para que otras hormigas supieran, por ejemplo, dónde hallar comida.

Hubo explicaciones para esto, pero decididamente algo limitaba el comportamiento de las simulaciones de la granja de la UCLA. Sin embargo, los resultados mostraron comportamientos emergentes muy interesantes que explican cómo evolucionan muchos organismos reales. La analogía de esta granja de vida artificial contra la vida real, era en ocasiones asombrosa.

Capítulo XV

Los polimundos de Larry Yaeger

La evolución es un tipo de topología: una en donde el que sobrevive persiste, una en donde la reproducción incrementa su número, las cosas cambian. Es encantador para mí que las cosas sean así de simples y elegantes, es sorprendente... Es magia... No necesito invocar ningún mito de la creación.

Larry Stephen Yaeger⁴⁴

La idea de la vida artificial parece haber permeado en muchos investigadores y estos han decidido trabajar sobre los diferentes conceptos que pueden analizarse a través de simular el fenómeno que llamamos vida, en el cual concurren una serie de factores: comida, sexo, seguridad, supervivencia del más apto, evolución, etcétera.

Uno de los sistemas más desarrollados para este tipo de investigaciones de vida artificial se debe a Larry Stephen Yaeger, el cual es un científico distinguido de la empresa Apple, además de ser profesor en la Universidad de Indiana en Bloomington. Actualmente Yaeger trabaja para Google. Cabe destacar que anteriormente a su trabajo en vida artificial, Yaeger tiene en su haber la creación

⁴⁴ De una larga entrevista que puede leerse en <http://matejhorvat.si/en/newton/larry/>.

del software de reconocimiento de escritura manuscrita para el dispositivo Newton de Apple, que desafortunadamente no tuvo éxito.

Pero en el campo de la vida artificial, Yaeger es conocido por su programa *Polyworld*. Este programa ya está en las plataformas Linux y Mac OS X, y permite hacer experimentos en este tema a través de conceptos programados que van de la selección natural hasta los algoritmos evolucionados. El algoritmo genético está incluido en ellos, desde luego.

Para presentar los resultados en la pantalla se usan dos ambientes gráficos Qt graphic toolkit y OpenGL. Pueden así verse en el entorno gráfico a poblaciones de trapezoides buscando comida, tratando de aparearse o incluso buscando cazarse. Es importante señalar que las poblaciones no rebasan el ciento de elementos, pues cada individuo generado es muy complejo y el medio ambiente consume muchos recursos de la computadora.

Tal vez lo más interesante y no deja de asombrarnos, es el hecho de que ciertos comportamientos han aparecido espontáneamente después de una prolongada evolución, es decir, después de que el programa ha corrido muchos miles de generaciones de individuos. Se han hallado comportamientos como el canibalismo o bien, el mecanismo ampliamente estudiado de *depredador-presa* e incluso mimetismo.

Yaeger puede programar en sus PolyWorlds a cada individuo, el cual toma decisiones basados en una red neuronal usando *aprendizaje hebbiano*⁴⁵. Así pues, la red neuronal se deriva de cada genoma que tiene cada individuo. Es importante decir que en este caso, el genoma no solamente especifica el cableado y pesos en las redes neuronales, sino que además, determina su color, tamaño, razón de mutación, entre otros factores. Yaeger hace que el genoma mute de

⁴⁵ Aunque Donald Hebb habla de células, en el caso que nos ocupa podríamos decir que "la persistencia de una actividad repetitiva tiende a inducir cambios celulares de largo plazo que promueven su estabilidad"

manera azarosa como un conjunto de probabilidad, el cual cambia [el genoma original] en los organismos descendientes de estos.

De acuerdo con el sitio web de Yaeger (ver bibliografía), Polyworld es ecología computacional, desarrollada para explorar fenómeno en la vida artificial. Los organismos simulados se reproducen sexualmente, luchan y se matan entre ellos, comen comida que crece en el mundo simulado y de alguna manera desarrollan estrategias (exitosas en ocasiones), que les permiten sobrevivir. En caso contrario, simplemente mueren. El comportamiento de cada organismo –dice Yaeger– cómo se mueve, da vueltas, ataca, come, se reproduce, etcétera, es controlado por la red neuronal a la cual el autor llama “cerebro”. Cada uno de ellos está determinado por su código genético (tamaño, número y composición del *cluster* neuronal (que puede ser excitatorio o inhibitorio) y de los tipos de conexiones que puede haber entre los clusters.

Yaeger indica que Polyworlds bien puede llevar a entender los problemas de optimización y de la dinámica de la selección natural. Una forma en donde se combinan estas ideas es precisamente en la visualización gráfica que da su programa. Una de las metas de Polyworlds fue el probar que había comportamientos emergentes a partir de un conjunto de comportamientos primitivos construidos en los organismos del programa, sus mecanismos para sensor acciones y la selección natural de sus sistemas neuronales. Estas estrategias de comportamiento son reconocibles en los organismos vivos y son puramente emergentes en el entorno de los Polyworlds. Esto es una muy interesante apreciación de Yaeger. Es decir, parece ser que el modelo de la evolución puede plantearse casi como un juego de computadora. Así entonces, de comportamientos primitivos que responden a presiones simples del entorno ecológico planteado, es posible quitarle un poco al misterio de la evolución en los organismos naturales.

Yaeger piensa que es más fácil contemplar y entender esos mecanismos en organismos simulados que en los que se ven en el mundo real, precisamente por ser simulados. Y el autor hace una acotación importante: “la bendición y maldición de la vida artificial es que es mucho más difícil para los humanos antropomorfizar (¿zoomorfizar, biomorfizar?), estos organismos en una máquina que en un organismo natural”. Sin embargo, esto parece tener una ventaja, que es el poder liberar la interpretación de los fenómenos de los prejuicios que tenemos comúnmente.

Y surge entonces la pregunta importante: ¿Están vivos los organismos de la vida artificial? De nuevo Yaeger dice que no podemos saberlo porque aún no tenemos una definición de vida. ¿Qué es la vida? Y abunda: Como dicen Farmer y Belin,⁴⁶ “si viajamos a otro planeta, ¿cómo sabremos si la vida está presente o no?” y haciendo esta analogía, Larry Yaeger se pregunta entonces: “Si *viamos* a un mundo artificial, ¿cómo podemos saber si hay o no vida presente?”. Farmer y Belin, buscando contestar esta interrogante, ofrecen un conjunto de propiedades que finalmente, están asociadas con la vida. Yaeger entonces las contrasta con su Polyworlds y afirma que los siguientes criterios de los investigadores mencionados se cumplen:

- *La vida es un patrón en el espacio tiempo* más que un objeto material específico. Esto significa que hasta un ser vivo es realmente un proceso (que persiste), en lugar de pensar en lo que define al individuo (lo cual no persiste). Con esto en mente, los organismos son patrones en la computadora dentro de Polyworlds más que un sustrato.
- *Auto-reproducción*. Hablamos que en los polimundos hay reproducción de acuerdo al contexto de este entorno. Hay involucrados ciertas reglas que

⁴⁶ Farmer, J. D., and A. d'A. Belin (1992), **Artificial Life: The Coming Evolution**, en Artificial Life II, editado por C. Langton, C. Taylor, J. Farmer, y S. Rasmussen. Santa Fe Institute Studies in the Sciences of Complexity Proc. Vol. X. Addison-Wesley, Redwood City, CA, 1992.

permiten la reproducción y por ende, bien podría argumentarse que no hay auto-reproducción aquí, sino que es guiada.

- *La información guarda una auto-representación (de sí mismo).* Dentro del contexto de los Polyworlds, los organismos ahí presentes claramente tienen una auto-representación de ellos mismos. No es una propiedad emergente, sino que se programó de antemano, pero en sentido estricto, se cumple con el criterio de Farmer y Belin
- *Metabolismo.* En Polyworlds los organismos efectivamente convierten comida hallada en el entorno en energía que usan para llevar a cabo sus procesos internos y sus actividades de comportamiento, como en el caso de los organismos naturales.
- *Interacción funcional con el entorno.* Claramente puede verse que en Polyworlds los individuos interactúan con el medio ambiente. Aparte de comer y gastar la energía, tienen interacción con otros individuos en el entorno.
- *Interdependencia de partes.* Los organismos en los Polyworlds pueden morir si se les separa de sus fuentes de energía. El dividir un cerebro en dos, por ejemplo, no producirá dos organismos que se comporten como el original. Se da por un hecho que los mecanismos emergentes más sofisticados nacen a partir de las arquitecturas de redes neuronales implementadas.
- *Estabilidad bajo perturbaciones.* Los organismos en los polimundos pueden sobrevivir a pequeños cambios en su medio ambiente.
- *Capacidad para evolucionar.* Los organismos en los polyworlds pueden y claramente evolucionan. Sin duda están limitados en su evolución, por ejemplo, no pueden tener el sentido del olfato a menos que se le programe externamente. Pero en el mundo real también los organismos están limitados en su evolución. Y de hecho, todos los seres vivos estamos limitados por las leyes de la propia física. Finalmente no podemos evolucionar, en la dirección que sea, que permita movernos a la velocidad de la luz.

Por otra parte, de acuerdo a Dawkins, la palabra “viviente” no necesariamente tiene que significar algo real. Para él, los sistemas que denominamos “vivos” emergieron de una serie acumulativa de procesos que favorecieron a los seres que se pudieron replicarse en mayor grado. Es decir, hablamos de los replicadores que tuvieron la habilidad de incluir en sí mismos una “máquina para sobrevivir”. Esto parece sugerir, de acuerdo a Dawkins, una continuidad entre lo vivo y lo no-vivo. No parece pues poderse definir una frontera clara entre lo inerte y lo que está vivo.

Farmer y Belin indican, por ejemplo, que hay cada vez más consenso entre los biólogos sobre el asunto de la continuidad. Parece ser que es más apropiado considerar *la vida como una propiedad continua de un patrón que está organizado*, pero Farmer y Belin añaden algo a esto, “que está más o menos vivo que otros”, lo cual de alguna forma nos deja perplejos: ¿Es que hay algo con más vida que otro? Fuera de este extraño argumento, es claro que podemos pensar en la vida, no como una cualidad absoluta que de pronto apareció, sino que gradualmente fue emergiendo al inicio de la evolución y de hecho, por eso estamos ante la imposibilidad de saber qué es vida. Se presume que la vida emergió entonces en un proceso largo, gradual, donde los proto-organismos se crearon poco a poco en la sopa primigenia de los mares. Por ello mismo, la vida no está completa, es un fenómeno indefinido en parte. No está totalmente determinado por el estado del ser”.

Por otra parte tenemos la opinión del famoso físico Erwin Schrödinger, quien propuso en 1944 una definición termodinámica de la vida, en donde gracias al alimento, el beber, el respirar y asimilar los nutrientes, definimos la vida como simplemente metabolismo. Farmer y Belin aceptan al metabolismo como una parte de lo que debe contener algo vivo, pero queda claro que sólo el metabolismo no parece ser suficiente.

Mientras llegamos a un acuerdo en lo que es la vida, es claro que los programas que simulan vida buscarían convencernos que estamos hablando de vida creada, a partir de una simulación en un entorno virtual. ¿Pero es esto suficiente? Da la impresión que se necesita algo más. Un escenario hipotético plantearía una situación como esta: imaginen que de pronto un programa toma el control de la computadora y además, se comunica con el usuario, el ser humano que la maneja. En ese momento, si la computadora “cobra vida propia” estaríamos ante un fenómeno innegable de vida, aunque quiero creer que en ese momento la reacción más elemental sería desconectar la computadora de la corriente.

Pareciera que estamos de nuevo en un callejón sin salida. Si una máquina fuese absolutamente independiente de nuestras decisiones ¿podría llamarse viva? ¿O necesita que tenga, por ejemplo, consciencia de sí mismo? ¿Esta característica es necesaria? ¿Los animales tienen consciencia de sí mismos, de su existencia? ¿Y los insectos? ¿Y los organismos unicelulares? Parece que estamos atrapados entre nuestras propias definiciones y no podemos resolver el enigma a la fecha.

Pero más allá de estas especulaciones, pensemos ¿por qué no podemos definir la vida? Cualquier niño de cinco años sabe distinguir entre lo vivo y lo muerto. ¿Por qué la definición de vida se escurre entre las manos como si fuese agua?

Capítulo XVI

Jugando a ser Dios

*No es el más fuerte o el más inteligente quien sobrevive,
sino quien puede lidiar mejor con los cambios.*

Charles Darwin

Imaginemos que somos Dios y que tenemos ya un entorno, un medio ambiente, en donde los seres vivos –que ya hemos creado– pueden desarrollarse, reproducirse, evolucionar y en algún momento, lidiar con las dificultades inherentes del propio entorno. Tal vez incluso tengan que eliminar a sus enemigos. ¿Cómo serían las cosas en un ambiente así? Imaginemos que dotamos a nuestros seres de estas posibilidades, de estos mecanismos de supervivencia y observamos cómo se desarrollan la vida de estos seres. Sería como jugar a ser Dios.

Quizás no necesariamente con esta idea, en agosto de 1961 Victor A. Vyssotsky,⁴⁷ M. Douglas McIlroy y Robert Morris Sr. inventaron un juego en Bell Labs (ahora perteneciente a AT&T), al cual denominaron “Darwin”, el cual corría en una máquina IBM 7090. Cabe decir que en ese tiempo las computadoras eran más una excepción que la regla y el costo del tiempo de máquina era prohibitivo en

⁴⁷ Vyssotsky parece ser el inventor del juego, de acuerdo con <http://www.cs.dartmouth.edu/~doug/darwin.pdf>

muchos casos. Sin embargo, Darwin se jugaba en las noches, cuando ya la máquina no tenía carga de trabajo.

El juego consistía en un programa, llamado “the umpire” -el árbitro- el cual apartaba un área de memoria que se denominaba “arena”. En la arena podían competir dos o más programas, escritos por los jugadores, los cuales eran cargados a la memoria de la máquina. Estos programas debían ser escritos en código de máquina de la IBM 7090 y podían llamar a una serie de funciones provistas por el árbitro para poder, por ejemplo, sondear lo que había en otras localidades de memoria de la arena, matar a los programas oponentes y reclamar la memoria vacante para hacer copias de sí mismos. El juego terminaba cuando ocurrían dos posibilidades: o se acababa el tiempo para jugar o cuando la copia de algún programa se mantenía con vida. El jugador que escribiese el programa sobreviviente era declarado el ganador.

Hasta 20 localidades de memoria, de cada programa, podían designarse como protegidas, de manera que si un programa sondeaba alguna localidad de memoria protegida, el árbitro transfería el control al programa que había sido sondeado. Éste se ejecutaba hasta que llegase a sondear un área protegida de otro programa y entonces se transfería el control al siguiente programa, y así sucesivamente.

Es interesante el hecho de que los programas podían copiarse y relocarse a ellos mismos dentro de la memoria, pero tenían prohibido alterar localidades de memoria fuera del rango permitido sin el permiso del árbitro. En la medida que los programas eran ejecutados directamente en la computadora, no había mecanismo físico que impidiera hacer trampa. Sin embargo, el código fuente de los programas en pugna estaba disponible para que lo estudiaran los otros y así se auto-controlaba la posibilidad de hacer trampa. Además, esto permitía aprender de los rivales.

Puede verse que el juego por sí mismo no es interactivo, es decir, los jugadores no pueden cambiar el comportamiento de sus programas cuando estos ya están corriendo. A lo más, sólo pueden ver las acciones que ocurren en la arena. Darwin así proveía los mecanismos de replicación y búsqueda de supervivencia, aplicando reglas muy sencillas en el sistema. Lo interesante en todo caso es que la inteligencia de los programadores se ponía en tela de juicio en un ambiente controlado, con reglas simples y precisas, las cuales llevarían al mejor programa a sobrevivir a los demás.

El juego tuvo una buena acogida y el programa más pequeño que podía reproducirse, localizar a los enemigos y matarlos, consistía de unas 30 instrucciones. Douglas McIlroy desarrolló un código de tan solo 15 instrucciones que era capaz de localizar y matar a sus enemigos, pero no era capaz de reproducirse. Sin duda no era el programa más letal de la historia, pero al menos parecía imposible de ser eliminado. Era además más pequeño que el área de 20 instrucciones protegidas. Debido a esto, en versiones posteriores del software el área de localidades protegidas fue disminuido.

Pero curiosamente, el programa más letal y que se definió como invencible, fue desarrollado por Robert Morris Sr. Éste tenía 44 instrucciones y empleaba una estrategia adaptable. Una vez que se localizaba el inicio del programa enemigo, el software sondearía a una pequeña distancia por encima de su posición. Si tenía éxito y podía matar al enemigo, recordaría esta distancia, la cual se usaría en encuentros sucesivos. Si se trataba de una localidad de memoria protegida, la siguiente ocasión que el sistema tuviese el control, elegiría una distancia diferente. Todas las nuevas copias generadas se inicializaban con un valor exitoso y de esta manera, el programa de Morris evolucionaba en muchas especies, cada una adaptada específicamente a un enemigo particular.

De acuerdo a los autores del juego, el mínimo número de instrucciones para lograr sondear, matar y reclamar la memoria era de 30. Las secuencias de las llamadas

eran muy eficientes en donde la mayoría del código se dedicaba a sondear y reproducirse. Este último paso, sin embargo, no era una tarea trivial, pues el código de un individuo no se podía simplemente copiar, sino que tenía que relocarse adecuadamente, Vyssotsky inventó un ciclo de mover y relocalizar, de cinco instrucciones, que instantáneamente se volvió un estándar en el juego. El programa, al ejecutarlo, usaba unas 10,000 localidades de memoria. El juego corría rápidamente en menos de un minuto pues los programas no usaban un intérprete de instrucciones, sino que era simplemente el nativo de la máquina de IBM.

Desde luego que las acciones implementadas en Darwin son meramente los conceptos fundamentales de reproducción, evolución, eliminación de los oponentes y sobrevivencia. En el mundo real estos mecanismos pueden ser por demás complejos, pero la esencia de ellos está claramente presente en el juego Darwin. Y aunque el juego no pasó de ser un divertimento de un par de semanas, pues Morris halló la manera de hacer un código que fuese invencible, esto dio pie a programas similares muchos años después, con la misma idea, en donde se aglutinan en algo llamado *Core War* (Guerras de núcleos)⁴⁸.

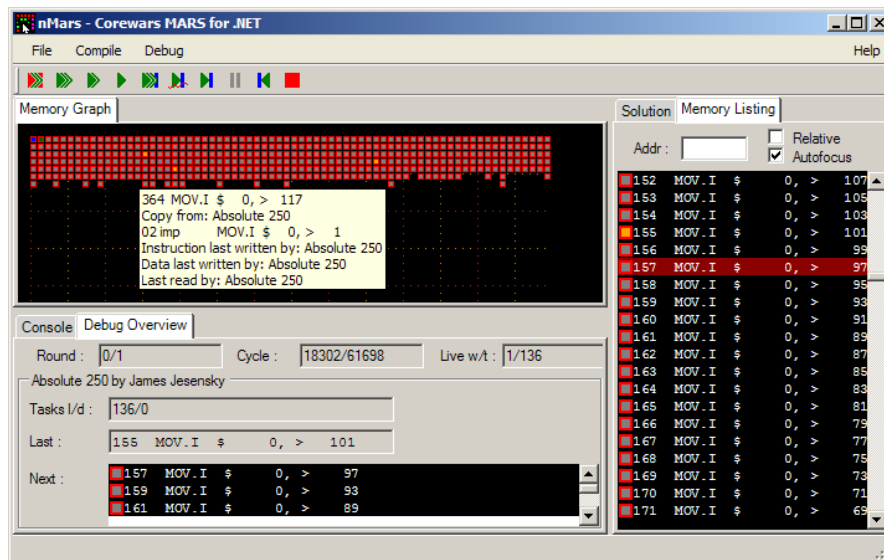
Basándose en Darwin, *Core War* se definió como un juego de programación de computadoras, diseñado por G. Jones y el programador y divulgador A.K. Dewdney, en donde dos o más programas luchan (llamados “guerreros”) por el control de una computadora virtual. Estos programas deben escribirse en un lenguaje ensamblador (que es el anterior al código de máquina puro), de una procesador inexistente llamado *RedCode*.

Al inicio del juego, los programas guerreros se cargan en memoria, en segmentos al azar, como si fuesen programas independientes unos de los otros, y entonces

⁴⁸ En un inicio, la memoria de las computadoras era una malla de pequeños aros de ferrita, que al paso de la corriente, se magnetizaban en una de dos formas. A eso le llaman núcleos. El nombre del juego mantiene en el nombre el hecho de que se trata de programas muy primitivos, que corren en un lenguaje por demás simple.

se genera un procedimiento de multitareas, en donde un sistema administrador (algo así como el árbitro en Darwin), realiza una sola instrucción por programa por turno. El objetivo es que los programas de los rivales terminen, se detengan. Esto suele pasar cuando quieren ejecutar una instrucción inválida, por ejemplo, dejando al ganador la posesión completa de la máquina virtual.⁴⁹

A.K. Dewdney describió la primera versión de RedCode, pero ésta difiere en muchos aspectos con respecto a los últimos estándares de la *International Core War Society*. Sin embargo esto no altera fundamentalmente la idea del juego y basta con leer la especificación del RedCode en boga para poder participar en estos torneos de Core Wars.



MARS – Un intérprete de código abierto para jugar Core War

La estrategia del juego se divide básicamente en tres elementos que tienen su analogía en el juego infantil de “piedra, papel y tijeras”.

- Papel (o replicador): Un replicador hace copias repetidas de él mismo y se ejecutan en paralelo, eventualmente llenando todo el núcleo con copias de

⁴⁹ Existen muchos entornos para jugar Core War, por ejemplo <http://nmars.sourceforge.net/>, el cual contiene código fuente incluso para la plataforma .NET.

su propio código. Se sabe que los replicadores son difíciles de eliminar, pero también tienen dificultades para matar a sus enemigos.

- Tijeras (o escáner): Un escáner está diseñado para batir a los replicadores. El escáner no ataca ciegamente. Trata de localizar los enemigos antes de lanzar su ataque.
- Piedra (o bombero): Un bombero, en el sentido de manipular artefactos que explotan, bombas pues, copia ciegamente una bomba en intervalos regulares en el núcleo (la arena), esperando golpear al enemigo.

Todas estas estrategias se programan en RedCode, el cual es el lenguaje de programación que se usa en Core War. Como este lenguaje no pertenece a ningún procesador real, se escribió una máquina virtual llamada *Memory Array Redcode Simulator* (MARS). El diseño de RedCode se asemeja al lenguaje de máquina de cualquier computadora CISC (1980) pero contiene características que no están presentes en ninguna arquitectura de computadoras real. MARS funciona pues como el entorno donde los programas guerreros luchan por el triunfo y apoderarse del control de la máquina. Esta es la trama de la película TRON, de hecho.

El desarrollo de Core War se inspiró no en Darwin, dicen algunos, sino en Creeper, que creaba programas inútiles en memoria, y en un programa posterior, Reaper, que destruía las copias creadas de Creeper. No obstante, Dewdney no conocía estos programas mencionados. La primera descripción de del lenguaje RedCode se publicó en marzo de 1984⁵⁰ y en mayo de ese mismo año Dewdney publicó en su columna de Scientific American un artículo hablando de ello.⁵¹ Otro artículo del mismo autor apareció en marzo de 1985⁵² y de nuevo en enero de 1987.

⁵⁰ <http://corewar.co.uk/cwg.txt>

⁵¹ Una versión de este artículo (puede verse en <http://www.koth.org/info/akdewdney/First.htm>

⁵² Ver <http://www.koth.org/info/akdewdney/Second.htm>

La Sociedad Internacional Core Wars (ICWS por sus siglas en inglés), se fundó en 1985. La ICWS publicó nuevos estándares para el lenguaje RedCode (1986 y 1988) y aún hubo una propuesta de actualización para 1994, aunque ésta no fue formalmente propuesta como el nuevo estándar. Aún hoy se usa el borrador de 1994 como el estándar de facto. La sociedad, dirigida por Mark Clarkson (1985–1987), William R. Buckley (1987–1992), y Jon Newman (1992–) ya no existe⁵³.

Como tema colateral, el jugar Core War puede ayudar a aprender a programar en lenguaje ensamblador, actualmente usado solamente cuando se necesita mucha velocidad en algún proceso, por ejemplo, en los videojuegos.

Si Core War es jugar a ser Dios, el siguiente paso en la evolución de esta idea parece ser aún más violenta. En el caso de Core War, cada programador escribía su propio código y a partir de ahí, lo ponía en tela de juicio frente a otros. Las reglas del juego se seguían ciegamente y eventualmente salía un ganador. En realidad, el juego pasaba en menos de un minuto, de acuerdo con declaraciones de sus inventores, y esto hacía que fuese un juego de computadoras poco emocionante, por decirlo de alguna manera.

Sin embargo, el siguiente paso fue crear algo que se llamó Robot War. El primer programa de esta naturaleza fue escrito por Silas Wagner, un regordete programador de mucho ingenio. RobotWar es una batalla de robots en la pantalla de la computadora. La batalla incluso puede verse y seguirse, con información adicional sobre el daño que recibe un robot en todo momento. Originalmente el programa se escribió con un programa llamado RobotWrite, el cual es un editor para escribir el código que ejecutará cada robot al entrar a la arena. El sistema se hizo en el lenguaje TUTOR para el sistema PLATO, en los años setenta del siglo pasado. Más tarde el programa fue comercializado y adaptado para la familia de computadoras Apple II, en 1981.

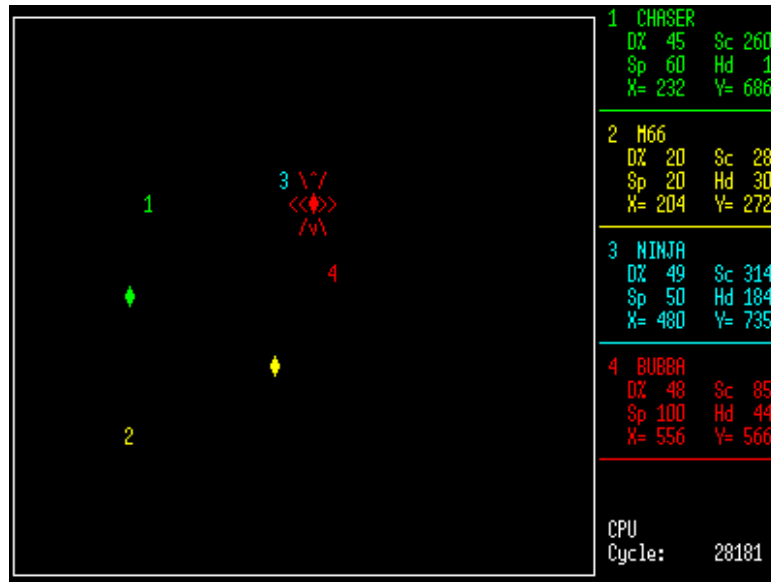
⁵³ Consultar, por ejemplo, <http://corewar.co.uk/history.htm>

Los jugadores de este video juego trabajan de la misma manera que lo hacen los programadores de Core War. Tienen que escribir un programa que opere a un robot (una simulación en la pantalla, no hay que armar un robot real). Los robots, o mejor dicho, el código de cada robot se carga a la memoria y aparecen representaciones de cada uno de ellos. Empieza la batalla y pueden observarse a los robots moviéndose a cierta velocidad, disparando sus armas, usando su radar para deducir distancias, etcétera. No se necesita ninguna destreza manual, pues el juego, como en el caso de Core War, se juega solo. Los jugadores solamente ven su creación cuando entra al cuadrilátero, a la arena en donde se desarrolla la batalla.

La arena o cuadrilátero es un cuadrado en la pantalla, el cual contiene a los robots y la animación que se ejecuta continuamente en el caso de disparar, moverse, dañar a un adversario o quedar dañado el propio robot. El sistema nos mantiene informado del porcentaje de daño. Si se llega al 100%, el robot es eliminado de la lucha.

El lenguaje de los robots es de alto nivel y se parece a BASIC. Cada robot contiene 34 registros que podrían usarse como variables para las funciones de entrada y salida del mismo.

El juego tuvo un relativo éxito y fue la inspiración a otros programas parecidos. Hoy en día, con todo el software libre que hay en Internet, podemos encontrar versiones que se programan en diversos lenguajes, C, Pascal, o versiones recortadas de los mismos. Por ejemplo, P-Robots es un lenguaje de programación de robots que está inspirado en C-Robots, programa creado en 1988 por David Malmberg. En el caso de P-Robots se trata de escribir un programa en un subconjunto del lenguaje Pascal que controle el movimiento, el escáner del robot, el escudo y el sistema de armas. Pueden competir al mismo tiempo cuatro robots y ganará el que no sea dañado totalmente, en un 100%.



P-Robots en acción

Existen muchos entornos para jugar este tipo de juegos⁵⁴ pero lo interesante es la posibilidad de observar qué estrategias son las más viables para tener éxito, para sobrevivir.

⁵⁴ Una referencia muy completa puede verse en <http://corewar.co.uk/crb/index.htm>

Conclusiones

Me estoy convirtiendo en una especie de máquina que observa hechos y llega a conclusiones.

Charles Darwin

Hemos hecho un largo recorrido sobre la vida artificial. Hemos visto muchos de los grandes científicos que han encontrado en este tema una cantidad de nuevas ideas que parecen dar con algunos aspectos de cómo se desarrolla la vida real. La vida artificial (*Alife*) es una interesante rama de la ciencia, donde se combina el cómputo (para las simulaciones) y la biología teórica.

Los sistemas de cómputo permiten poner a prueba conceptos de evolución, auto-organización, entre otros. Pero tal vez el tema se vuelve apasionante cuando aparecen las características emergentes, que nadie sabe a ciencia cierta de donde salen, pero que de pronto parecen organizar a los individuos. Es un gran reto el de la biología el tratar de explicar la vida y posiblemente los estudios en vida artificial permitan entender o dar más luz a temas como el origen de la vida, la auto-organización, la evolución y la relación entre ser apto y adaptado a las circunstancias del entorno.

La vida en la Tierra está organizada en cuatro niveles en lo que podríamos llamar su estructura: el nivel molecular, el celular, el del organismo y el de las poblaciones. En todos estos casos los sistemas complejos se han adaptado y exhiben comportamientos que emergen de la interacción de un número de elementos que vienen de niveles más simples. Por eso, entender la vida requiere al final de cuentas en el conocimiento de todos estos niveles.

Es claro que la computadora aquí cobra un papel preponderante. Por una parte, cambia el paradigma que en ciencia parece ser el denominador común, el cual implica describir un fenómeno y aplicar un modelo que lo explique, que en general involucra matemáticas y ecuaciones. Entonces lo que hacemos es intentar resolver esas ecuaciones y ver qué tanto encajan en el modelo. Sin embargo, con la computadora podemos hacer algo más, que es ponerla a trabajar bajo supuestos de modelos que antes nos eran imposibles. Así, ya no escribimos código que resuelva un conjunto de ecuaciones. Ahora planteamos un modelo de reglas, las cuales se ejecutan maravillosamente bien (y de manera ciega), sobre los elementos que queremos. El resultado de estos cálculos dista de ser obvio porque en la mayoría de los casos es indecible y en el fondo no tenemos una respuesta, sino una interpretación de lo que parece estar pasando.

Esto es precisamente lo que ocurre cuando hallamos características emergentes, aquellas que parecen surgir de la nada y que sorpresivamente crean una especie de auto-organización, punto fundamental para la existencia de la vida, de acuerdo a John von Neumann. ¿Por qué se producen estas características emergentes? ¿Qué inteligencia está detrás de ella? ¿Hay inteligencia atrás de ello o es parte de la naturaleza misma en la que estamos inmersos? Los experimentos parecen mostrar que no hay ninguna inteligencia atrás, que no hay un plan preconcebido en el desarrollo de los organismos y las pruebas de ello parecen apoyarse seriamente en los diversos experimentos con Genesys o bien, con los experimentos hechos por Kauffman.

Dawkins, por ejemplo, halla fascinante que reglas ciegas lleven a comportamientos emergentes y la creación de sus *biomorfismos* son una prueba de ello, en donde el afamado biólogo reconoce que le fueron inesperados. Y es porque la ciencia tiene que luchar cotidianamente con los prejuicios que muchas otras experiencias nos han hecho creer que pareciera que hay un plan perfectamente creado para los fenómenos naturales. Ya Einstein habría dicho que “Dios no juega a los dados”, pero aunque él pensaba en términos de la física, bien podría desdeñarse su frase porque en este caso de la vida artificial parece ser que efectivamente *Dios parece estar jugando a los dados*.

En la vida artificial se han tenido que implementar, desde luego, modelos que permiten analizar muchos miles de generaciones. Y esto es un punto que no debemos pasar por alto. A la naturaleza le tomó mil millones de años más o menos para que las primeras células se formaran. Sin embargo se necesitaron otros tres mil millones de años para que estas células se convirtieran en organismos multicelulares. Hay muchas interrogantes sobre cómo ocurrió todo esto y evidentemente son especulaciones –de alguna manera educadas– sobre lo que pudo haber pasado realmente. Gracias a la computadora podemos ahora simular miles y miles de generaciones y ver qué pudo haber ocurrido. La vida de los seres humanos es un paréntesis en la evolución y por ello no nos queda más remedio que hacer estos supuestos porque no podemos analizar los cambios y mutaciones, los cuales no necesariamente se dan rápidamente.

Probablemente la vida artificial tenga que combinar sus experiencias con la teoría de caos. Los atractores extraños y/o cíclicos son temas que merecen mayor profundidad. Piénsese en el increíble comportamiento de la hormiga de Langton, la cual no requiere además, una serie de sofisticadas reglas. Nuestra experiencia nos habría hecho creer que en este experimento virtual, la hormiga estaría caminando azarosamente por siempre, pero hasta la generación 10,000 aproximadamente, vemos cómo se crea esa carretera (*highway* del original en

inglés), que es aparentemente inexplicable, pero que nos habla de una propiedad emergente.

Hay desde luego más elementos: la reproducción como parte fundamental de las simulaciones y la evolución, la adaptación al entorno y la lucha por sobrevivir. Esto plantea una interrogante enorme, por ejemplo, ¿por qué los individuos en muchas de las simulaciones solamente logran llegar a un máximo local? ¿Por qué no buscan un nuevo máximo? Este problema se plantea fundamentalmente en la Inteligencia Artificial y es el llamado algoritmo de *hill climbing* (o de ascenso de una colina). Trata de un procedimiento que busca la optimización. Es un algoritmo que se llama un sinnúmero de veces, hasta que ocurre una condición que decide el final del mismo. Se inicia casi siempre con una solución arbitraria, que bien puede ser muy mala, pero entonces se busca hallar una solución mejor, la cual varía algunos parámetros y la valora de forma incremental.

Cualquier modelo de la vida artificial que involucre la evolución tendrá que explicar por qué en los sistemas biológicos los individuos cada vez son más aptos, es decir, buscan nuevos máximos aunque sean locales, y no parecen detenerse en esta búsqueda que se antoja incesante.

Pero independientemente de todo esto, hay una crítica que hay que contemplar en todo lo que se refiere a vida artificial. Los investigadores de este tema hacen simulaciones, analizan y observan lo que ocurre con los “organismos” que han definido en sus programas. Cuando emerge una nueva característica inesperada entonces buscan describirla y de algún modo explicarla. Pero este terreno es demasiado pantanoso.

Por ejemplo, imaginemos que estamos observando a una comunidad real de hormigas y de pronto un conjunto de ellas toma una decisión curiosa, digamos, alejarse del nido donde viven. Como observadores e investigadores de este fenómeno, intentaremos elaborar alguna hipótesis por la cual las hormigas han

tomado este derrotero. ¿Qué tan certera será nuestra idea de lo que está pasando? Difícil saberlo. Lo que nos obligaría en todo caso a plantear algún experimento que busque demostrar nuestra hipótesis original. Pero esto en ocasiones lo pasan por alto los investigadores de vida artificial. Hacen como Kauffman que de pronto observa el comportamiento cíclico en su popular modelo y a partir de ahí ya se decide que estamos ante una propiedad emergente.

La realidad es que, a lo más, estamos de nuevo en el terreno de las especulaciones. Desde luego no parece posible tener control de todas las variables y como en el caso de las poblaciones, asumimos hipótesis que afectan globalmente a toda la población y no a cada individuo *per se*. Esto podría ser sin duda un tema que deba discutirse más a fondo.

Por otra parte, no debemos olvidar dos conceptos fundamentales, la reproducción y la recursión como mecanismo para que las siguientes generaciones tengan una réplica de la carga genética. De alguna manera la Naturaleza ha encontrado el mecanismo recursivo como el más eficiente para copiar información de padres a hijos. Desde luego que en esto también tenemos que hablar de la recombinación de genes en la reproducción sexual, la cual como ya vimos, parece ser mucho más apta que la reproducción asexual. Por ello hemos dedicado un capítulo a la recursión en cómputo, porque el mecanismo es sorprendentemente eficiente a pesar de que en cómputo requiere de más recursos que cuando usamos la iteración simple. ¿Por qué la Naturaleza usa recursión? Es una pregunta que no podemos aún contestar.

Hay que decir que las simulaciones que se han hecho en los diferentes entornos creados por los investigadores, van más allá de mover “hormigas” en un medio ambiente. De hecho cada hormiga es un programa que hace algo, y que además, puede modificar su actuación en la medida que sus genes se recombinan a través de reglas ciegas. El uso de redes neuronales, como hace Larry Yaeger permite que los individuos de las poblaciones usen un esquema de extrapolación, que es

en esencia lo que se puede hacer con esta tecnología de redes neuronales. Esto parece ser un avance notable en las simulaciones.

Queda quizás la pregunta de si todos estos experimentos son realmente simulaciones o bien es vida artificial. Es difícil decirlo. Es la opinión del autor de que estas simulaciones son en mucha medida la vida artificial y los virus computacionales demuestran que en el entorno de las computadoras estos programas generan comportamientos que pueden no gustarnos, pero que evidentemente “enferman” a las computadoras. De alguna manera es vida ¿O no?

Como sea, el viaje que hemos emprendido ha sido fantástico y apenas estamos empezando a entender de qué se trata la vida, todo a través de estos experimentos virtuales en donde la computadora es como el nuevo microscopio del biólogo moderno.

Bibliografía

Capítulo I: ¿Qué es la vida?

- **Artificial Life**, *Steven Levy*, Vintage Press 1992
- **Vida simulada en el ordenador**, *Emmeche Claus*, Gedisa Editorial (2000)
- **Adventures in Artificial Life**, *Clayton Walnum*, Editorial Que (1993)

Capítulo II: John von Neumann y los autómatas celulares

- **Theory of Self-Reproducing Automata** (libro escaneado en línea), *von Neumann, John; Burks, Arthur W.* (1966), www.walenz.org.

Capítulo III: El juego de la vida de John Conway

- **The fantastic combinations of John Conway's new solitaire game “life”**, *Martin Gardner*, Scientific American, Mathematical Games, October 1970
- **On cellular automata, self-reproduction, the Garden of Eden, and the game “life”**, *Martin Gardner*, Scientific American, Mathematical Games, February 1971

Capítulo IV: La nueva ciencia de Stephen Wolfram

- **A New Kind of Science**, *Stephen Wolfram*, Wolfram Media (2002)
- **Physics Like Models of Computations**, *Norman Margulus*, *Physica* 10D (1984), pp 81-95
- **Discrete Systems, Cell-Cell Interactions and Color Pattern of Animal (I) Conflicting Dynamics and Pattern Formation**, *G. Cocho, R. Pérez-Pascual, J.L. Rius*, UNAM
- **Discrete Systems, Cell-Cell Interactions and Color Pattern of Animal (II) Conflicting Dynamics and Pattern Formation**, *G. Cocho, R. Pérez-Pascual, J.L. Rius*, UNAM
- **Cellular Automata as an alternative to differential equations in modeling physics**, *Tomaso, Toffoli*, *Physica* 10D (1984) pp. 117-127
- **Simulating Physics with Cellular Automata**, *Gérard Y. Vichniac*, *Physica* 10D (1984), pp 96-116
- **Universality and Complexity in Cellular Automata**, *Stephen Wolfram*, *Physica* 10D, pp. 1-35
- **Statistical Mechanics of Cellular Automata**, *Stephen Wolfram*, *Review of Modern Physics*, Vol 55, No. 3, Julio 1983.
- **Essays on Cellular Automata**, *Arthur W. Burks* (editor), University of Illinois Press (1970)
- **Computer Software in Science and Mathematics**, *Stephen Wolfram*, *Scientific American*, September 1984

Capítulo V: Los autómatas primitivos de Heiserman

- **Projects in Machine Intelligence for your Home Computer**, *David L. Heiserman*, Tab Books 1982

Capítulo VI: Virus informáticos y vida artificial

- **Computer Viruses** (Dissertation), *Fred Cohen*, Universidad del Sur de California, Enero 1986 (la tesis completa puede consultarse en este sitio: <http://all.net/books/Dissertation.pdf>).

Capítulo VII: Autoreferencia y recursión

- **Self-Reproducing programs**, *Burger, John, David Brill, and Filip Machi*, Byte. Agosto 1980. pp. 74-75.
- **Computer Recreations; Self-Reproducing Automata, Software - Practice & Experience**, *Bratley, Paul and Jean Millo*, Vol. 2 (1972). pp. 397-400.
- **Self-Reproducing Programs**, Creative Computing. *Hay, Louise*, Julio, 1980. pp. 134-136.
- **Godel, Escher, and Bach: an Eternal Golden Braid**, *Hofstadter, Douglas R.* Basic Books, Inc. New York, New York. pp. 498-504.

Capítulo VIII: La hormiga caótica de Langton y sus bucles

- **Artificial Life: An Overview**, *Christopher G. Langton* (Editor), MIT Press, 1995.
- **Self Reproduction in Cellular Automata**. *Christopher G. Langton*, Physica D, **10**, 1984.
- **A Self-Repairing Multiplexer-Based FPGA Inspired by Biological Processes**, *Gianluca Tempesti*, Tesis doctoral (Que puede verse aquí: <http://lslwww.epfl.ch/pages/embryonics/thesis/>)

Capítulo IX: La teoría evolutiva moderna, según Kauffman

- **Origin of Order: Self-Organization and Selection in Evolution**, *Kauffman Stuart*, Oxford University Press 1992
- **Investigating Kauffman's NK Model for Agent-Based Modelling**, Richard Edward Mellor, University of Bath (2007). Una versión en línea puede hallarse en <http://www.cs.bath.ac.uk/~mdv/courses/CM30082/projects.bho/2006-7/Mellor-RE-dissertation-2006-7.zip.pdf>
- **Stability of the Kauffman Model**, *Sven Bilke y Fredrik Sjunnesson*, University of Lund (2001). Una versión en formato PDF puede verse en <http://arxiv.org/pdf/cond-mat/0107035.pdf>.

Capítulo X: Los sistemas-L de Lindenmayer

- **The Algorithmic Beauty of Plants**, *Przemyslaw Prusinkiewicz, Aristid Lindenmayer*, Springer -Verlag 2000. Hay una versión legal en la red (formato PDF) en <http://algorithmicbotany.org/papers/abop/abop.pdf>
- **Paradigms of pattern formation: Towards a computational theory of morphogenesis**, *Przemyslaw Prusinkiewicz*, In Pattern Formation in Biology, Vision, and Dynamics. Hay una versión legal (formato PDF) en <http://algorithmicbotany.org/papers/paradigms.pf2000.pdf>

Capítulo XI: Los biomorfismos de Dawkins

- **The Blind Watchmaker**, *Dawkins, Richard*, New York: W. W. Norton & Company, Inc. (1996). El libro en línea, legal, puede hallarse en <http://uath.org/download/literature/Richard.Dawkins.The.Blind.Watchmaker.pdf>

Capítulo XII: La genial idea de John Holland

- **Adaptation in Natural and Artificial Systems**, *Holland, John*, Cambridge, MA: MIT Press, 1992
- **Theory of Genetic Algorithms**, *Schmitt, Lothar M*, Theoretical Computer Science 259 (2001), pp. 1–61

Capítulo XIII: El dios darwiniano de Danny Hillis

- **The pattern on the stone: The simple ideas that make computers work**, Hillis, D. Basic Books (1998)

Capítulo XIV: Las colonias de hormigas de la UCLA

- **The Genesys System: Evolution as a Theme in Artificial Life**, *David Jefferson, Robert Collins, Claus Cooper, Michael Dyer, Margot Flowers, Richard Korf, Charles Taylor, Alan Wang, UCLA*. Hay una versión en línea: <http://www.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html>

Capítulo XV: Los polimundos de Larry Yaeger

- **Functional and Structural Topologies in Evolved Neural Networks**, *Joseph T. Lizier, Mahendra Piraveenan, Dany Pradhana, Mikhail Prokopenko y Larry S. Yaeger*, University of Sidney (2008). Una versión previa del artículo final puede verse en http://www.shinyverse.org/larry/LizierEtAl2009_Trends_ECAL.pdf
- <http://www.shinyverse.org/larry/> (sitio web de Larry Yaeger)

- **Computational Genetics, Physiology, Metabolism, Neural Systems, Learning, Vision, and Behavior or PolyWorld: Life in a New Context**, Larry Yaeger. Una versión en línea (formato PDF) puede leerse en <http://www.shinyverse.org/larryy/Yaeger.ALife3.pdf>.

Capítulo XVI: Jugando a ser Dios

- **In the game called Core War hostile programs engage in a battle of bits**, A. K. Dewdney, Scientific American, Computer Recreations, May 1984
- **A Core War bestiary of viruses, worms and other threats to computer memories**, A. K. Dewdney, Scientific American, Computer Recreations, March 1985
- **A program called MICE nibbles its way to victory at the first Core War tournament**, A. K. Dewdney, Scientific American, Computer Recreations, January 1987
- **Of worms, viruses and Core War**, A. K. Dewdney, Scientific American, Computer Recreations, March 1989.

Apéndice I

Programa (software) de la hormiga de Langton y sus bucles

El programa está escrito en *Delphi 7* (actualmente de la empresa *Embarcadero*), y modela el comportamiento de la hormiga de Langton en el espacio LR (left-right) solamente.



Programa para ver y analizar el comportamiento de la hormiga de Langton

Contiene un menú básico con las siguientes opciones:

Configuración

Permite que la hormiga camine en una malla blanca o bien, que se le añadan puntos negros. El usuario puede definir si quiere un rango de 5% a 95% de puntos negros. Esto hace que el comportamiento de la hormiga requiera de muchos más pasos para que, eventualmente, produzca un atractor. La pregunta obligada es si éste se producirá siempre.

Procesa

Ejecuta el programa, es decir, la hormiga empieza a moverse de acuerdo a las reglas definidas.

Resultados

La imagen resultante del proceso de ejecutar el autómata puede guardarse como una imagen con formato BMP.

¡Alto!

Detiene temporalmente la ejecución del autómata, se puede reanudar dando click en el comando *Procesa*.

La rutina principal se llama ***MueveHormiga***, y está definida de esta manera:

```
procedure MueveHormiga;  
  
    //Si la casilla es blanca, se convierte en negra. Se mueve la hormiga una  
    //casilla y se gira la derecha 90 grados.  
    //Si la casilla es negra, se convierte en blanca. Se mueve la hormiga una  
    //casilla y se gira a la izquierda 90 grados.  
  
begin  
    if Form1.Imagel.Picture.Bitmap.Canvas.Pixels[PosX,PosY] = clWhite then  
        begin  
            Form1.Imagel.Picture.Bitmap.Canvas.Pixels[PosX,PosY] := clBlack;
```

```

        Cambio := TRUE; //vuelta a la derecha
        Direccion := Direccion + 1;
    end
else
    if Form1.Imagen1.Picture.Bitmap.Canvas.Pixels[PosX,PosY] = clBlack then
    begin
        Form1.Imagen1.Picture.Bitmap.Canvas.Pixels[PosX,PosY] := clWhite;
        Cambio := FALSE; //vuelta a la izquierda
        Direccion := Direccion - 1;
    end;
    if Direccion > 3 then Direccion := 0;
    if Direccion < 0 then Direccion := 3;

    Case Direccion of
        0      : begin
                    PosX := PosX + 1;
                    Dir := 'E'; //dirección Este
                end;
        1      : begin
                    PosY := PosY + 1;
                    Dir := 'S'; //dirección Sur
                end;
        2      : begin
                    PosX := PosX - 1;
                    Dir := 'O'; //dirección Oeste
                end;
        3      : begin
                    PosY := PosY - 1;
                    Dir := 'N'; //dirección Norte
                end;
    end;
    Form1.Label3D4.Caption := Dir;
    Application.ProcessMessages;
end; // MueveHormiga

```

Al crearse el campo en donde se mueve la hormiga, se genera una imagen BMP de 400 x 400 pixeles, suficiente para ver el desarrollo. Si la hormiga se sale de los límites, entonces se detiene la ejecución del software.

Estos son los elementos principales de la rutina de inicialización de las gráficas:

```

//pinta la arena de blanco, que es donde se moverá la hormiga
Imagen1.Canvas.Pen.Color:= clWhite;
for i := 0 to 399 do
begin
    Imagen1.Canvas.MoveTo(0,i);
    Imagen1.Canvas.lineTo(399,i);
end;
//ponla en el centro del cuadrilatero
PosX := 200;
PosY := 200;
Imagen1.Canvas.MoveTo(200,200);

```

Con respecto a los bucles de Langton, el usuario *Bluatigro*, del foro FreeBasic.net (<http://www.freebasic.net/forum/viewtopic.php?f=3&t=20479>), decidió programar esta idea de Langton. Para ello usó precisamente FreeBasic⁵⁵. Éste es su código, en dominio público:

```
screen 20 , 32 , 2
dim as integer p( 256 , 256 ), q( 256 , 256 )
dim as integer y , x
y = 0
dim as integer itern
itern = 1
dim as long clr(7) = { 0 _
                      , rgb( 0 , 0 , 255 ) _
                      , rgb( 255 , 0 , 0 ) _
                      , rgb( 0 , 255 , 0 ) _
                      , rgb( 255 , 0 , 255 ) _
                      , rgb( 0 , 255 , 255 ) _
                      , rgb( 255 , 255 , 0 ) _
                      , rgb( 255 , 255 , 255 ) }
dim shared as integer i , horizMin , horizMax , vertMin , vertMax , flag
dim as string a , in
while a <> ""
  read a
  for x = 1 to len( a )
    p( x + 128 , y + 128 ) =val( mid( a , x , 1 ) )
  next x
  y = y + 1
wend

data " 22222222"      '   initial conditions
data "2170140142"
data "2022222202"
data "272      212"
data "212      212"
data "202      212"
data "272      212"
data "21222222122222"
data "20710710711111"
data " 222222222222"
data ""

dim as integer lookUp( 100000 ) ,pp,index,op

open "langton.txt" for input as #1
dim rule as string
while not( eof( 1 ) )
  line input #1 , rule      '   old      N   E   S   W      new
  in =left( rule , 5 )
  op =val( right( rule , 1 ) )

  pp =val( in ) :                      lookUp( pp)
=op
''   print "Rule <"; left( rule, 1); " "; mid( rule, 2, 4); " "; right( rule,
1); ">"; _
```

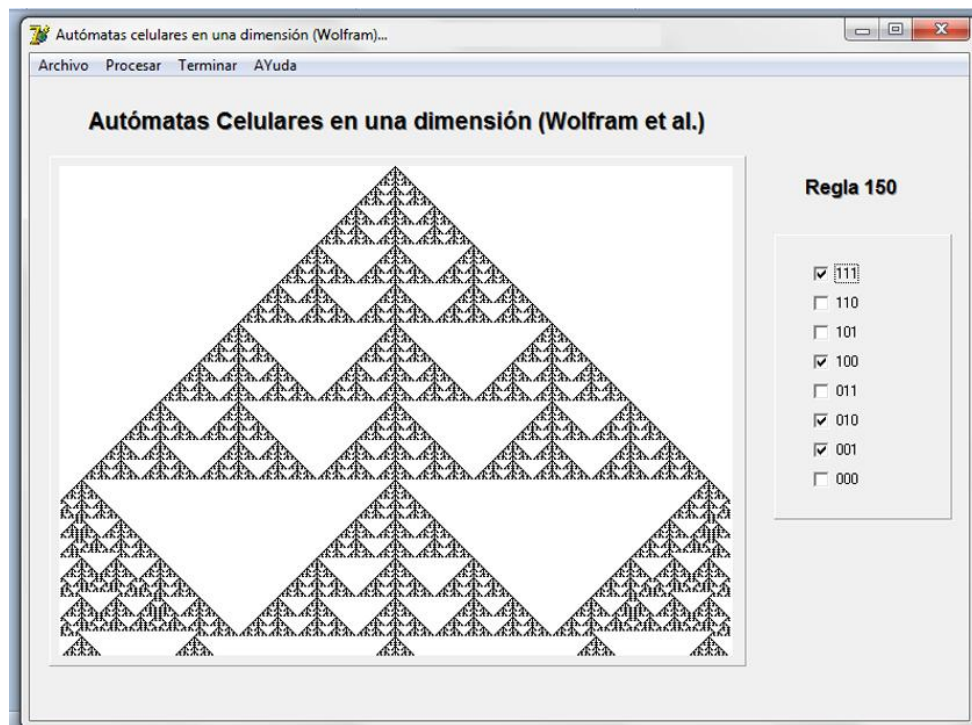
⁵⁵ El compilador FreeBasic es de código abierto y gratuito. Puede descargarse de <http://www.freebasic.net/>.

000000	014721	111244	200326	206222	402520
000012	016251	111251	200423	206722	403221
000020	017221	111261	200517	207122	500022
000030	017255	111277	200522	207222	500215
000050	017521	111522	200575	207422	500225
000063	017621	112121	200722	207722	500232
000071	017721	112221	201022	211222	500272
000112	025271	112244	201122	211261	500520
000122	100011	112251	201222	212222	502022
000132	100061	112277	201422	212242	502122
000212	100077	112321	201722	212262	502152
000220	100111	112424	202022	212272	502220
000230	100121	112621	202032	214222	502244
000262	100211	112727	202052	215222	502722
000272	100244	113221	202073	216222	512122
000320	100277	122244	202122	217222	512220
000525	100511	122277	202152	222272	512422
000622	101011	122434	202212	222442	512722
000722	101111	122547	202222	222462	600011
001022	101244	123244	202272	222762	600021
001120	101277	123277	202321	222772	602120
002020	102026	124255	202422	300013	612125
002030	102121	124267	202452	300022	612131
002050	102211	125275	202520	300041	612225
002125	102244	200012	202552	300076	700077
002220	102263	200022	202622	300123	701120
002322	102277	200042	202722	300421	701220
005222	102327	200071	203122	300622	701250
012321	102424	200122	203216	301021	702120
012421	102626	200152	203226	301220	702221
012525	102644	200212	203422	302511	702251
012621	102677	200222	204222	401120	702321
012721	102710	200232	205122	401220	702525
012751	102727	200242	205212	401250	702720
014221	105427	200250	205222	402120	
014321	111121	200262	205521	402221	
014421	111221	200272	205725	402326	

Apéndice II

Programa (software) de los autómatas celulares (1D)

El *software* para estudiar los autómatas celulares unidimensionales utiliza una imagen bmp como un “canvas” (lienzo), en donde se van dibujando las generaciones de acuerdo a las reglas locales de evolución. El programa no se limita a reglas “legales”, en donde se cumplen los criterios de Wolfram, sino que permite estudiar las 256 posibles reglas de evolución.



*Pantalla del programa de los autómatas celulares
(mostrando la **regla 150**)*

El sistema presenta una ventana en donde hay una imagen, un *bitmap* (mapa de bits), de 550 x 400 pixeles. Es ahí donde se dibujan las generaciones y evolución del autómatas. A la derecha se muestran las reglas locales (con el número de regla asociado).

Las opciones del software son relativamente sencillas:

- *Archivo*
- *Procesar*
- *Terminar*
- *Ayuda*

Va la descripción de cada opción en particular:

Archivo

Esta opción contiene cinco sub-opciones:

- *Leer regla*
- *Crear regla*
- *Guardar imagen*
- *Una sola célula*
- *Autómata al azar*

Leer regla: permite al usuario leer un archivo de texto ASCII (sin caracteres especiales) que contiene 8 líneas. Cada una representa un valor binario (0 ó 1), los cuales se le asignan a cada una de las posibilidades de tres células (en donde

la célula de en medio es la célula de interés). Por ejemplo, la regla 30 se define en el archivo “regla30.txt” como:

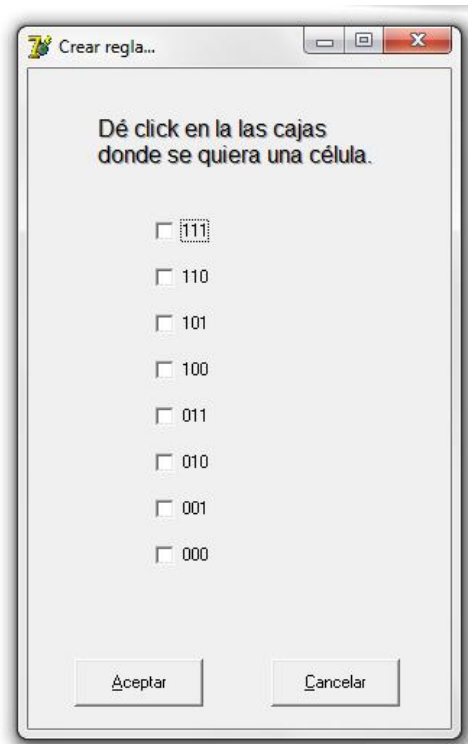
0	(asociado al 111)
0	(asociado al 110)
0	(asociado al 101)
1	(asociado al 100)
1	(asociado al 011)
1	(asociado al 010)
1	(asociado al 001)
0	(asociado al 000)

Cabe decir que solamente aparecen las cifras de la izquierda. Lo puesto a la derecha (en paréntesis), define cuál regla local se usa.

Crear regla: Permite al usuario definir las reglas locales de evolución. Al usar esta opción, aparecerá la siguiente ventana (ver más abajo), en donde simplemente hay que dar click en la caja correspondiente a la regla local que nos interesa. Es ahí donde habrá un 1 (si se marca la caja) ó 0 (si no se marca).

Guardar imagen: permite al usuario guardar la imagen resultante de la simulación de las reglas locales de evolución de un autómata particular. La imagen se guarda en formato BMP (*bitmap*).

Una sola célula: le dice al usuario del programa que la simulación correrá empezando con una sola célula en la parte superior de la pantalla (en medio de la misma). Esto permite ver la evolución y los patrones que se generan en las diversas reglas.



*Ventana para crear una regla
en los autómatas celulares unidimensionales*

Autómata al azar: En este caso el programa crea una generación inicial con puntos al azar en toda la línea. Usa una función *random* que puede dar dos valores, poner una célula en la línea o no ponerla. Es una función del 50% exactamente.

Procesar

Genera la simulación del autómata celular unidimensional. El proceso es muy rápido y tarda pocos segundos, incluso cuando se usa el autómata inicial al azar.

Terminar

Finaliza el software.

Ayuda

Aquí hay dos sub-opciones:

- *Manual*
- *Acerca de...*

La primera (*Manual*), despliega un texto en la pantalla en donde se explican los fundamentos de los autómatas celulares unidimensionales y su relación con Wolfram. La segunda sub-opción (*Acerca de...*) simplemente muestra los créditos del autor y la forma de contactarlo en Internet.

La rutina clave del programa es el procesamiento de cada regla y el hecho de que se dibuja finalmente en la pantalla. Las generaciones, a diferencia del juego de la vida de Conway, se van pintando una a una debajo de la generación anterior. Este es el procedimiento más importante del programa:

```
procedure TForm1.Procesar1Click(Sender: TObject);
var
  i      : integer;
  j      : integer;
  Pix    : array[1..3] of integer;
  PixC   : array[1..3] of chAR;
  Value  : integer;
begin
  if Regla = FALSE then ShowMessage('Cuidado, no se ha definido ninguna regla
local a usar...')
  else
    // revisamos cada línea y usamos la regla definida
    for j := 0 to 399 do
      begin
        for i := 1 to 547 do
          begin
            //lee pixeles de la línea
            if Image1.Canvas.Pixels[i-1,j] = clWhite then
              begin
                Pix[3] := 0;
                PixC[3] := '0';
              end
            else
              begin
                Pix[3] := 1;
                PixC[3] := '1';
              end;
            end;
          end;
        end;
      end;
    end;
```

```

if Imagen1.Canvas.Pixels[i,j] = clWhite then
begin
    Pix[2] := 0;
    PixC[2] := '0';
end
else
begin
    Pix[2] := 1;
    PixC[2] := '1';
end;
if Imagen1.Canvas.Pixels[i+1,j] = clWhite then
begin
    Pix[1] := 0;
    PixC[1] := '0';
end
else
begin
    Pix[1] := 1;
    PixC[1] := '1';
end;
if Imagen1.Canvas.Pixels[i,j] = clWhite then Pix[2] := 0 else Pix[2] := 1;
//ponemos el valor que corresponda de acuerdo a la regla
Value := (Pix[3] * 4) + (Pix[2] * 2) + (Pix[1] * 1);
if ((i < 0) or (i > 549)) then else
begin
if Valor[Value] = 0 then Imagen1.Canvas.Pixels[i,j+1] := clWhite
else Imagen1.Canvas.Pixels[i,j+1] := clBlack;

end;
//IF ((PixC[3] = '1') or (PixC[2] = '1') OR (PixC[1] = '1')) then
showmessage(PixC[3]+PixC[2]+PixC[1] + ' ' + inttostr(valor[Value]));
end;
Application.ProcessMessages;
end;
end;

```

Las reglas se piden en una forma aparte (en Delphi una forma es básicamente una ventana). El código fuente es muy simple:

```

procedure TForm2.Button1Click(Sender: TObject);
var
    Nombre      : string;
    Value       : integer;
    i           : integer;
    T           : TextFile;
begin
    if CheckBox1.Checked = True then Valor[7] := 1 else Valor[7] := 0;
    if CheckBox2.Checked = True then Valor[6] := 1 else Valor[6] := 0;
    if CheckBox3.Checked = True then Valor[5] := 1 else Valor[5] := 0;
    if CheckBox4.Checked = True then Valor[4] := 1 else Valor[4] := 0;
    if CheckBox5.Checked = True then Valor[3] := 1 else Valor[3] := 0;
    if CheckBox6.Checked = True then Valor[2] := 1 else Valor[2] := 0;
    if CheckBox7.Checked = True then Valor[1] := 1 else Valor[1] := 0;
    if CheckBox8.Checked = True then Valor[0] := 1 else Valor[0] := 0;
    value := (valor[7] * 128) + (valor[6] * 64) + (valor[5] * 32) + (valor[4] * 16)
+
            (valor[3] * 8) + (valor[2] * 4) + (valor[1] * 2) + (valor[0] * 1);
    Nombre := 'Regla' + inttostr(value) + '.txt';
    Label13d2.caption := Nombre;
    SaveDialog1.FileName := Nombre;
end;

```

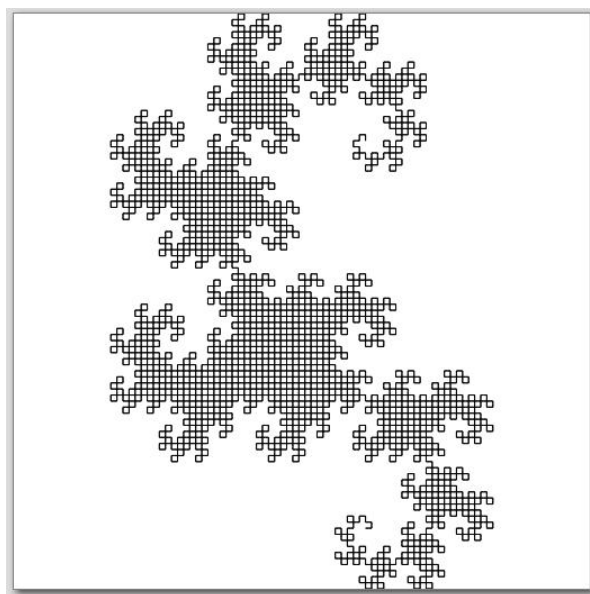
```
if SaveDialog1.Execute = TRUE then
begin
  AssignFile(T,Nombre);
  Rewrite(T,Nombre);
  for i := 7 downto 0 do
    writeln(T,inttostr(valor[i]));
  Closefile(T);
  ShowMessage('Se ha guardado el archivo '+ Nombre);
end;
Close;
end;
```

La rutina puede guardar la regla (la cual usa el número de regla para crear el archivo), lo que facilita volver a probar o hacer un análisis de reglas parecidas sin perder la configuración.

Apéndice III

Sistemas L (Lindenmayer)⁵⁷

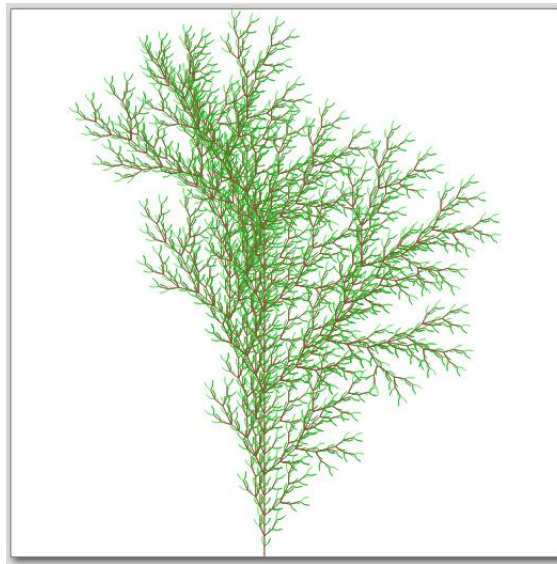
Las ideas de Lindenmayer pueden representarse gráficamente y para sorpresa de muchos, simular cómo se ven, por ejemplo, algunas plantas y helechos. Hay diversos programas en Internet que pueden incluso ejecutarse en el navegador de su elección. Uno de los más sencillos y poderosos está en la página siguiente, <http://www.kevs3d.co.uk/dev/lsystems/#>, la cual permite definir el axioma a usar y las reglas de transformación. Por ejemplo, tomando un ángulo para la tortuga de Logo de 90 grados y como axioma FX, donde las reglas de cambio se dan por 1. $X = X + YF$ y 2. $Y = -FX - Y$, obtenemos la siguiente imagen:



⁵⁷ **The Algorithmic Beauty of Plants**, de Przemyslaw Prusinkiewicz y Aristid Lindenmayer; Springer-Verlag (1990, 1996), puede conseguirse en formato electrónico (PDF) de manera gratuita en <http://algorithmicbotany.org/papers/#abop>. Vale mucho la pena por las imágenes coloridas que presenta, amén de un análisis de los sistemas L.

Que se llama “curva de Hilbert”. Esta es una imagen característica en el estudio de los fractales, a todo esto.

Si tomamos el mismo axioma, FX y utilizamos las reglas de transformación $F=C0FF-[C1-F+F]+[C2+F-F]$ y $X=C0FF+[C1+F]+[C3-F]$, con tan solo 5 iteraciones y con un giro de 25 grados, encontraremos esta sorprendente imagen:



Lo cual finalmente se asemeja a una planta. De alguna manera la visualización de los sistemas L, de Lindenmayer, da una representación gráfica que se acerca mucho a lo que vemos en la Naturaleza. Cabe señalar que estos sistemas se usan incluso para simular plantas para las películas animadas en tres dimensiones. El modelo de la visualización es extraordinario en este caso.

Todos estos desarrollos de software a veces obligan a replantear los programas en términos de lenguajes que sean más fáciles o adecuados para realizar ciertas tareas. De hecho, los autores de este Sistema L en línea, usaron la herramienta *Smart Mobile Studio*, la cual permite escribir en el lenguaje Pascal (con un sabor a Delphi y Free Pascal), pero generar código HTML5, la cual es una aplicación que

puede correrse en una página web y por ende, útil incluso para usarse en el mercado de los dispositivos móviles. Más información en la página oficial.⁵⁸

⁵⁸ <http://smartmobilestudio.com/>

Apéndice IV

Juego de la vida (Conway)

El juego de la vida de Conway es quizás uno de los programas más populares desde que las computadoras fueron relativamente accesibles. Originalmente Conway usaba un tablero de Go, el juego oriental, pero pronto un programa de computadora reemplazó esto, con la virtud de evitar los errores humanos al manipular las células dentro del tablero.

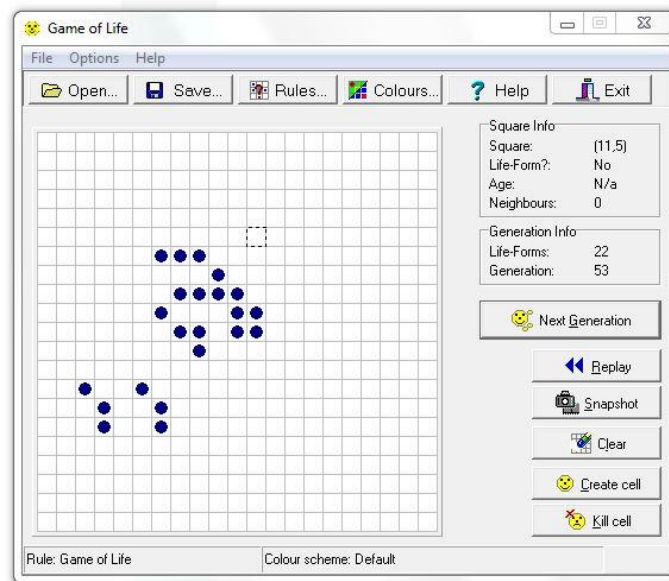
Las reglas para el juego son muy simples y esto es quizás lo que lo hace aún más intrigante:

- Cada célula que no tenga al menos un vecino, muere (de soledad).
- Cada célula con cuatro o más vecinos (es decir, en su vecindad), muere (por sobrepoblación).
- Una célula con dos o tres vecinos sobrevive a la siguiente generación.
- Un espacio vacío, una celda sin estar ocupada, rodeada por tres células, generan una nueva. (nacimiento).

El juego procesa cada célula y sus ocho vecindades y entonces pinta la nueva generación. Esto se hace en paralelo, es decir, si procesamos una célula y sus vecindades, el resultado de la misma no afecta a otras células en esa generación, sino hasta que se procesan todas las células correspondientes.

Existen muchísimos programas que ya han sido escritos para jugar el juego de la vida. Nosotros usaremos el que se encuentra en la página

<http://www.delphidabbler.com/software/life>, pues permite interactuar con cualquier configuración que se le ponga. Se pueden ver las configuraciones y paso a paso, cómo van cambiando las generaciones.



Juego de la vida de Conway

Hay infinidad de código fuente sobre la idea del juego de la vida de Conway.⁵⁹ Unos tienen más características que otros, pero el núcleo de su funcionamiento se reduce a las reglas que el matemático británico puso y que son extraordinarias en su funcionalidad.

Por ejemplo, he aquí la versión de BBC BASIC, de autor desconocido:⁶⁰

```
dx% = 64
dy% = 64
DIM old&(dx%+1,dy%+1), new&(dx%+1,dy%+1)
VDU 23,22,dx%*4;dy%*4;16,16,16,0
```

⁵⁹ Véase http://rosettacode.org/wiki/Conway%27s_Game_of_Life para la implementación del juego de la vida en una multitud de lenguajes de programación diferentes.

⁶⁰ BBC BASIC para Windows es una implementación más del lenguaje BASIC (32 bits). Fue creada por Richard T. Russell y tiene algunas características que lo hacen más poderoso que otras implementaciones. El autor permite descargar una versión gratuita (limitada a 16K), la cual es totalmente funcional. La versión comercial cuesta unas 30 libras esterlinas. Más información en <http://www.bbcbasic.co.uk/bbcwin/bbcwin.html>.

```

OFF

REM Set blinker:
old&(50,50) = 1 : old&(50,51) = 1 : old&(50,52) = 1
REM Set glider:
old&(5,7) = 1 : old&(6,7) = 1 : old&(7,7) = 1 : old&(7,6) = 1 : old&(6,5) =

1

REM Draw initial grid:
FOR X% = 1 TO dx%
  FOR Y% = 1 TO dy%
    IF old&(X%,Y%) GCOL 11 ELSE GCOL 4
    PLOT 69, X%*8-6, Y%*8-4
  NEXT
NEXT X%

REM Run:
GCOL 4,0
REPEAT
  FOR X% = 1 TO dx%
    FOR Y% = 1 TO dy%
      S% = old&(X%-1,Y%) + old&(X%,Y%-1) + old&(X%-1,Y%-1) + old&(X%+1,Y%-
1) + \
      \      old&(X%+1,Y%) + old&(X%,Y%+1) + old&(X%-1,Y%+1) +
old&(X%+1,Y%+1)
      O% = old&(X%,Y%)
      N% = -(S%=3 OR (O%=1 AND S%=2))
      new&(X%,Y%) = N%
      IF N%<>O% PLOT X%*8-6, Y%*8-4
    NEXT
  NEXT X%
  SWAP old&(), new&()
  WAIT 30
UNTIL FALSE

```

Apéndice V

Core Wars (P-Robots), de David Malmberg

Si usted quiere jugar Core Wars y demostrar su habilidad para competir en una lucha de “vida o muerte”, bien puede utilizar alguno de los sistemas que hay en código abierto o incluso dominio público.⁶¹

Cabe decir que estas implementaciones de software no contemplan los sistemas de 64 bits, por lo cual, en caso de tener una versión de Windows en 64 bits, para poderse ejecutar la batalla entre robots, se requerirá de usar un emulador. Para el caso de P-Robots (la versión en Pascal), se puede usar DosBox.⁶²

P-Robots es creación de David Malmberg, que escribió su primera versión en 1988. En 1994 la puso en la modalidad de código abierto y libre, es decir, ya está en el dominio público. Está escrita en Turbo Pascal (Borland) y el código de los robots es precisamente un subconjunto del lenguaje creado por Niklaus Wirth.⁶³

⁶¹ Hay una serie de implementaciones de esta idea en diversos lenguajes, C, Pascal e incluso TCL. Véase <http://corewar.co.uk/probots/>

⁶² El software de emulación x86 puede hallarse en <http://www.dosbox.com/>.

⁶³ Wirth diseñó, junto con su equipo de trabajo los lenguajes Algol W, Euler, Pascal y Modula. Posteriormente sacó una versión más refinada de este último, Modula-2, el cual terminó llamándose Oberon. Desafortunadamente la disciplina de programación en este último lo hace difícil de usar en términos prácticos. Por ejemplo, las variables del lenguaje tienen que ir en mayúsculas y esto causa muchísimos errores de compilación. Afortunadamente, con los editores de programas inteligentes (llamados *IDE*), esta dificultad ha desaparecido pues el ambiente de edición substituye las palabras reservadas cuando las ve. Wirth recibió el denominado “Nobel de la computación”, el Premio Turing, otorgado por la *Association of Computing Machinery* de los Estados Unidos, en 1984. Hoy en día el científico está jubilado (desde 1999) (http://es.wikipedia.org/wiki/Niklaus_Wirth).

En la distribución de P-Robots viene una demostración ejecutando la batalla de tres robots: Chaser, Ninja y M66. Este es el código público de cada uno de ellos:

Chaser:

```
(*****)
(*)                                     W A R N I N G                                     *)
(*)                                     *)
(*) This Robot has NOT been designed to take advantage of the advanced *)
(*) features of P-ROBOTS, such as, Shields, Fuel, Teams or Obstructions. *)
(*****)

PROCEDURE CHASER;

{

Based on a C-Robot by Kazuhiro Yabe

}

VAR
    save_dmg      : Integer;
    cur_dmg       : Integer;
    speed         : Integer;
    dir           : Integer;
    Range         : Integer;
    degree        : Integer;

PROCEDURE chkdmg;
BEGIN
    cur_dmg := damage;
    IF (cur_dmg <> save_dmg) THEN
        BEGIN
            save_dmg := cur_dmg;
            speed := 50; {maximum speed that robot can still turn}
            dir := Random(359);
        END;
    END; {chkdmg}

PROCEDURE walking;
BEGIN
    speed := 50; {maximum speed that robot can still turn}
    IF (loc_x < 200) THEN dir := 0
    ELSE IF (loc_x > 800) THEN dir := 180
    ELSE IF (loc_y < 200) THEN dir := 90
    ELSE IF (loc_y > 800) THEN dir := 270;
    drive(dir, speed);
END; {walking}

PROCEDURE shoot;
VAR return      : Boolean;
BEGIN
    return := False;
    REPEAT
        REPEAT
            Range := scan(degree, 10);
            IF (Range > 0) THEN
                IF ObjectScanned = Enemy
```

```

        THEN BEGIN
            cannon(degree, Range); {while he's there, shoot at him. }
            drive(degree, 100); {charge foe at top speed!}
            return := True;
        END;
    UNTIL Range = 0;
    IF (degree >= 360)
    THEN degree := 0
    ELSE degree := degree+20;
    walking;
    UNTIL return;
END; {shoot}

BEGIN {Chaser Main}
    speed := 50; {maximum speed that robot can still turn}
    dir := 0;
    degree := 0;
    save_dmg := damage;

    REPEAT { Until Dead or Winner }
        shoot;
        chkdmg;
        walking; { move at max speed that can still turn }
    UNTIL Dead OR Winner;

END; { end of Chaser main }

```

M66:

```

(*****
(*)                               W A R N I N G                               *)
(*)                               *)
(*) This Robot has NOT been designed to take advantage of the advanced      *)
(*) features of P-ROBOTS, such as, Shields, Fuel, Teams or Obstructions.    *)
(*****)

```

```

PROCEDURE M66;
    { Based on C-Robot M66 : programmed by OBI-ONE }

```

```

VAR
    drv_dir      : Integer; { drive direction }
    scn_dir      : Integer; { scan direction }
    step         : Integer; { scan step }
    degs         : Integer; { half of scan step }
    Range        : ARRAY[-2..2] OF Integer; { range to oponent }
    range_sv     : Integer; { range of last scan }
    found        : Boolean; { foe found ? }
    x, y         : Integer;

```

```

PROCEDURE Move;
BEGIN
    x := loc_x;
    y := loc_y;
    IF (x < 200)
    THEN drv_dir := Random(45)
    ELSE IF (x > 800) THEN drv_dir := Random(45)+180;

    IF (y < 200)
    THEN drv_dir := Random(45)+90

```

```

ELSE IF (y > 800) THEN drv_dir := Random(45)+270;

drive(drv_dir, 100);
END; {Move}

PROCEDURE attack;
VAR I, Pick : Integer;
BEGIN
  IF ObjectScanned = Enemy
  THEN cannon(scn_dir, range_sv);{shoot at last foe postion}
  Pick := 99;
  FOR I := -2 TO 2 DO
  BEGIN
    Range[I] := scan(scn_dir+step*I, degs);
    IF Range[I] > 40 THEN
    BEGIN
      Pick := I;
      I := 2;
    END;
  END;
  IF Pick <> 99 THEN
  BEGIN
    found := True;
    range_sv := Range[Pick];
    scn_dir := scn_dir+step*Pick;
    IF ObjectScanned = Enemy
    THEN cannon(scn_dir, range_sv);
  END
  ELSE scn_dir := scn_dir+120;
END; {Attack}

BEGIN {M66 Main}
drv_dir := Random(360); { initialize }
scn_dir := drv_dir+120;
found := False;
range_sv := 700;
step := 20;
degs := step DIV 2;

REPEAT { main loop }

  { drive at full speed. }
  { when close to wall, turn around at random. }

  Move;

  { scan every 20 degrees resolution. }
  { if you find a foe, then attack it with your cannon. }

  attack;

UNTIL Dead OR Winner; { end of main loop }

END; {M66 Main}

```

Ninja:

```

(*****)
(*)           W A R N I N G           (*)
(*)                                           (*)

```

```
(* This Robot has NOT been designed to take advantage of the advanced *)
(* features of P-ROBOTS, such as, Shields, Fuel, Teams or Obstructions. *)
(*****)
```

```
PROCEDURE Ninja;
```

```
{ Author unknown }
```

```
{ Based on C-Robot Ninja }
```

```
{ Locks on to a target and attacks }
```

```
VAR vector      : Integer; { current attack vector }
```

```
PROCEDURE charge(vec, Range, maxspd : Integer); { charge at an enemy }
{ vec : attack vector to use }
{ range : range to target }
{ maxspd : maximum speed to use }
BEGIN
  IF (Range > 40) AND (Range < 800) THEN { good shooting range }
    IF ObjectScanned = Enemy
      THEN cannon(vec, Range); { fire! }
    drive(vec, maxspd); { charge! }
  END; { charge }
```

```
FUNCTION Pin(vec : Integer) : Integer; { pin down a target }
{ vec : initial vector }
```

```
VAR
```

```
  tv      : Integer; { trial vector }
```

```
  ts      : Integer; { trial scan results }
```

```
  n       : Integer; { index }
```

```
  return  : Integer;
```

```
BEGIN
```

```
  tv := vec-10; { coarse screen }
```

```
  n := 2;
```

```
  ts := scan(tv, 10);
```

```
  WHILE (n > 0) AND (ts = 0) DO
```

```
    BEGIN
```

```
      tv := tv+20;
```

```
      n := n-1;
```

```
      ts := scan(tv, 10);
```

```
    END;
```

```
  IF ts = 0
```

```
  THEN return := 0
```

```
  ELSE BEGIN
```

```
    charge(tv, ts, 50);
```

```
    tv := tv-10; { medium screen }
```

```
    n := 4;
```

```
    ts := scan(tv, 5);
```

```
    WHILE (n > 0) AND (ts = 0) DO
```

```
      BEGIN
```

```
        tv := tv+10;
```

```
        n := n-1;
```

```
        ts := scan(tv, 5);
```

```
      END;
```

```
    IF ts = 0
```

```
    THEN return := 0
```

```
    ELSE BEGIN
```

```
      charge(tv, ts, 50);
```

```

    tv := tv-4; { fine screen }
    n := 3;
    ts := scan(tv, 2);
    WHILE (n > 0) AND (ts = 0) DO
        BEGIN
            tv := tv-4;
            n := n-1;
            ts := scan(tv, 2);
        END;

    IF ts = 0
    THEN return := 0
    ELSE BEGIN { found it! }
        return := ts; { say how far away it is }
        vector := tv;
    END;
    END; { fine screen }
END; { medium screen }

    Pin := return;
END; { Pin }

PROCEDURE attack;
VAR
    Range          : Integer; { range to locked target }
    return          : Boolean;
BEGIN
    return := False;
    REPEAT
        Range := scan(vector, 0);
        IF (Range > 0)
        THEN charge(vector, Range, 100) { got him! }
        ELSE IF Pin(vector) = 0 THEN { lost him! }
            return := True; { can't find him }
        UNTIL return;
    END; { Attack }

FUNCTION Find  : Integer; { find a new target }
VAR
    off           : Integer; { offset }
    dir           : Boolean; { direction }
    tv            : Integer; { trial vector }
    return        : Integer;
BEGIN {Find}
    off := 180; { half circle sweep }
    dir := False; { counter-clockwise first }
    tv := (vector+180) MOD 360; { look behind us first }
    return := 0; {no target found - default}

    WHILE (off >= 0) AND (return = 0) DO { full scan }
        BEGIN
            IF (scan(tv, 10) > 0) THEN { see anyone? }
                return := Pin(tv); { nail him! }
            dir := NOT dir;
            IF dir { alternate sides }
            THEN BEGIN
                off := off-20;
                tv := vector+off;
            END

```



```

        ELSE tv := vector-off+360;
        tv := tv MOD 360;
    END;

    Find := return;
END; {Find}

BEGIN {Ninja Main}
    vector := Random(360);
    REPEAT
        IF (speed = 0) THEN drive(vector, 30)
        ELSE IF (Find > 0)
            THEN attack { if we see anyone attack }
            ELSE vector := Random(360);
        UNTIL Dead OR Winner;
    END; {Ninja Main}

```

Para ejecutar la batalla de robots, hay que cargarlos en el entorno, en la arena donde se desarrolla la lucha. Esto puede hacerse, desde el *prompt* de MsDOS (si está usando DosBox), escribiendo:

P-ROBOTS CHASER M66 NINJA [Enter].

Acto seguido verá a los robots dispararse y moverse en la arena. Ganará el que quede como sobreviviente después de aceptar y recibir los disparos de los oponentes y propios, respectivamente. No siempre gana el mismo robot pero si los ponemos a luchar diez veces, aparentemente M66 gana con más frecuencia.

El ejercicio de programación suele ser divertido, sobre todo cuando nuestro código es quien lucha contra el de los rivales. Esto, aparte de ser como un juego entretenido, tiene la virtud de enseñar algunos trucos de programación en Pascal.⁶⁴

⁶⁴ La documentación completa sobre el subconjunto del lenguaje Pascal puede verse aquí: <http://corewar.co.uk/probots/p-robo4.txt>.

Apéndice VI

Usando los autómatas celulares para estudiar la Conjetura de Collatz

Hay una clase de problemas en aritmética que pueden ser asociados a un problema computacional de ciclos, iteraciones, “loops” o “bucles”. La idea es generar una serie de enteros de acuerdo a cierta regla. Se pregunta entonces uno, si la serie acabará entrando en uno o más bucles, en los que un conjunto de enteros se va repitiendo periódicamente. Y aunque esto suene difícil de comprender, veámoslo con la “conjetura de Collatz”:

Tómese algún número entero positivo. Divídase entre dos si es par; si es impar, multiplíquese por tres y sumémosle uno al producto. Si aplicamos este procedimiento repetidamente, eventualmente llegaremos a uno. Si esto ocurre, diremos que el número inicial con el que empezamos la secuencia es maravilloso.

Por ejemplo, consideremos el número 12. Como es par, dividámoslo entre 2. El resultado es 6. Como este es par, dividámoslo de nuevo entre dos. El resultado es 3. Este último entero es impar, por lo que procedemos a multiplicarlo por tres y al producto sumarle uno. Hallamos que esto da 10. Ahora bien, como 10 es par, dividimos entre 2. Esto nos da 5, que al ser impar, multiplicamos por 3 y le sumamos uno, lo cual da 16. El 16 lo dividimos entre 2 y nos da 8. Siendo par 8, dividimos entre dos y da 4. Este 4 dividido entre 2 es 2 y como éste resultado es par, dividimos entre 2 de nuevo y nos da 1. Por ende, el 12 es un número maravilloso (aunque yo preferiría llamarle número de Collatz, pues fue quien propuso este problema).

La pregunta fundamental de esta conjetura es la siguiente: Dado cualquier número entero positivo y utilizando el procedimiento descrito, ¿se caerá en la secuencia cíclica 2, 1, 4, 2, 1, 4,...? Nadie hasta ahora ha podido demostrar que esto ocurra forzosamente. Nadie ha podido, sin embargo, hallar un contraejemplo. A este problema se le denomina también el problema $3x + 1$, el cual se resiste a los esfuerzos por resolverlo.⁶⁵

Por ese entonces, un grupo del laboratorio de inteligencia artificial del MIT puso a prueba a través de la computadora dicha conjetura, y se halló que todos los enteros positivos hasta el 60,000,000, terminan en el ciclo 4, 2, 1... (es decir, todos esos números califican como maravillosos), pues no se encontró una sola excepción. También se halló que si la regla $3n + 1$ utilizada cuando es impar se reemplaza por $3n - 1$, el resultado, en valores absolutos, es el mismo que si se comenzase con un número entero negativo y se siguiese la antigua regla. En este caso se descubrió que todos los enteros negativos hasta -100,000,000 caían en uno de estos tres bucles:

2, 1, 2, ...

5, 14, 7, 20, 10, 5, ...

17, 50, 25, 74, 37, 110, 55, 164, 82, 41, 122, 61, 182, 91, 272, 136, 68, 34, 17, ...⁶⁶

Parece ser que a nadie se le ha ocurrido una idea feliz que permita establecer el caso general para todos los enteros no nulos (el cero pertenece, evidentemente, al

⁶⁵ Según Richard Guy, este problema fue propuesto antes de la Segunda Guerra Mundial por Lothar Collatz, matemático de la Universidad de Hamburgo (que falleció en 1990), cuando era estudiante. H.S.M Coxeter, a principios de los años 70 del siglo pasado, ofreció 50 dólares por una demostración que él pudiera entender y 100 dólares por un contraejemplo. Tal fue la cantidad de demostraciones erróneas o falsas, que Coxeter dijo no estar ya dispuesto a evaluarlas. Paul Erdős, uno de los más extraordinarios matemáticos de todos los tiempos, en 1982 expresó su opinión de que *si la conjetura es verdadera, la teoría de números carece hoy de instrumental para demostrarla*.

⁶⁶ Los investigadores Michael Beemer, William Gosper y Rich Schroepel dan estos resultados en HACKMEM (abreviatura de Hacker Memo), #239, MIT 1972.

bucle $0,0,0,\dots$). Nadie sabe tampoco si hay enteros que generen sucesiones divergentes hacia infinito carentes de bucle. Vamos, hay muchas incógnitas aún. Por ello, si se busca un contraejemplo, éste tendría que ser un número que, o bien, fuese generando números siempre mayores, sin repetir jamás ninguno, o bien, cayese en un bucle distinto al $4, 2, 1$. De existir algún número que no cumpliera con la conjetura, tendría que ser superlativamente grande, porque según Guy, la conjetura ha sido verificada en todos sus números, desde 1 hasta el 7×10^{11} .

Al poco tiempo de trabajar en el problema, se descubrió que no era necesario probar los números pares, ni tampoco los impares de la forma $4k + 1$, $16k + 3$ o $128k + 1$. De este modo, la máquina no tiene que hacer los cálculos de forma supérflua.. Evidentemente, tan pronto como una sucesión tropieza con una potencia de 2, muchas veces, tras una serie de subidas y bajadas, cae de pronto irremediablemente en la secuencia $4, 2, 1$. Por cierto, la potencia de 2 hacia la que más sucesiones convergen es 16.

Entre los números menores de 50, el más simpático y de peor comportamiento es el 27. Tras 77 pasos alcanza un tope de 9232. Bastan después 34 pasos para reducirlo a 1. Cuando el matemático John H. Conway presenta en sus lecciones la conjetura $3x + 1$, le gusta ir a la pizarra y decir: “tomemos un número al azar, el 27 por ejemplo, y veamos qué sucede”.

Pero ¿cuál es la relación de los autómatas celulares con la conjetura de Collatz? Consideremos un autómata celular unidimensional de longitud finita. Pensemos en una línea de celdas o sitios, en los cuales se pueden poner las células correspondientes. Tomemos cada uno de los enteros, del 0 al 9, como un tipo de célula. Así entonces, en cada sitio del autómata podemos poner un valor correspondiente a uno de los diez posibles valores. La regla de evolución de dicho autómata (para las siguientes generaciones del mismo), puede ser expresada en

términos de la conjetura de Collatz. Si el autómatas es par, divídase entre dos. Si el autómatas es impar, multiplíquese por tres y al producto súmele la unidad.

Lo anterior define la conjetura de Collatz como un autómatas unidimensional con $k = 9$, esto es diez posibles valores (contamos desde el cero), para cada sitio. Aquí la regla de evolución de la siguiente generación se define con respecto a la última célula en la línea del autómatas unidimensional.

Cabe señalar que sin embargo, todos las celdas pueden estar o no afectadas por la regla usada. Por ejemplo, si tenemos que el autómatas tiene una configuración impar, habrá que multiplicar por tres cada valor del autómatas y llevar el acarreo a la siguiente célula a la izquierda. Esto por sí mismo es indecidible. No podemos saber a priori hasta dónde lleva el acarreo de valores de la cifra anterior a la siguiente. Este punto me parece notable. Dicho en otras palabras, la regla de evolución puede ir de 0 hasta $n-1$. A esto le llamamos r . La r es irreducible computacionalmente en la evolución del autómatas. Si se considera que un autómatas con $k = 9$ es muy difícil de simular, considérese un autómatas con $k = 1$, en donde cada entero del autómatas que pretende simular la conjetura de Collatz está en su expresión en binario. Esto podría reducir la complejidad en la operación de división entre dos, pues en binario es simplemente recorrer al autómatas a la derecha. Aún así, no queda muy claro qué más pasos deben darse para evitar la simulación explícita.

Así entonces, si la r es irreducible computacionalmente, estamos frente a la indecidibilidad del fenómeno descrito, el cual sólo puede ser expresado en términos de la simulación explícita. En otras palabras, la conjetura de Collatz se ha transformado en averiguar si el autómatas que la describe puede ser computacionalmente irreducible o no. Hay pues que analizar los hechos que se presentan en la simulación del fenómeno. En el caso del número 27, el comportamiento del autómatas es caótico y para casos como las potencias de 2, el resultado es elemental y reducible computacionalmente de manera trivial. Esto

quiere decir que para números (o líneas de autómatas con $k = 9$), de la forma 2^n , se requieren $n-1$ pasos para llegar a 1. No obstante esto, para ciertos números impares o pares distintos a potencias de dos, el problema parece ser indecidible.⁶⁷

Podemos ver pues que la conjetura de Collatz puede representarse como un autómata celular unidimensional elemental, en donde aparentemente la regla de evolución del mismo no es reducible computacionalmente para ciertos casos y en consecuencia, el teorema de Gödel apoya la indecidibilidad de la conjetura, la cual solamente puede ser, en principio, simulada explícitamente. La consecuencia directa es que no puede existir una demostración algebraica o analítica del problema en cuestión. Esto implica que la conjetura de Collatz es indecidible y computacionalmente irreducible. Con esto en mente, y afirmado por el teorema de Gödel, la conjetura debe ser verdadera, es decir, todos los números tienen la propiedad de ser maravillosos. Si no fuese así, sería fácil dar un contraejemplo para demostrar que la conjetura es falsa, pero nadie a la fecha ha encontrado alguno.

Cabe pues entonces el trabajo de escribir un programa que pruebe la conjetura para cualquier número entero positivo (recuérdese que el cero cae en el ciclo 0, 0, 0, etcétera. Sin embargo, esto presenta algunas dificultades porque los máximos valores que se pueden poner en variables de tipo entero no pasan de 65,536. Por ello, se necesita una biblioteca que pueda manejar números muy grandes. Aquí se usó una de código abierto.⁶⁸

En esencia, las rutinas de esta biblioteca de números muy grandes hace las operaciones aritméticas necesarias procesando arreglos dinámicos en memoria, es decir, arreglos de dígitos, en donde las operaciones aritméticas asociadas se hacen como los seres humanos las hacemos. La diferencia es que aquí la

⁶⁷ Un problema se considera indecidible si no se puede construir un algoritmo que lleve a un resultado de sí o no.

⁶⁸ Véase http://www.delphiforfun.org/Programs/Library/big_integers.htm, escrita por Gary Darby, que al momento de escribir esto es un hombre de más de 70 años.

computadora las hace a toda velocidad y si se tiene una máquina verdaderamente poderosa, entonces los resultados pueden llegar a ser asombrosamente rápidos.



Programa para probar la conjetura de Collatz

Se pueden probar números hasta de 1000 cifras, lo cual parece ser un límite superior por demás aceptable. El programa trabaja bajo Windows y el núcleo del mismo (en Delphi 7), es éste:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i1,i2 : Tinteger;
  i3, i4: Tinteger;
begin
  i1 := TInteger.create;
  i2 := TInteger.create;
  i3 := TInteger.create;
  i4 := TInteger.create;
  memol.Lines.clear;
  Edit1.Text := Edit6.Text;
  memol.Lines.Add(Edit6.Text);
  repeat
    i1.assign(edit1.text); //número módulo 2 - edit1 mod edit4
    i2.assign(Edit4.text);
    i1.modulo(i2);
    Edit5.Text := i1.converttoDecimalString(false);
    if edit5.Text = '0' then Par := TRUE else Par := FALSE;
```

```

if Par = TRUE then //divide entre dos
begin
  i1.assign(edit1.Text);
  i1.divide (i2);
  memo1.lines.add(i1.converttoDecimalString(false));
  Edit1.Text := i1.ConvertToDecimalString(false);
end;

if Par = FALSE then //multiplica por 3 y suma 1
begin
  i1.assign(edit1.Text);
  i3.assign(edit2.Text); //tomamos el valor 3
  i1.Mult (i3);
  Edit1.Text := i1.ConvertToDecimalString(false);
  i4.assign(edit3.Text); //ponemos uno para sumar
  i1.add (i4);
  Memo1.lines.Add(i1.ConvertToDecimalString(false));
  Edit1.Text := i1.ConvertToDecimalString(false);
end;

until Edit1.Text = '1';

i1.free;
i2.free;
i3.free;
i4.free;
end;

```

Una idea interesante era asignar colores a los números y ver si el autómata celular (las cifras a probar), podían tener algún patrón específico o generar quizás un comportamiento caótico o bien incluso un atractor. Se hicieron las modificaciones del caso y se halló que no existe ningún patrón evidente. He aquí el resultado de ello (con las últimas mil iteraciones):



La conjetura de Collatz usando colores en lugar de números

Esto parece sugerir que en los autómatas celulares, para que existan comportamientos emergentes, requiere que las reglas ciegas no funcionen en todos los organismos del autómata, sino en una vecindad de ellos nada más. Esto es una especulación, la cual no ha sido demostrada.

Apéndice VII

El algoritmo genético

John Holland es el creador del algoritmo genético, una de las ideas más interesantes en lo que se refiere a vida artificial. Este algoritmo se basa en los siguientes factores: 1. la evolución biológica, en el mejor estilo darwiniano y 2. en su base genética. La idea de Holland es pues ejecutar este algoritmo, que no es otra cosa que una serie de pasos que conforman un programa, sobre una población de organismos, llámense estos a seres ficticios con una carga ficticia también determinada. En el algoritmo genético hay muchas acciones que son al azar, como en la evolución, por ejemplo, mutaciones o combinaciones entre genes de los individuos en cuestión. Igualmente, hay un criterio de “selección”, en donde se establecen las normas para definir a los organismos que pueden sobrevivir, que son los más aptos. En este caso, como en el de la evolución biológica real, los organismos menos aptos son eliminados.

Si vamos a los detalles, un algoritmo genético es realmente un método de búsqueda basada en la probabilidad de que ocurra un evento. Si las condiciones iniciales del problema plantean organismos como los más aptos que son intocables, en la medida que pase el tiempo podrá entenderse que esto es un óptimo en el transcurso de la ejecución del algoritmo.

Cabe decir que los algoritmos genéticos son en general modelos, no es la vida real, y se hacen abstracciones a nivel elemental de bits para así codificar los elementos y se puedan poner a prueba hipótesis. Por ejemplo, se toma una población de organismos, codificando la información de cada uno de ellos como

una cadena de bits a los que se les llama genéricamente “cromosomas”. A estos símbolos que forman la cadena se les llaman entonces “genes”. Los organismos evolucionan a través de las corridas de generaciones de individuos (iteraciones) y en cada generación, los cromosomas se evalúan a partir de alguna medida definida que describa qué tan aptos son. Las siguientes generaciones (con nuevos cromosomas), se generan a partir de lo que se denominan operadores genéticos, los cuales se definen en base a selección, mutación, reemplazo y cruzamiento.

Un algoritmo genético sigue, en general, los siguientes pasos. Es importante destacar que las variaciones en los resultados se deben a los operadores genéticos ya mencionados, que pueden hacer crecer o disminuir a las poblaciones incluso:

- **Inicialización:** Se crea una población de individuos al azar. Esta población tiene un conjunto de cromosomas, los cuales son la representación de una posible solución al problema.
- **Evaluación:** A cada cromosoma de cada individuo de una generación, se le aplicará una función que defina qué tan apto es, lo cual es equivalente a decir que mientras más apto, más se acerca a la solución.
- **Condición terminal:** El algoritmo genético debe tener manera de detenerse y esto ocurre cuando se alcanza una solución óptima. La dificultad es que no sabemos a priori cuál es esa solución óptima, por lo que se usan dos posibles criterios para detener el programa:

1. Ejecutar el algoritmo por una cantidad finita y definida de generaciones
2. detenerlo cuando no haya ya población.

Mientras no se cumplan estas condiciones, se ejecutan las tareas de “selección” (los cromosomas más aptos tienen más probabilidad de ser seleccionados); “recombinación o cruzamiento”, que es realmente el principal operador genético.

Aquí puede representarse entre otros, la reproducción sexual, que opera sobre dos cromosomas a la vez y genera dos descendientes donde además, y esto es muy importante, se combinan las características de los cromosomas de los padres. Finalmente “reemplazo”, que es donde al haber aplicados los operadores genéticos, se crea la siguiente generación en base a los mejores individuos.

Apéndice VIII

Redes neuronales

Una red neuronal artificial (para distinguirla de la red neuronal del cerebro humano, por ejemplo), es un paradigma basado en la manera en cómo funciona el sistema nervioso de los animales. La idea original se basa sobre la concepción conocida de cómo funcionan las neuronas. En el caso de una red neuronal artificial, se trata de una serie de conexiones de neuronas que entre sí terminan por producir un estímulo en la salida,.

Las redes neuronales se definen a través de neuronas. Cada una de ellas recibe un estímulo de entrada y emite un estímulo en la salida. Esta salida viene dada por tres funciones, a saber:

- i. una “función de propagación”, la cual consiste en muchas ocasiones en simplemente la suma de cada entrada, quizás en ocasiones multiplicada por el peso de su interconexión en la red. Si el peso es positivo, entonces el estímulo se llama “excitatorio” y en caso negativo, se denomina “inhibitorio”;
- ii. una “función de activación”, la cual puede modificar la función de propagación. Puede no existir incluso.
- iii. Una “función de transferencia”, que se aplica al valor dado por la función de activación. En general esto se usa para limitar de alguna manera el valor de salida que entrega la neurona.

Las redes neuronales permiten entre otras cosas:

- el aprendizaje: en donde se le proporciona a la red los datos de entrada y lo que se espera como salida.
- la auto-organización: una red neuronal puede crear su propia representación de la información, quitando la necesidad de que el programador lo haga.
- tolerancia a errores: como la información en una red neuronal se guarda en forma redundante, es menos susceptible a fallos, incluso si se daña una parte de ésta.
- procesos en tiempo real: debido a su naturaleza, la red neuronal trabaja en paralelo (al menos idealmente).

Por ello, es interesante su uso en aplicaciones y sistemas que tienen que ver con vida artificial. El sistema *PolyWorlds* de Yaeger usa esta técnica y aunque utiliza muchos recursos de máquina, las simulaciones suelen dar mucha información.

Existen desde luego redes neuronales para todo tipo de problemas. Por ejemplo, hay redes neuronales que sirven para jugar juegos como Doom/Quake.⁶⁹

⁶⁹ Véase <http://homepages.paradise.net.nz/nickamy/neuralbot/> para una descripción de cómo este sistema puede jugar Quake, incluso con código fuente en C++.

Apéndice IX

El experimento de Fred Cohen

Fred Cohen es un especialista en virus computacionales. De hecho, su tesis doctoral trata de este tema. Cohen, en 1983, concibió un experimento para presentarlo en un seminario sobre seguridad computacional. Después de trabajar en un sistema Vax 11/750 por unas 8 horas, el “virus” estuvo listo para ser demostrado. Se pidió permiso a la universidad para hacer estos experimentos. El virus no causaría daño a los sistemas, a lo más, quizás metería una carga considerable de trabajo a los mismos, aunque nadie en ese momento lo sabía.

El 10 de noviembre de ese año empezó la demostración: la infección inicial se implantó en “vd”, un programa que despliega la estructura de los archivos en un sistema operativo Unix de forma gráfica. Para mantener el control y que las cosas no se salieran de las manos, la “infección” se tenía que producir manualmente por el operador y el sistema sólo reportaba lo que estaba haciendo.

Se hicieron cinco experimentos y el virus tenía a priori todos los permisos del sistema para poder actuar e infiltrarse. El ataque más corto duró menos de cinco minutos, pero en promedio los ataques fueron de media hora. Además del ataque se proveyó de un mecanismo para “desinfectar” los archivos supuestamente infectados. La idea era dar certeza y seguridad al experimento.

Una primera conclusión fue que los operadores de los sistemas Unix infectados, que sabían del experimento, al término del mismo decidieron cambiar algunas políticas de uso, poner prohibiciones, en lugar de resolver el problema técnico. Parece ser más fácil prohibir que tomar la decisión técnica para evitar que ocurra

la dificultad de nuevo. De hecho, después de esta demostración, los operadores y personal administrativo decidieron impedir estos experimentos sobre seguridad informática sin siquiera explicar sus motivos, a decir de Cohen.

En marzo de 1984 se comenzaron negociaciones para el desarrollo de un experimento de seguridad en la máquina *Bell-LaPadula*, implementada en una Univac 1108. El experimento se iba a llevar en unas horas después de haber pedido los permisos correspondientes pero hubo muchos contratiempos por lo que a la postre las negociaciones terminaron (favorablemente) para julio de 1984. En un período de dos semanas, se pondrían los elementos para realizar el experimento de seguridad y demostrar si era posible infectar una máquina como la mencionada a través de un prototipo.

Hubo dificultades en la implementación por la falta de experiencia en ese sistema. El programador asignado a la tarea llevaba 5 años sin tocar esa máquina y el virus creado tardaba 20 segundos en realizar una infección, cuando ésta se podría hacer en menos de un segundo. Se dejaron demasiado rastros del virus por la misma inexperiencia en la implementación, rastros que podían haber sido borrados muy fácilmente, haciendo más complicada la detección del virus. Después de 18 horas de tiempo de conexión, la primera infección se desarrolló. El virus se demostró a un grupo de 10 personas, incluyendo programadores, administradores y personal de seguridad. El virus demostró su habilidad para brincarse las barreras tradicionales de seguridad.

El virus estaba constituido por 5 líneas de lenguaje ensamblador, 200 líneas de código en lenguaje Fortran y unas 50 líneas de comandos de archivos. Se estimó que teniendo a un programador más competente, un virus mucho mejor (valga la expresión), podría haberse escrito en menos de dos semanas. Cuando se entendió la naturaleza del ataque, desarrollar un ataque específico resultó una tarea muy sencilla. Cada programador en la reunión admitió que podría construir un mejor virus en el mismo tiempo que lo hizo el programador del ataque.

Fred Cohen muestra un virus ejemplo, el cual busca archivos ejecutables no infectados y coloca una firma al inicio de los mismos. La firma es una cifra "1234567". El virus entonces verifica si hay condiciones para ser lanzado y causar el daño. Finalmente el virus se ejecuta. Cuando el usuario corre un programa infectado, lo que hace es llamar al virus para realizar otra infección. El ejemplo en pseudo-código es éste:

```
program compression-virus:=
{01234567;

subroutine infect-executable:=
{loop:file = get-random-executable-file;
  if first-line-of-file = 01234567 then goto loop;
  compress file;
  prepend compression-virus to file;
}

main-program:=
{if ask-permission then infect-executable;
  uncompress the-rest-of-this-file into tmpfile;
  run tmpfile;}70
```

El mecanismo de infección generalizado funciona de la siguiente manera:

1. El virus busca archivos en el directorio donde el sistema se encuentra en ese momento. Si hay más de un archivo, toma el primero y lo carga a memoria.
2. Carga una copia del virus a memoria
3. Abre el archivo a infectar, copia el código del virus en el lugar adecuado. Cierra el archivo infectado.
4. Carga el siguiente archivo a infectar y regresa al paso anterior. Si no hay más archivos a infectar, termina su ejecución.

Curiosamente el comportamiento de los virus no necesariamente es peligroso o indeseable. Por ejemplo, supongamos que queremos bloquear sitios web

⁷⁰ Véase <http://vxheavens.com/lib/afc01.html#p21>

inapropiados para los niños que usan la computadora. Se puede hacer un virus que esté corriendo en el sistema y bloquee los sitios no deseados.

Cohen llega a la conclusión de que se puede crear un programa, un virus, que tenga el potencial de difundirse por todo el sistema. La mayoría de los sistemas, a pesar de sus medidas de seguridad, están propensos a estos ataques porque el único sistema seguro es el que está apagado. Igualmente, Cohen indica que las políticas de seguridad pueden minimizar el problema de los virus computacionales, pero que se requiere más que eso, lo cual quiere decir que hay que observar el mecanismo por el cual ocurrió un ataque y técnicamente, eliminarlo.